Universidad de Sevilla

Escuela Politécnica Superior de Sevilla

Trabajo Fin de Grado en Ingeniería Electrónica Industrial

Diseño e implementación de un módulo de Propiedad Intelectual en hardware bajo arquitectura AMBA bus para el escalado de imágenes.

Autor: Luis Miguel González Berrocal

Tutor: Carlos Jesús Jiménez Fernández

Cotutor: Laurentiu Acasandrei

Febrero de 2015

Las estrellas clavan su luz en la nada para que sean otros los que claven su mirada en ellas.

Agradecimientos

Llega el momento de asumir que no queda nada para finalizar. Aún recuerdo la corrección de un trabajo, sobre que queríamos ser de mayor, de mi profesora de Educación Plástica de 1º de ESO, en la que halagaba mis intenciones de futuro, y como finalmente mi pasión por la Electrónica ha contribuido a sembrar los cimientos de este.

Detrás de todo el esfuerzo quedan las personas que han aportado, de cualquier forma, su granito de arena para hacer posible este objetivo.

A quienes primero dedico este trabajo y agradezco su apoyo y esfuerzo es a mis padres, sin los cuales afrontar la carrera universitaria hubiera sido imposible. A mi madre por soportar las incontables "cruzadas" con mis hermanos y a mi padre por las tonterías de niño chico que tiene en la cabeza. Igualmente, a mi hermana, protagonista de la gran mayoría de esas riñas y al enano, por su entusiasmo al verme cuando vuelvo al pueblo; a ambos, poned el mismo empeño en los estudios para aparecer yo en los agradecimientos de vuestros trabajos.

A mis abuelos, Concepción y Miguel, este trabajo va especialmente dedicado a vosotros por no estar presentes, os quiero. Y a todos los demás familiares, tíos y primos, gracias por todo.

A mi tutor Laurentiu, por la confianza depositada en mí, las alegrías al ver funcionar las cosas y las risas cuando no, sin él este trabajo hubiera resultado una tarea muy dura. Y como no, a los demás profesores que durante los cuatro años de carrera han contribuido a la consecución de los objetivos de esta.

A mis amigos del pueblo y los que en verano vienen a pasar allí sus vacaciones, que acompañaron mi infancia y adolescencia, gracias por los buenos momentos y como no, las rutas en bici.

Por último, a mis andaluces favoritos, quienes hicieron que estos años pasasen volando y dejaran su huella en mí. Como olvidar las madrugadas en "el Bunker" de Reina Mercedes, las innumerables discusiones para hacerme entrar en razón, un abrazo…, en definitiva, la suerte de conoceros, y las fiestas, risas y alegrías que formaron parte del camino para conseguir lo que hoy agradezco aquí a todos.

Y no podría olvidarme de los chicos y chicas del equipo de vóley de Farmacia, cualquier día pasaba a ser un poco mejor si lo terminaba junto a ellos con las rodillas magulladas después de los entrenamientos, gracias.

Nuevamente, muchas gracias a todos.

Olvida de noche lo que de día te atormenta. Observa a las estrellas, que de día no se dejan ver porque de noche se aprecia mejor su belleza.

Resumen

En la mayoría de los algoritmos de visión por computador, el escalado de imagen consiste en el proceso de modificar las dimensiones de una imagen digital. El escalado es un proceso no trivial que implica una compensación entre eficiencia, suavidad y nitidez, siendo un paso de preprocesamiento obligatorio en los algoritmos de visión por computador. Hoy en día, debido a las más estrictas restricciones de energía y de costes, la utilización de sistemas de visión basados en PC está obsoleta. Debido a la aparición de un nuevo campo, *Embbeded Vision* [1] los algoritmos de visión tradicional basados en PC se han aplicado en sistemas empotrados o sistemas en chip (*SoC*) que tienen un bajo consumo de energía, pequeño tamaño, movilidad y coste reducido. Un sistema empotrado o *SoC* está formado por componentes heterogéneos, más genéricamente conocidos como núcleos o módulos de Propiedad Intelectual (IP). Un ingeniero de diseño de sistemas empotrados tiene que encontrar la mejor combinación de software y hardware de IPs que satisfagan las restricciones del diseño.

El objetivo de este trabajo es el diseño y prueba de un módulo IP de escalado de imagen. El IP debe tener una interfaz de Arquitectura de Bus Avanzada para Microprocesadores (AMBA bus). El IP debe implementar el algoritmo de interpolación bilineal. Este IP se integrará en *SoC* y se va a utilizar para acelerar una aplicación de detección de rostros que actualmente utiliza una función de software para el escalado de imagen. Las dimensiones máximas de la imagen de entrada es 1920x1080 (full HD). Teniendo en cuenta el área y las restricciones temporales será responsabilidad del estudiante elegir la mejor arquitectura (paralelo, serie o mixto) para el escalado de la imagen. Asimismo, el estudiante es responsable de la creación de los controladores del dispositivo IP. Por último, las medidas de rendimiento se realizarán bajo condiciones de la vida real, cuando el IP este integrado en *SoC* y una aplicación de detección de rostros se esté ejecutando.

Habilidades requeridas: C/C ++, lenguaje HDL (VHDL o Verilog) y conocimiento básico de las FPGAs y arquitecturas de microcontroladores.

Palabras clave: IP, bus AMBA, C/C ++, VHDL, Verilog, FPGA, SoC, IP, escalado de imagen.

Abstract

In computer graphics, image scaling is the process of resizing a digital image. Scaling is a non-trivial process that involves a trade-off between efficiency, smoothness and sharpness. Image scaling is a mandatory preprocessing step in computer vision algorithm. Nowadays, due to the more stringent power and cost restrictions, using PC based vision systems is obsolete. Due to the emergence of a new field, that is "Embbeded Vision" [1] the traditional PC based vision algorithm are now implemented on embedded systems or System on Chip (SoC) that have low power consumption, small size, mobility and reduced cost. An embedded system or SoC is formed by heterogeneous components, more generically known as Intellectual Property (IP) cores or modules. An embedded system design engineer has to find the best combination, of software and hardware IPs that meet the design constraints.

The aim of this project is the design and testing of an IP for image scaling. The IP must have an Advanced Microprocessor Bus Architecture (AMBA bus) interface. The IP must implement the bilinear interpolation algorithm. This IP will be integrated in SoC and it will be used to accelerate a face detection application that at the moment uses a software function for image scaling. The maximum size of the input image is 1920x1080 (full HD). Given the area and timing constrains it will be the responsibility of the student to choose the best architecture (parallel, sequential or mixed) for image scaling. Also the student is responsible for creating the IP device software drivers. Finally the performance measurements will be done under real life conditions, when the IP is integrated in SoC and a face detection application is running.

Skill requirements: C/C++, HDL language (VHDL or Verilog) and basic knowledge of FPGA's and Microcontroller architectures.

Key Words: IP, AMBA bus, C/C++, VHDL, Verilog, FPGA, SoC, IP, Image scaling

Índice de contenidos

Índice de imágenes

Índice de tablas

1. Introducción

1.1 Antecedentes, contexto y motivación

Actualmente, el término visión por computador está adquiriendo gran importancia. El permitir la toma de decisiones a un ordenador a partir de la información que recibe, en este caso de imágenes tomadas por dispositivos digitales, facilita y disminuye en gran medida la interacción entre el hombre y el medio físico, que por condiciones de diversa índole no sea factible de ser realizada personalmente por el ser humano.

La visión por computador o visión artificial es un campo de la Inteligencia Artificial cuyo objetivo es la obtención, procesado, análisis e interpretación de imágenes digitales. A continuación se muestra un esquema de las etapas que conforman una aplicación de visión por computador.

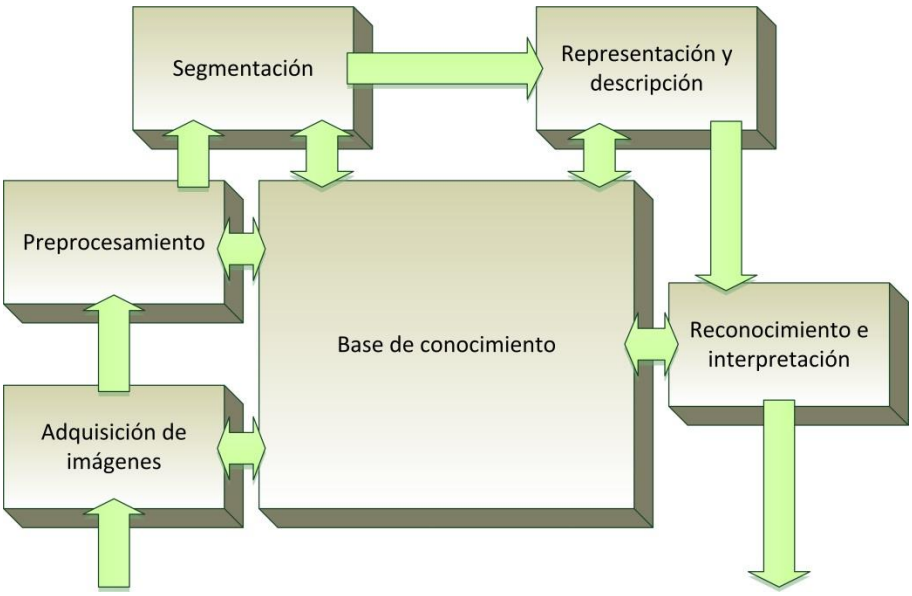


Imagen 1.1 – Etapas de una aplicación de visión artificial

Dentro de las distintas fases del preprocesamiento o procesado de imágenes, y en la cual se centra este trabajo, se encuentra el escalado de imágenes, proceso por el cual se modifican las dimensiones de la misma para pasar a ser tratada por la siguiente fase a la que esté ligada. El escalado es un proceso no trivial que implica una compensación entre eficiencia, suavidad y nitidez, siendo un paso de preprocesamiento obligatorio en los algoritmos de visión por computador.

Dado el avance tecnológico experimentado en los últimos años, se imponen más restricciones en cuanto a costes y potencia, por lo que el uso de los sistemas de visión basados en computador está quedando obsoleto. Por ello y gracias a la aparición de un nuevo campo, *Embbeded Vision* [1], los algoritmos de visión artificial pasan a ser implementados en sistemas empotrados o sistemas en chip (*System on Chip, SoC*), que combinan un bajo consumo de energía con un tamaño y coste reducido. Un sistema empotrado o *SoC* integra en un solo chip todos los componentes de un ordenador u otro sistema electrónico específico, a los que nos referimos técnicamente como módulos de Propiedad Intelectual (*Intellectual Property, IP*).

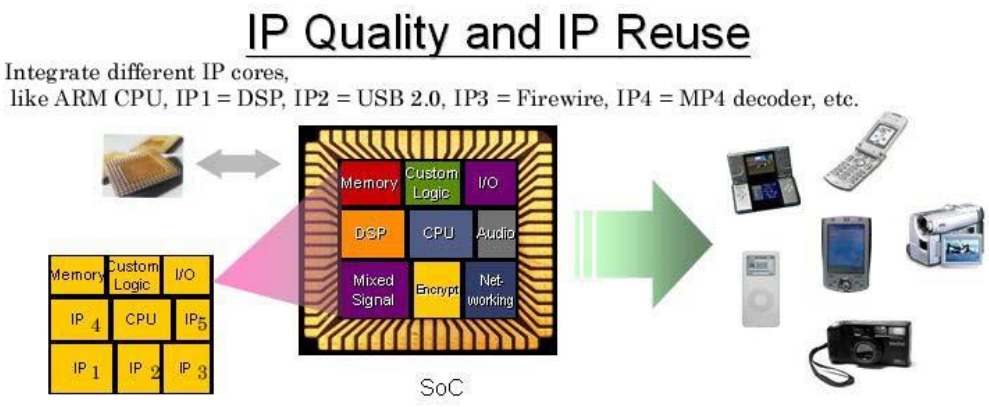


Imagen 1.2 – Modulo IP [2]

En aplicación de los conocimientos adquiridos en el transcurso del Grado en Ingeniería Electrónica Industrial cursado en la Escuela Politécnica Superior de Sevilla, en concreto aquellos relacionados con la Electrónica y el diseño digital en VHDL y la programación en C, y en colaboración con el Instituto de Microelectrónica de Sevilla, se busca encontrar la mejor combinación entre software y hardware que satisfagan las especificaciones de diseño requeridas.

1.1 Objetivos del trabajo

El objetivo principal de este trabajo es el diseño e implementación de un módulo IP en hardware para el escalado de imágenes. Este IP se integrará y se utilizará para acelerar una aplicación de detección de rostros que actualmente utiliza una función software para el escalado de imagen, presente en la biblioteca libre de visión por computador OpenCV [3].

Imagen 1.3 – Logotipo OpenCV [4]

Para nuestra aplicación, la función software ha sido adaptada y modificada para reducir la complejidad de su implementación hardware. El algoritmo que se ha implementado es el de interpolación bilineal y las dimensiones máximas de la imagen de entrada son 1920x1080 (full HD).

Matemáticamente, la interpolación bilineal se obtiene mediante una serie de interpolaciones lineales a partir de cuatro valores conocidos. Se realizan dos primeras interpolaciones en una dirección (horizontal o vertical) con cada par de valores y a continuación, con los dos valores calculados, otra interpolación más en la dirección contraria (vertical u horizontal), obteniendo así el valor en la posición buscada.

La interpolación bilineal es, por tanto, una extensión de la interpolación lineal para funciones de dos variables. Así pues, a pesar de emplearse funciones de interpolación lineal, la interpolación bilineal se comporta como una función cuadrática.

Enfocando este razonamiento desde el punto de vista de una imagen, diríamos que sería una extensión de la interpolación n*earest neighbor* (vecino más cercano), usando cuatro pixeles originales que rodean al punto deseado de interpolación (vecindad 4 diagonal).

Otros algoritmos de interpolación son la interpolación bicúbica, interpolación en escalera, S-Spline, interpolación de Lanczos, de Mitchell, interpolación Genuine Fractals, etc.

El módulo IP deberá ser compatible con la Arquitectura de Bus Avanzada para Microcontroladores ABMA bus (*Advanced Microcontroller Bus Architecture*). AMBA bus [5] es un estándar abierto para la conexión y gestión de bloques funcionales (*IP Cores*) en los diseños *SoC*. Facilita el desarrollo de diseños con varios procesadores y gran número de controladores y periféricos. La especificación AMBA 2.0 define dos protocolos de interfaz:

- AMBA 2.0 AHB Interface: interfaz que permite la interconexión de alta eficiencia entre maestros en un sistema de frecuencia única.
- AMBA 2.0 APB Interface: interfaz compatible con las transferencias de bajo ancho de banda necesarias para acceder a los registros de configuración de los periféricos y el tráfico de datos a través de los periféricos de bajo ancho de banda.

Respecto al *SoC*, se ha escogido un sistema de alto rendimiento y código libre (bajo licencia GNU GPL) que permite la integración del módulo IP, el procesador LEON3 desarrollado por Aeroflex Gaisler [6]. El LEON3 es un modelo VHDL sintetizable de un procesador de 32 bits compatible con la arquitectura SPARC V8. Es un modelo altamente configurable y especialmente adecuado para diseños en sistemas *SoC*.

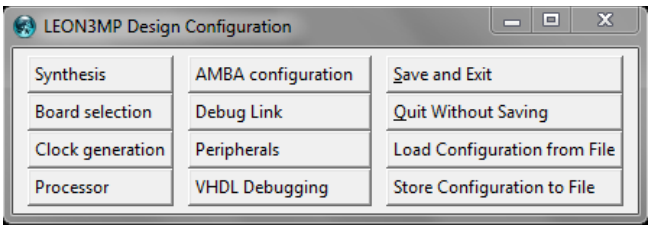


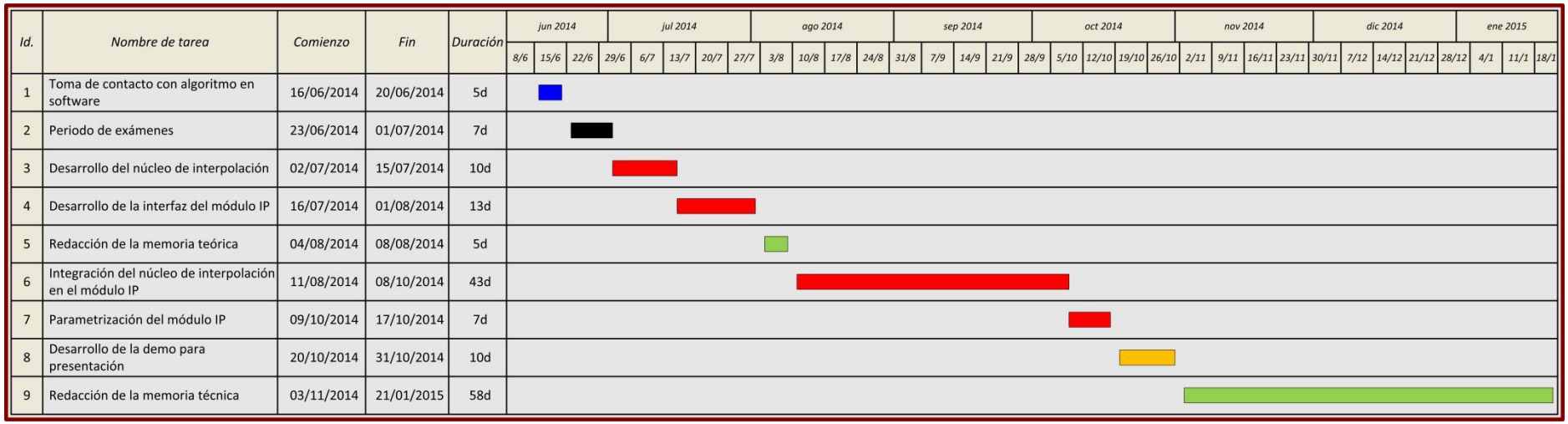
Imagen 1.4 – Herramienta Xconfig para la configuración del procesador LEON3

Como segundo objetivo asociado al objetivo principal del trabajo, se plantea la ejecución en tiempo real de una aplicación de escalado de imágenes con el módulo IP integrado en la plataforma *Virtex 5 FPGA Evaluation Platform ML505*.

1.2 Desarrollo temporal del trabajo

El trabajo se ha realizado desde Junio hasta Enero. En el diagrama que se muestra a continuación no se han contabilizado los fines de semana, pero si se ha realizado trabajo durante casi todos estos (en el Capítulo 7 se detallarán estos aspectos).

Las tareas en azul se corresponden al analisis del proyecto y distribucion de requerimientos. En negro se han representado los periodos en los que no se realizaron tareas en relacion con el trabajo. Las tareas relacionadas con el desarrollo del código del módulo IP estan representadas en rojo y en naranja las del código de la demo para la presentación. Finalmente, en verde se representan las tareas de redaccion de la presente memoria.



Id.	Nombre de tarea	Comienzo	Fin	Duración
1	Toma de contacto con algoritmo en software	16/06/2014	20/06/2014	5d
2	Periodo de exámenes	23/06/2014	01/07/2014	7d
3	Desarrollo del núcleo de interpolación	02/07/2014	15/07/2014	10d
4	Desarrollo de la interfaz del módulo IP	16/07/2014	01/08/2014	13d
5	Redacción de la memoria teórica	04/08/2014	08/08/2014	5d
6	Integración del núcleo de interpolación en el módulo IP	11/08/2014	08/10/2014	43d
7	Parametrización del módulo IP	09/10/2014	17/10/2014	7d
8	Desarrollo de la demo para presentación	20/10/2014	31/10/2014	10d
9	Redacción de la memoria técnica	03/11/2014	21/01/2015	58d

Imagen 1.5 – Diagrama de Gantt

1.3 Estructura de la memoria

El presente documento se encuentra estructurado en ocho capítulos, que pretenden ofrecer un conocimiento del estudio realizado, desde los conceptos más generales hasta la descripción de los aspectos más concretos del objetivo del trabajo, para terminar con los resultados experimentales y conclusiones junto con un apartado del presupuesto generado. Con objeto de facilitar la lectura del trabajo, se expone a continuación una descripción del contenido de cada capítulo.

- Capítulo 1: Introducción. Breve capitulo introductorio donde se aborda la problemática presente, se indican los objetivos del trabajo, así como la organización y desarrollo del mismo.
- Capítulo 2: Estado del arte. Capítulo dedicado al estudio teórico de la problemática de la visión por computador y el desarrollo de módulos IP.
- Capítulo 3: Preprocesamiento de imágenes. Aquí se realiza una recopilación de las técnicas de preprocesamiento de imágenes más empleadas, finalizando el capítulo con los aspectos concernientes al objeto del trabajo: el escalado de imágenes.
- Capítulo 4: Diseño en software. Capítulo en el que se describe el funcionamiento del algoritmo de interpolación bilineal implementado en software.
- Capítulo 5: Diseño en hardware. En este capítulo se realiza una introducción al lenguaje de diseño de circuitos VHDL, tras lo cual se procede a la descripción del funcionamiento del diseño del módulo IP, primero a nivel de componentes y después a nivel funcional (flujo de datos).
- Capítulo 6: Resultados. Capítulo en el que se presentan los resultados obtenidos de la síntesis del diseño y del funcionamiento en tiempo real, comparándolo con la ejecución en software.
- Capítulo 7: Presupuesto. Este capítulo contiene una valoración del presupuesto generado tras la ejecución del presente trabajo.
- Capítulo 8: Conclusiones y líneas de trabajos futuros. Para finalizar, se presenta un breve capitulo en el que se analizan las conclusiones obtenidas durante la ejecución del trabajo y se exponen los puntos donde se puede seguir trabajando con el presente diseño.

2. Estado del arte

2.1 Introducción

Como seres humanos, percibimos la estructura tridimensional del mundo que nos rodea con aparente facilidad. Se estima que entre el 60 y 70% de la información que procesa el cerebro humano es visual, parece lógico pues, que el objetivo de dotar a las maquinas del sentido de la vista supondrá un salto cualitativo en sus capacidades de actuación.

Al igual que le sucede al hombre, las maquinas "necesitan ver" para adquirir información y aprender de su entorno para realizar operaciones análogas a las que efectúa el ser humano.

Es increíble como los animales y el ser humano son capaces de procesar esta información, sin embargo, y a pesar de los avances, los algoritmos de visión por computador son altamente propensos a errores y la capacidad de procesamiento, comparada con la humana, sigue siendo reducida.

Esta elevada tasa de fallos se debe en parte, a que la visión artificial es un problema inverso. Tratamos de obtener información que no es inherente a la vista, es decir, existen incógnitas y en consecuencia, diversas soluciones posibles, dada la insuficiente información proporcionada por las imágenes. Por tanto, debemos recurrir a modelos físicos y probabilísticos para eliminar la ambigüedad entre las posibles soluciones.

Reducida la tasa de fallos en los algoritmos de visión artificial nos queda conseguir que la ejecución de estos se asemeje, en términos temporales, a la capacidad de procesado de la misma información por el ser humano. La tecnología nos permite en gran medida alcanzar este objetivo, pero el afán humano no se queda ahí. El siguiente desafío pasa por conseguir una integración y consumo reducidos, y es aquí donde entra en juego el diseño de módulos IP. Los módulos IP son unidades reutilizables de lógica que se usan en la fabricación de circuitos integrados para aplicaciones específicas (ASIC) o en diseños lógicos para FPGA.

En los siguientes apartados tratamos un poco más a fondo la visión por computador y finalizamos el capítulo hablando del diseño de los módulos IP.

2.2 Visión por computador

La visión por computador o visión artificial es una rama de la inteligencia artificial que tiene por objetivo modelar matemáticamente los procesos de percepción visual en los seres vivos y generar programas que permitan simular estas capacidades visuales por computadora [7], [8].

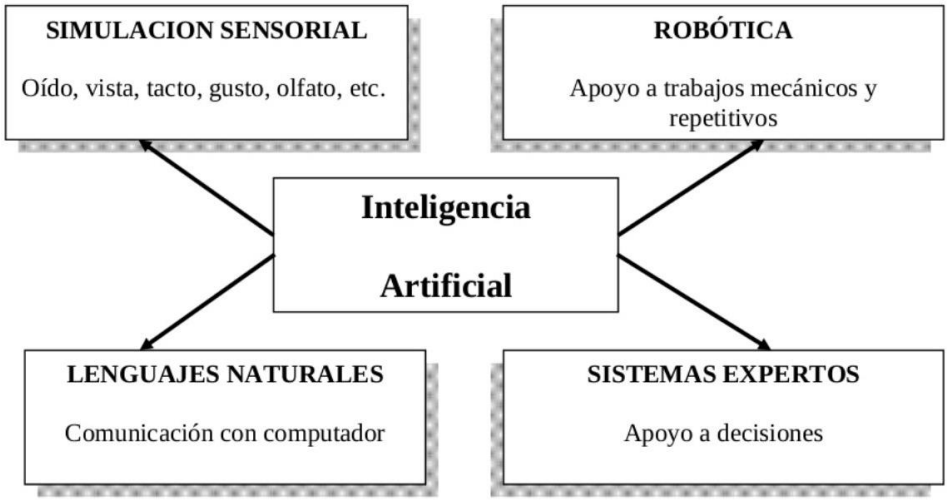


Imagen 2.1 – Campos de la inteligencia artificial [9]

Es el conjunto de todas aquellas técnicas y modelos que nos permiten la adquisición, procesamiento, análisis y explicación de cualquier tipo de información espacial del mundo real obtenida a través de imágenes digitales, que proporcionan de alguna forma el sentido de la vista a una máquina.

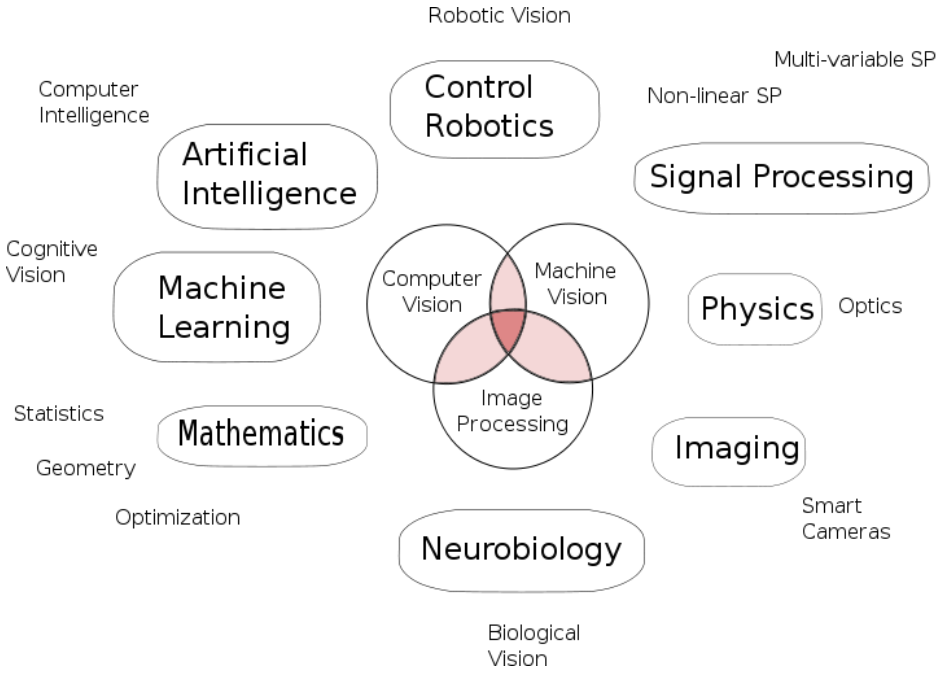


Imagen 2.2 – Relación entre la visión por computador y otros campos [10]

La visión artificial pretende capturar la información visual del entorno físico para extraer características visuales relevantes, utilizando procedimientos automáticos. Se trata de un objetivo ambicioso y complejo que actualmente se encuentra en una etapa primitiva.

2.2.1 Componentes de un sistema de visión

Los elementos hardware mínimos necesarios para un sistema de visión artificial son los siguientes:

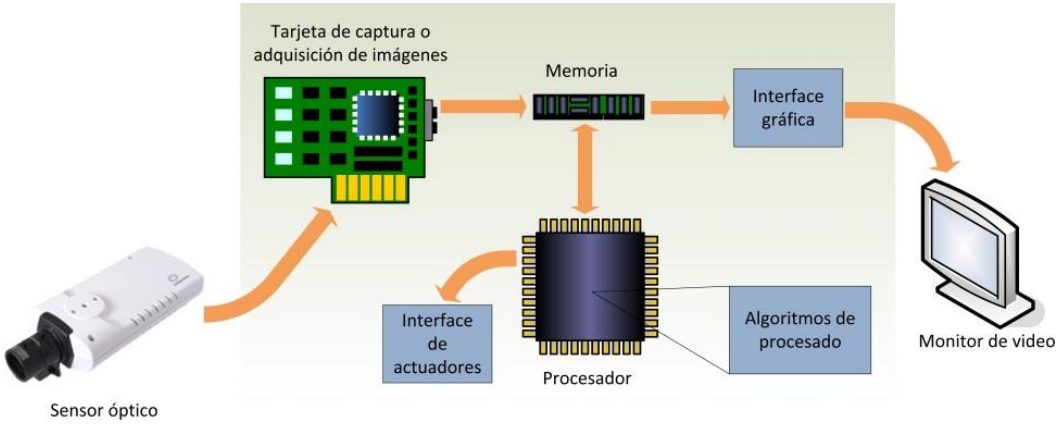


Imagen 2.3 – Elementos hardware de un sistema de visión artificial

- Sensor óptico:
  Conjunto encargado de recoger las características del objeto estudiado y proporcionar los datos para su procesado, por medio de una imagen digital. El tipo de sensor, su tamaño y la resolución deben escogerse en función de los elementos que se desea ver. El sensor puede ser una cámara color o monocromo que produce una imagen completa de la escena o también un escáner, que produce una línea en cada instante.
- Tarjeta de captura o adquisición de imágenes:
  Es la interfaz entre el sensor y el computador o módulo de procesado que permite al mismo disponer de la información capturada por el sensor de imagen. La imagen de entrada se divide en pixeles formando filas y columnas que abarcan toda la zona de la imagen y representan los niveles de gris en una imagen monocromática o la codificación de color en una imagen a color.
- Memoria:
  Dispositivo en el que se almacena la imagen digitalizada para estar disponible para el ordenador o equipo de procesamiento.
- CPU:
  Es el sistema que analiza las imágenes recibidas para extraer la información de interés en cada uno de los casos, implementando y ejecutando los algoritmos diseñados para la consecución de los objetivos.
  El procesado puede ser realizado por un ordenador de propósito general, o para obtener mejor rendimiento en esta etapa, por un sistema empotrado específico para el procesamiento de imágenes.
- Monitor de video:
  Permite visualizar tanto las imágenes o escenas captadas como los resultados del procesamiento de dichas imágenes.

2.2.2 Etapas de un sistema de visión artificial

Aunque cada aplicación de visión artificial tiene sus características específicas, existe una estructura de etapas comunes entre ellas, y no necesariamente deben cubrirse todas en una implementación concreta [11], [12].

La siguiente división pretende agrupar las distintas etapas de un sistema de visión artificial en tres niveles, en las que a medida que se avanza de un nivel a otro, se ve reducida la cantidad de información a procesar:

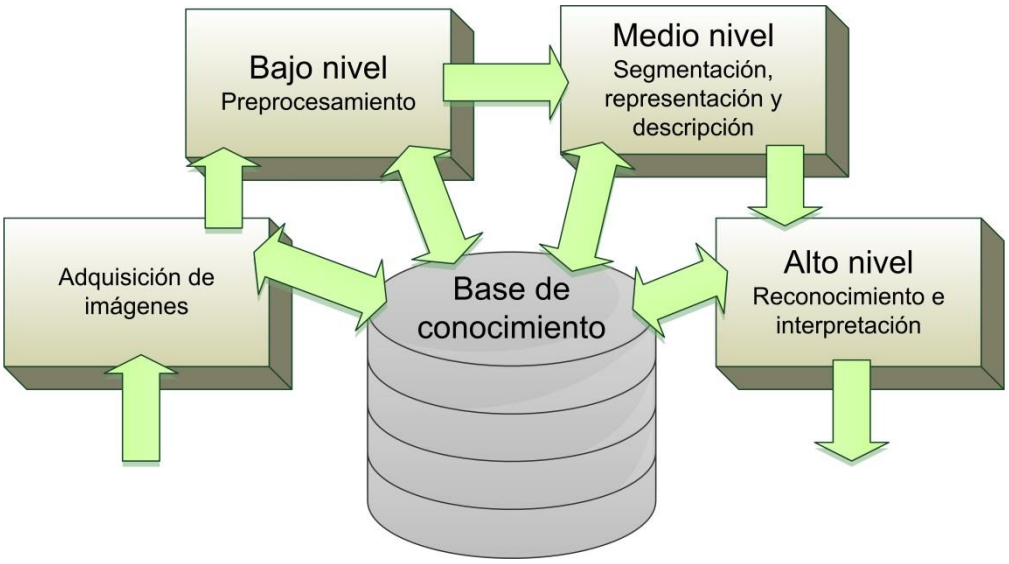


Imagen 2.4 – Etapas de un sistema de visión artificial

- Bajo nivel:
  Comprende principalmente la etapa de preprocesamiento, aunque también se suele incluir en este nivel la etapa de adquisición de imágenes. Aquí se ejecutan algoritmos y técnicas de filtrado, restauración de la imagen, realce, extracción de contornos, etc.
- Nivel medio:
  Comprende las etapas de segmentación, representación y descripción. Se trata de etapas no interpretativas que buscan conseguir descripciones de las imágenes a mayor nivel que las que proporcionan los pixeles por separado. Incluye algoritmos de segmentación de áreas, extracción de características y etiquetado de estas, etc.
- Alto nivel:
  Comprende las etapas de reconocimiento e interpretación, etapas próximas a la inteligencia artificial. Se emplean algoritmos de redes neuronales, lógica difusa y detección de formas, análisis geométrico, medición de objetos, etc.

A continuación se describen brevemente cada una de las etapas establecidas para un sistema de visión artificial:

- Adquisición de imágenes:
  Es la etapa en la cual se capturan las imágenes. El objetivo es obtener mediante técnicas fotográficas, las características visuales de los objetos. Es quizás la etapa más importante para el desarrollo del proceso, pues una buena adquisición de imágenes posibilitará enormemente la consecución de los objetivos de las siguientes etapas.

- Preprocesamiento:
  Esta etapa consiste en el tratamiento digital de las imágenes obtenidas en la etapa de adquisición. El objetivo es mejorar la "calidad informativa" de la imagen adquirida y facilitar la búsqueda de información en ella.
  Este tema se trata en el siguiente capítulo con mayor profundidad debido a que es donde se enclava el objeto del trabajo.
- Segmentación:
  En esta etapa se divide la imagen en unidades o áreas para realizar la extracción de información, es decir, se divide la imagen en partes con significado (se decide que partes de la imagen pasan a ser procesadas en las etapas posteriores y cuales no). Estas partes son homogéneas respecto a ciertas características como pueden ser el color, la textura o la intensidad.
- Representación:
  Esta etapa se encarga de realizar una parametrización de las partes obtenidas en la etapa de segmentación.
- Descripción:
  En esta etapa se busca extraer del objeto de estudio, características que lo diferencian de los demás. Para ello se deben extraer las características invariantes, es decir, independientes a rotación, escalas, excentricidad, área, perímetro, etc.
- Reconocimiento:
  Esta es la penúltima etapa, en la cual se realiza la clasificación de los objetos de la imagen a partir de los descriptores obtenidos en la etapa anterior.
- Interpretación:
  En la última etapa se busca proporcionar un significado al conjunto de objetos reconocidos en una imagen.

La base de conocimientos codifica el conocimiento sobre un dominio del problema en un sistema de procesamiento de imágenes. Este conocimiento detalla las regiones de una imagen donde se sabe que se ubica información de interés, limitando así la búsqueda que ha de realizarse para hallar tal información.

2.3 Intellectual Property Core

A lo largo de la breve historia de la industria de los semiconductores, la innovación y el crecimiento se han visto impulsados por la rápida evolución tecnológica. Factores tales como la optimización del tamaño de los circuitos integrados, el rendimiento, costes, la reducción del consumo de energía, la flexibilidad y capacidad de programación han dado lugar a la aparición de empresas dedicadas al desarrollo de componentes que han proliferado rápidamente en una gama muy diversa de productos de consumo [13], [14].

Como la tecnología permite la integración de miles de millones de transistores en un solo chip, es inconcebible construir nuevos chips desde cero. En lugar de ello, los diseñadores disponen de grandes bibliotecas de componentes o módulos IP a partir de los cuales construyen los nuevos chips combinando, modificando y complementando los diseños anteriores. Así pues, se pueden combinar gran cantidad de módulos en un solo chip para crear circuitos integrados de aplicación específica (*ASIC*), productos estándar de aplicación específica (*ASSP*) y sistemas en chip (*SoC*). A su vez proporcionan la base para productos tales como teléfonos móviles,

cámaras digitales, reproductores MP3, controladores de procesos industriales, tarjetas inteligentes, y básicamente todo lo que utiliza o procesa información y datos.

Los bloques semiconductores de propiedad intelectual, más conocidos como módulos IP o *IP Cores*, son componentes de diseño lógico reutilizables que se usan en el desarrollo de circuitos integrados. La denominación IP hace referencia a la disponibilidad de su diseño bajo condiciones de licencia (comercialización) o incluso de uso libre [15]. Con ello, las empresas pueden volver a utilizar los módulos desarrollados internamente en sus propios productos o pueden acceder a otros diseños, consiguiendo así distribuir el coste de desarrollo entre los distintos fabricantes de circuitos integrados. Como resultado, los fabricantes de chips desarrollan innovaciones más rápidamente. En la actualidad (2013), el proveedor líder a nivel mundial es ARM Holdings (43.2% de la cuota de mercado), seguido de Synopsys Inc. (13,9% de la cuota de mercado), Imagination Technologies (9% de la cuota de mercado) y Cadence Design Systems (5.1% de la cuota de mercado).

La reusabilidad de los módulos IP permite modificar el diseño de estos de acuerdo a las necesidades del problema que se plantee. Combinando esto con las tecnologías de bajo coste de implementación, se justifica su uso en el desarrollo de circuitos integrados de aplicación específica (ASIC) y en el campo de las FPGA.

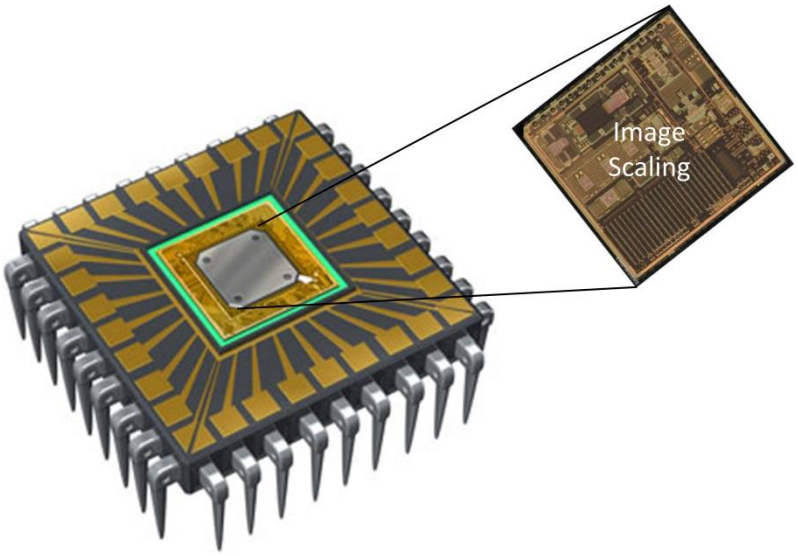


Imagen 2.5 – ASIC

Como el diseño de módulos IP a menudo requiere experiencia en el diseño microelectrónico y exige el dominio de determinadas aplicaciones, las empresas especializadas en el desarrollo de módulos IP representan un segmento altamente intensivo en conocimiento de la industria de las TIC (Tecnologías de la Información y la Comunicación).

Los módulos IP se utilizan en casi todos los nuevos diseños de chips semiconductores, como los microprocesadores programables, microcontroladores, procesadores de señales digitales (DSP), bloques de procesamiento de señal mixta analógico-digital, arquitecturas de computación configurables, etc. Estos son de vital importancia para la introducción exitosa de nuevos productos electrónicos.

Podemos hacer una distinción dentro de los módulos IP, a los cuales nos referiremos como:

- S*oft core* (núcleos blandos):
  Se denominan "núcleos blandos" debido a que permiten la modificación y síntesis del diseño original.
  Los módulos IP se ofrecen como RTL (*Register Transfer Level*, transferencia a nivel de registro) sintetizable. Los módulos sintetizables son liberados en un lenguaje de descripción de hardware como Verilog o VHDL. Estos son análogos a los lenguajes de alto nivel como C en el campo de la programación de computadoras. Los fabricantes de chips adquieren la licencia para poder modificar y comercializar estos diseños.
  A veces también se ofrecen los módulos IP como netlists genéricos a nivel de puerta. Esta descripción brinda una protección contra la ingeniería inversa que pretendería violar la propiedad intelectual.
- *Hard core* (núcleos duros):
  Se denominan "núcleos duros" debido a que el diseño no puede ser significativamente modificado (por ejemplo, solo se permite su parametrización) por el fabricante de chips.
  Por la naturaleza de su representación de bajo nivel, ofrecen una mejor previsibilidad del rendimiento de los chips en términos de tiempo y área.
  Los diseños analógicos y de lógica de señal mixta se definen generalmente en un nivel inferior, con una descripción gráfica. Así pues, estos módulos IP (PLL, DAC, ADC, PHY, etc.) se proporcionan a los fabricantes en un formato de diseño de transistores, como el GDSII.

En el siguiente capítulo nos centraremos en el procesado de imágenes y las distintas operaciones que se llevan a cabo en esta etapa y a continuación pasaremos a describir en profundidad la operación de escalado de imágenes.

3. Preprocesamiento de imágenes

Toda imagen que se adquiere por medios ópticos, electroópticos o electrónicos sufre en cierta medida los efectos de la degradación que se manifiestan en forma de ruido, perdida de definición y de fidelidad de la imagen. La degradación viene provocada por el ruido de los sensores de captura, imprecisiones en el enfoque de la cámara, movimiento de la misma o perturbaciones aleatorias, etc. Los mecanismos que tratan de contrarrestar estos efectos se incluyen dentro de la etapa de preprocesado [16].

Las dos principales causas que producen perdida de información cuando se captura una imagen son la naturaleza discreta de los pixeles de la imagen (muestreo) y el rango limitado de valores de intensidad luminosas (cuantificación) que es capaz de ofrecer el sistema de digitalización.

- Efecto del muestreo:
  El muestreo de una imagen tiene el efecto de reducir la resolución espacial de la misma. En las siguientes imágenes se muestra el efecto de captar una misma imagen a distintas resoluciones.

Imagen 3.1 – Lena original (arriba izquierda) y Lena muestreada (1/2 arriba derecha, 1/4 abajo izquierda y 1/8 abajo derecha)

- Efecto de la cuantificación:
  El efecto de cuantificación viene dado por la imposibilidad de tener un rango infinito de valores de medida para la intensidad de brillo de los pixeles. La tecnología actual permite en algunos casos llegar hasta 10 bits de información, aunque lo general es tener 8 bits, o equivalentemente 256 niveles de gris para codificar este valor lumínico. La siguiente figura muestra el efecto de representar una imagen con distintos números de bits.

Imagen 3.2 – Lena original (arriba derecha) y Lena cuantificada a diferentes niveles de gris (16 arriba derecha, 8 abajo izquierda y 2 abajo derecha)

Generalmente el preprocesado pretende reparar la imagen de los desperfectos producidos o no eliminados por el hardware. Los algoritmos de preprocesado permiten modificar la imagen para eliminar el ruido, transformarla geométricamente, mejorar la intensidad o el contraste, etc. Por supuesto, como estos algoritmos necesitan gran cantidad de tiempo de ejecución, es lógico que lo mejor sea utilizar un conveniente hardware que los acelere.

3.1 Conceptos generales

A continuación se introducen algunos conceptos relacionados con la representación de imágenes [17].

3.1.1 Pixel

La palabra píxel surge de la combinación de dos palabras inglesas comunes, *picture element*. Es el menor de los elementos de una imagen al que se puede aplicar individualmente un color o una intensidad o se puede diferenciar de los otros mediante un determinado procedimiento.

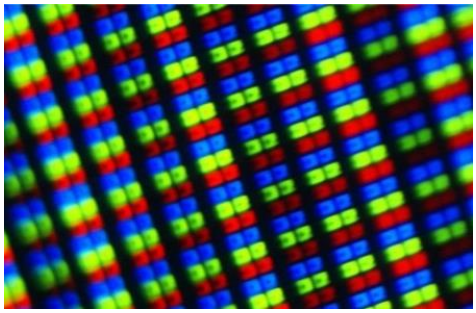


Imagen 3.3 – Densidad de pixeles de una pantalla [18]

3.1.2 Imagen digital

Una imagen digital es una representación bidimensional de una imagen a partir de una matriz numérica. Dicha matriz es una agrupación de pixeles, cada uno con un valor de intensidad o brillo asociado correspondiente a la imagen representada.

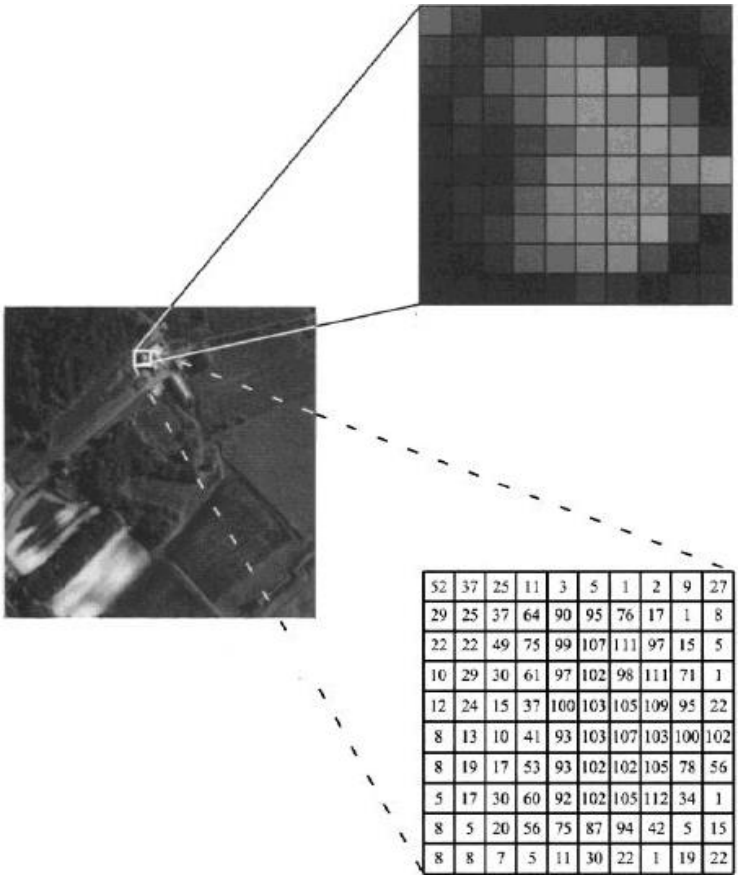


Imagen 3.4 – Matriz de una imagen [19]

3.1.3 Clasificación de las imágenes digitales

Dependiendo del rango de valores que tomen los pixeles de la imagen podemos distinguir los siguientes tipos de imágenes:

- Imagen binaria: se corresponde con una profundidad de color de 1 bit. La imagen está formada por pixeles blancos o negros puros.

Imagen 3.5 – Lena binaria

- Imagen de intensidad: también conocidas como imágenes en escala de grises, maneja el canal negro y permite hasta 256 tonos de gris entre el blanco y el negro puros.

Imagen 3.6 – Lena gris

- Imagen en color indexado: utiliza un canal de color indexado de 8 bits, pudiendo obtener con ello hasta un máximo de 256 colores.

Imagen 3.7 – Lena indexada

- Imagen en color: cada color se forma por una combinación de tres canales. Cada canal se corresponde con un color primario (rojo, verde y azul). Se asigna un valor de intensidad a cada color que oscila entre 0 y 255.

Imagen 3.8 – Lena color

3.1.4 Espacios de color

Un espacio de color es la forma por la que se puede especificar, crear y visualizar un color [20].

- Modelo RGB (Red, Green, Blue): está basado en la síntesis aditiva de las intensidades de luz relativas a los colores primarios (rojo, verde y azul) para conseguir los distintos colores, incluidos el negro y el blanco. La representación gráfica del modelo RGB se realiza mediante un cubo unitario con los ejes R, G y B.

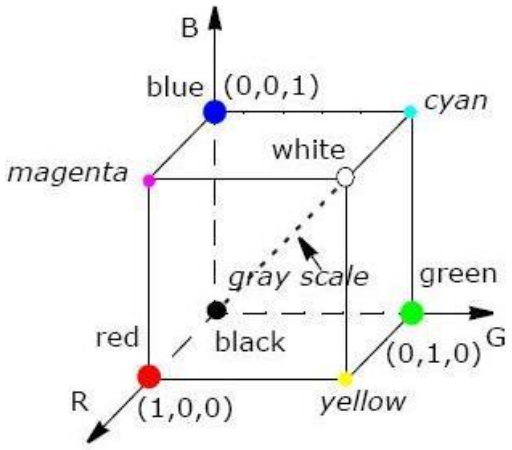


Imagen 3.9 – Modelo RGB

- Modelo HSV (Hue, Saturation, Value): representación tridimensional del color basada en los componentes matiz, saturación y valor o brillo (HSB). El sistema coordenado es cilíndrico y el subconjunto de este espacio donde se define el color es una pirámide de base hexagonal.

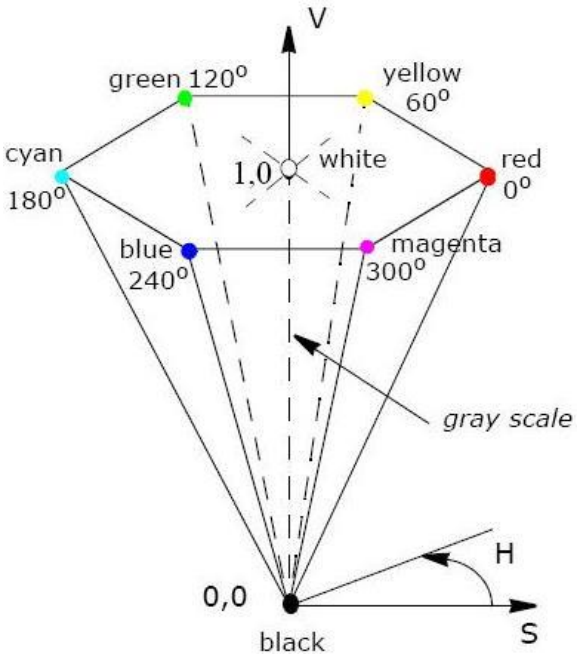


Imagen 3.10 – Modelo HSV

- Modelo YCbCr: es una codificación no lineal del espacio de color RGB, usada comúnmente en la compresión de imágenes. El color es representado por la luminancia o claridad del color (Y) y por dos valores de color (Cb ubica el color entre el azul y el amarillo y Cr ubica el color entre el rojo y el verde) que son características colorimétricas de este.

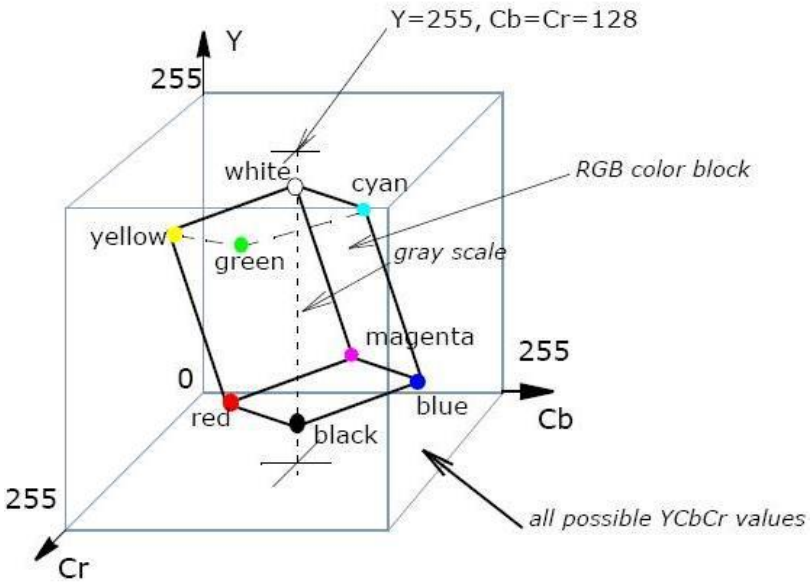


Imagen 3.11 – Modelo YCbCr

- Modelo CMYK (Cian, Magenta, Yellow, Key): es un modelo de color sustractivo que se utiliza en la impresión en colores. Está formado por los colores cian, magenta y amarillo, que son los colores complementarios al rojo, verde y azul, además del negro. El sistema coordenado es el mismo que el modelo RGB, pero con dichas coordenadas intercambiadas.

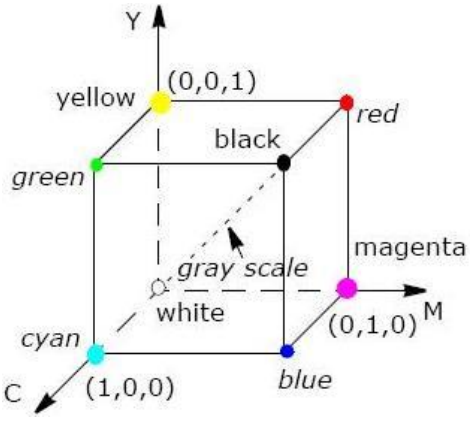


Imagen 3.12 – Modelo CMYK

- Modelo HLS (Hue, Luminosity, Saturation): corresponde a un modelo de color definido por el tono, la luminosidad y la saturación. El espacio se define sobre una doble pirámide hexagonal.

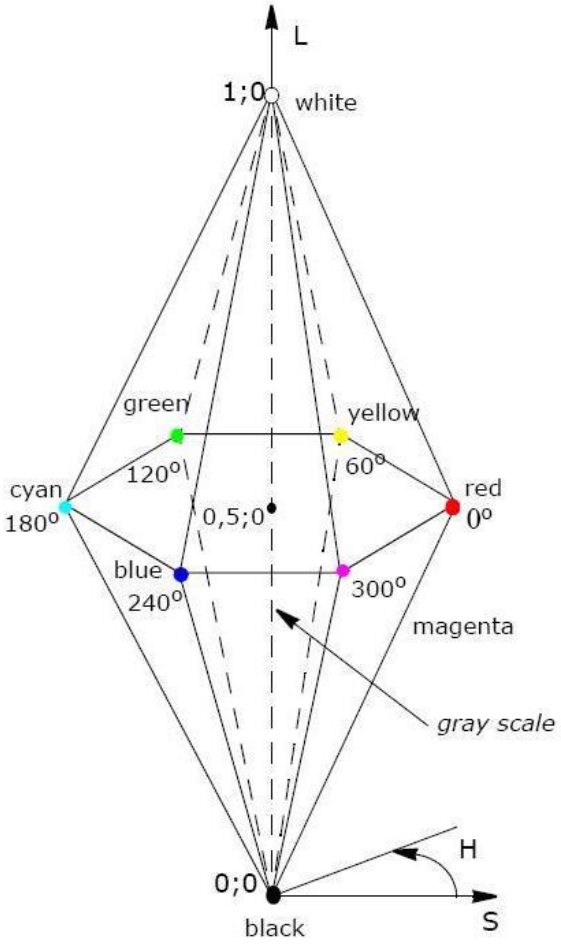


Imagen 3.13 – Modelo HLS

A continuación se describen brevemente las principales operaciones de preprocesado sobre imágenes digitales, para después pasar a describir en profundidad los algoritmos y técnicas de escalado de imágenes.

3.2 Operaciones básicas

Estas operaciones son las más usadas a cualquier nivel en un sistema de tratamiento de imágenes, ya que son las que se utilizan para leer y dar valores a los pixeles de las imágenes.

3.2.1 Operaciones aritméticas

- Adición y substracción:

  La suma de imágenes se utiliza principalmente para aplicaciones de efectos especiales, eliminación de ruido y modelado del mismo [21].

  La diferencia de imágenes tiene dos aplicaciones principales, la comparación entre objetos o escenas y la detección y análisis del movimiento dentro de una escena.

- Producto o división por una constante o por otra imagen:
  La multiplicación de imágenes se utiliza principalmente para aplicaciones de filtrado. La división de imágenes se usa para la detección de cambios fraccionales o ratio de una imagen (rationing)
- Logaritmo y exponencial:
  La función logaritmo se utiliza para aumentar el contraste de las zonas oscuras en detrimento de las zonas claras. La función exponencial se utiliza para aumentar el contraste de las zonas claras en detrimento de las oscuras.

3.2.2 Operaciones lógicas

- Complementación (obtención del negativo, NOT):
  Consiste en invertir el valor de los pixeles de la imagen. Se cambia el nivel de los grises de forma que los blancos se convierten en negro y los negros a blancos. El efecto que se obtiene es como el negativo de una foto.

Imagen 3.14 – Lena original y Lena complementada

- AND, OR y XOR:
  Son muy útiles en el uso de máscaras que sirven para obtener o eliminar una porción determinada de la imagen. La operación AND se usa para borrar pixeles de una imagen. La operación OR se usa para añadir pixeles a una imagen.

3.2.3 Umbralización

También conocida como thresholding, consiste en eliminar los valores superiores o inferiores (poniéndolos a cero) respecto a un valor conocido como umbral.

Imagen 3.15 – Lena original (arriba izquierda) y Lena umbralizada a distintos niveles (50 arriba derecha, 100 abajo izqiuerda, 150 abajo derecha)

3.2.4 Binarización

La binarización es una variante de la umbralización y consiste en dejar a cero todos los pixeles menores de un umbral y a uno aquellos que son iguales o mayores, quedando constituida la imagen final por un conjunto de unos y ceros.

Imagen 3.16 – Lena original (arriba izquierda) y Lena binarizada a distintos niveles (50 arriba derecha, 100 abajo izquierda, 150 abajo derecha)

3.3 Transformaciones geométricas de imágenes

Estos algoritmos modifican las características geométricas de las imágenes, su uso frecuentemente se aplica en la reconstrucción de imágenes deformadas, el giro y ajuste de las mismas o la deformación intencionada de ciertos rasgos para posteriores análisis. Generalmente los más utilizados son los de escalado, traslación, giro y espejo.

Todos los algoritmos correspondientes a transformaciones geométricas se basan en realizar una nueva distribución de los pixeles según lo que se pretenda. De esta forma el proceso de transformación geométrica se fundamenta en:

- Determinar las nuevas coordenadas de cada pixel (i, j) en la rejilla transformada (i', j'). Estas nuevas coordenadas generalmente, no serán valores enteros. Este proceso depende del tipo de transformación a realizar.
- Una vez obtenidos (i', j'), hay que calcular los valores de los pixeles (x, y) en la rejilla destino. Este proceso es común a todas las trasformaciones y se denomina interpolación.

El uso de estas transformaciones geométricas en un entorno en tiempo real es muy reducido, debido al tiempo de proceso que necesitan.

Se exponen a continuación las aplicaciones más comunes de cada tipo de algoritmo.

3.3.1 Algoritmos de traslación o desplazamiento

Consiste en sustituir cada pixel por el correspondiente a sus coordenadas más el desplazamiento en cada dirección k y l. Si los desplazamientos son enteros en ambas direcciones el proceso consiste en sustituir cada pixel (i, j) por el pixel (i + k, j + l), mientras que si tienen parte decimal es necesario realizar una interpolación.

En las aplicaciones prácticas el proceso de traslación o desplazamiento se utiliza cuando se quiere posicionar cierto objeto detectado en un punto determinado para realizarle posteriores procesos, como por ejemplo, el uso de funciones lógicas como máscaras para obtener o eliminar ciertas partes de este, procesos de unión o fusión con otras imágenes, etc.

3.3.2 Algoritmos de rotación o giro

Los algoritmos de giro son generalmente los más complejos y por lo tanto los más costosos en tiempo de procesado. Debido a esto, solo se utilizan cuando es posible obtener una posición de giro que simplifique más posteriores procesos. Tras el giro será necesario realizar una interpolación.

Por ejemplo, si se tiene una imagen formada por múltiples rectángulos y cuadrados situados de forma perpendicular entre ellos y la imagen esta girada, es muy útil alinearlos respecto a los ejes x e y de la imagen. De esta forma los procesos posteriores de segmentación y análisis sobre estos rectángulos o líneas serán más rápidos y eficaces.

3.3.3 Algoritmos de escalado y de zoom

Los algoritmos de escalado permiten reducir o aumentar la imagen, así como realizar el zoom de ciertas partes de la imagen. Dado un factor de escalado α para las coordenadas x y otro β para las coordenadas y, se requerirá una interpolación cuando i/α o j/β no sean enteros.

El uso de las funciones de escalado no se circunscribe únicamente a la etapa de preprocesado sino que pueden incluirse en pasos de otras etapas. Por ejemplo, existen procesos posteriores (erosionados, reducción de la resolución, etc.) que no necesitan tanta cantidad de información, reduciéndose la imagen para que estos procesos sean más rápidos ya que tratan con una imagen de tamaño reducido.

3.3.4 Espejos

Algoritmos que realizan un espejo respecto de la horizontal, vertical o diagonal.

3.4 Filtrado

Como se ha expuesto en apartados anteriores, existen diversos tipos de ruidos que se producen por múltiples circunstancias: los sensores, ruidos eléctricos, perturbaciones en el medio de transmisión, efectos térmicos, campos electromagnéticos, etc.

La mayoría de las implementaciones de filtros se realizan en dos dominios: el dominio espacial y el dominio frecuencial. Los métodos basados en el dominio espacial hacen referencia a la manipulación directa de la luminancia de los pixeles, mientras que los métodos basados en el dominio frecuencial modifican indirectamente la luminancia de cada pixel utilizando como factores de ponderación los valores de los otros pixeles de la imagen o del entorno del punto y las relaciones numéricas entre ellos.

3.4.1 Filtros frecuenciales

Las técnicas basadas en filtros de dominio frecuencial se basan en la Transformada Discreta de Fourier (DFT) y la Transformada Discreta de Fourier Inversa (IDFT).

Las operaciones frecuenciales se pueden entender como la modificación directa del espectro de frecuencias de la imagen, por supuesto, este espectro de frecuencias se define como frecuencias espaciales para una imagen. El proceso consiste en obtener el espectro de la imagen y del filtro a aplicar, y multiplicar ambos para realizar el filtrado.

3.4.2 Filtros espaciales

Este tipo de filtros suele consistir en recorrer toda la imagen, pixel a pixel, y realizar alguna operación aritmética con un mínimo concreto de pixeles vecinos. Al conjunto de estos vecinos se les denomina ventana, la cual deberá tener un tamaño cuadrado.

La forma de operar de estos filtros es por medio de la utilización de máscaras que recorren toda la imagen centrando las operaciones sobre los pixeles que se encuadran en la región (ventana) de la imagen original que coincide con la máscara h y el resultado se obtiene mediante una computación entre los pixeles originales y los diferentes coeficientes de las máscaras.

El filtrado espacial se realiza trasladando una matriz cuadrada de dos dimensiones que contiene pesos o ponderaciones sobre la imagen en cada localización de pixel. Se evalúa el pixel central de la ventana de acuerdo con los pixeles de alrededor y sus valores de ponderación. El proceso de evaluar la vecindad ponderada del pixel se denomina convolución bidimensional.

Se exponen a continuación algunos de los filtros empleados con mayor frecuencia.

- Filtro de media:
  Es un filtro paso de baja que se utiliza fundamentalmente para reducir el ruido de alta frecuencia que se produce en una imagen. Consiste en sustituir cada pixel por la media aritmética de los puntos que tiene alrededor (incluido el mismo). Se toma una ventana de NxN (tamaño de la máscara) puntos donde el punto a sustituir es el central, se obtiene la media de la suma de todos los valores de los pixeles de la ventana y se sustituye el pixel de la nueva imagen por el valor obtenido. El efecto del filtro aumenta a medida que lo hace el tamaño de la máscara.

$$h_1 = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \quad h_2 = \frac{1}{10}\begin{bmatrix}1 & 1 & 1\\ 1 & 2 & 1\\ 1 & 1 & 1\end{bmatrix} \quad h_3 = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} \qquad [3.1]$$

  Se puede observar que la suma de los coeficientes de la matriz es 1. Los resultados que produce son equivalentes a una difuminación de la imagen produciendo un suavizado de los gradientes en toda la imagen. Es de gran utilidad para eliminar ruido espúreo producido por un muestreo deficiente o una transmisión ruidosa.

- Filtro de mediana:
  Es un filtro paso de baja que consiste en obtener la lista de todos los valores de los pixeles de la ventana, ordenarlos y coger el del medio (mediana), es decir, el valor que tenga igual número de valores superiores e inferiores a él dentro de la ventana.
  Este filtro, al igual que el anterior, es muy usado para eliminar el ruido impulsivo (ruido sal y pimienta) de una imagen. La ventaja de este con el anterior, es que, el valor final del pixel es un valor real presente en la imagen y no uno promediado, de este modo se reduce el efecto borroso que tienen las imágenes que han sufrido un filtro de media.

Imagen 3.17 – Lena con ruido sal y pimienta y Lena tras aplicación de filtro de mediana

- Filtro de moda:
  La moda de un conjunto de valores se define como el valor que más se repite dentro de ellos. Por lo tanto, el filtro de moda (filtro paso de baja) consiste en calcular el valor más repetido dentro de todos los pixeles de una ventana. Si hay más de una moda, puede haber varios criterios como promediar las modas o seleccionar aquella más cercana al valor del pixel que se está tratando.
- Filtro gaussiano:
  Es un tipo de filtro paso de baja que simula una distribución gaussiana bivariante. El valor máximo aparece en el pixel central y disminuye hacia los extremos tanto más rápido cuanto menor sea el parámetro de desviación típica σ [22]. Un ejemplo de máscara puede ser:

$$h = \begin{bmatrix} 0.0113 & 0.0838 & 0.0113 \\ 0.0838 & 0.6193 & 0.0838 \\ 0.0113 & 0.0838 & 0.0113 \end{bmatrix} \quad [3.2]$$

- Filtros detectores de bordes mediante gradientes:
  Este tipo de filtros buscan y acentúan aquellos puntos donde el gradiente es mayor. El gradiente mide la rapidez en que los valores de los pixeles cambian en la distancia y en las direcciones x e y. Un operador muy conocido para el cálculo del gradiente es el operador de Sobel, cuyas matrices son:

$$\Delta x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \Delta y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad [3.3]$$

Combinando (media cuadrática) los resultados de la aplicación de ambas matrices se obtendría la imagen con detección de bordes.

Imagen 3.18 – Lena gradiente dx (arriba izquierda), Lena gradiente dy (arriba derecha), detección de bordes de Lena (abajo)

Otro operador es el de Prewitt:

$$\Delta x = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \Delta y = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad [3.4]$$

- Filtro laplaciano:

  La ventaja del laplaciano frente a los gradientes de primer orden, es que detecta mejor los bordes cuando las variaciones de intensidad de la imagen no son lo suficientemente abruptas. El problema es que estos suelen ser muy sensibles al ruido y reducen mucho el contraste de la imagen final.

$$h = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad [3.5]$$

- Filtro de contorno:
  Son utilizados para hacer que los bordes sean más marcados. Realmente se acentúan los puntos que más se diferencias de los vecinos, aumentando por lo tanto las altas frecuencias. En este caso el ruido de alta frecuencia aumenta (filtro paso de alta).

$$h_1 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad h_2 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 5 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad h_3 = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{bmatrix} \quad [3.6]$$

3.5 Operaciones basadas en histogramas

Se conoce como histograma de los niveles de cuantización (niveles de gris) de la imagen, o simplemente histograma de la imagen, a un diagrama de barras en el que cada barra tiene una altura proporcional al número de pixeles que hay para un nivel de gris determinado. Habitualmente, en el eje de abscisas se disponen los diferentes niveles de cuantización de los valores que pueden tomar los pixeles de tal imagen, mientras el eje de ordenadas refleja el número de pixeles que habrá para cada nivel de cuantización [23].

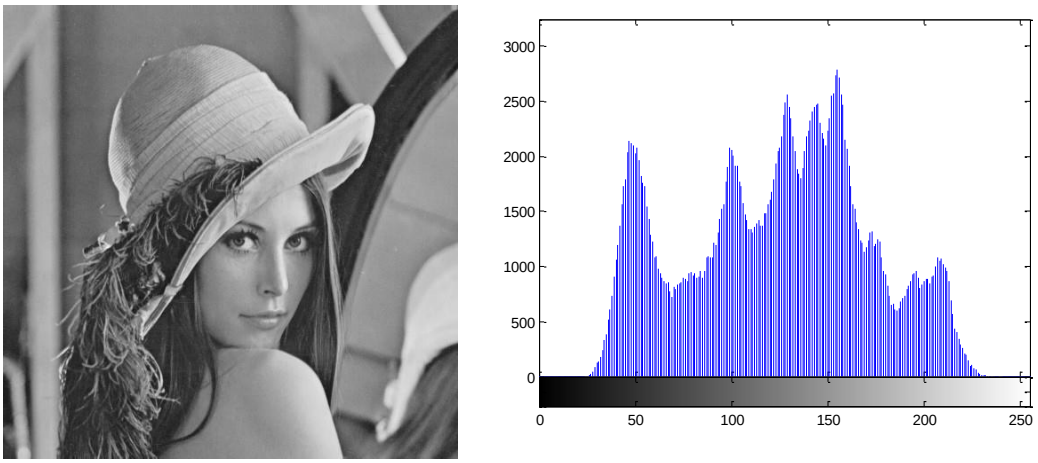


Imagen 3.19 – Lena e histograma correspondiente

Se debe notar que los histogramas no dicen nada sobre la disposición espacial de los pixeles. Por ello, un histograma es una forma de representación de una imagen en la que se produce perdida de información. De esto se deriva que, aunque una imagen solo tiene un histograma, imágenes diferentes podrían tener el mismo histograma.

3.5.1 Transformaciones del histograma

Una vez obtenido el histograma de una imagen, se puede multiplicar por una nueva función de transferencia que permite modificar el contraste de la imagen de una forma muy sencilla. El algoritmo se encarga de sustituir cada nivel de luminancia de los pixeles de la imagen por un nuevo valor obtenido de una tabla de transformación.

- Función cuadrado:
  El resultado de esta transformación es un oscurecimiento general, mejorando el contraste de los niveles bajos de gris pero empeorando el de los niveles altos.
- Función cubica:
  El resultado de esta función sigue en la línea del anterior, pero mucho más acentuado.

- Función raíz cuadrada:
  El resultado de esta función es el contrario a de la función cuadrado.
- Función raíz cubica:
  El resultado de esta función acentúa más los efectos en el blanqueo de la imagen comparándola con la función raíz cuadrada.

3.5.2 Ecualización del histograma

El proceso de ecualizado tiene por objetivo obtener un nuevo histograma, a partir del histograma original, con una distribución uniforme de los diferentes niveles de intensidad.

La ecualización del histograma trata de repartir los pixeles, de forma que la luminancia de estos esté más distribuida, consiguiendo, de esta forma, aumentar el contraste y distinguir mejor los distintos objetos que forma la imagen. Como defecto fundamental, la ecualización del histograma tiende a aumentar el ruido.

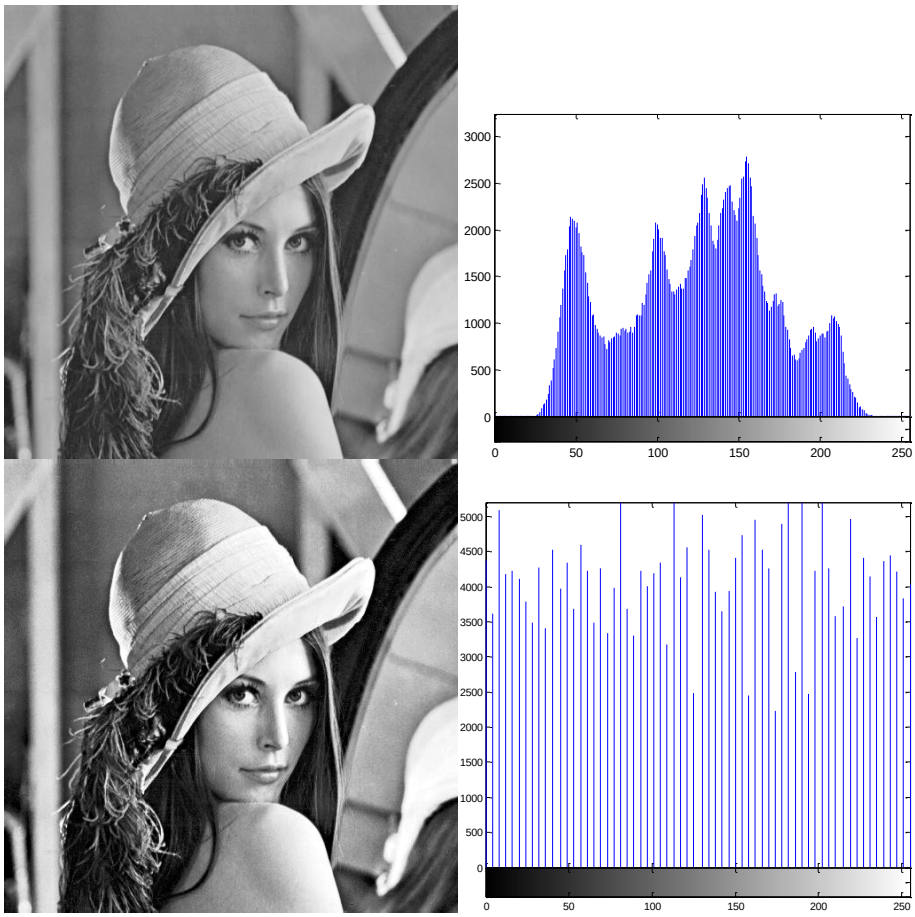


Imagen 3.20 – Lena e histograma original (arriba) y Lena e histograma mejorado (abajo)

3.6 Escalado de imágenes

Con la reciente integración de multimedia en casi todos los aspectos de la vida cotidiana, los consumidores están viendo imágenes, video y otros datos visuales en una amplia variedad de productos, que van desde televisores a las pantallas de ordenador pasando por una gran selección de dispositivos portátiles. La diversidad entre las pantallas de los usuarios finales va

más allá de lo que los proveedores de contenido pueden predecir, introduciendo así la necesidad de adaptar correctamente los datos de entrada al formato de salida apropiado [24].

Si bien es más eficiente transmitir datos visuales de baja resolución (a menudo combinado con la compresión), puede ser necesaria una aproximación de alta resolución para la presentación de los datos visuales. Por lo tanto, el escalado de imágenes es un paso esencial en muchas aplicaciones, que van desde diversos productos de consumo a funciones críticas dentro de la medicina, la seguridad o los sectores de defensa.

El escalado de imágenes es un proceso no trivial que implica una compensación entre eficiencia, suavidad y nitidez. Es obvio que, si se quiere acelerar el proceso de escalado, se reducirá la calidad de la imagen final. Es por ello, que dependiendo de la potencia de proceso del sistema que ejecute los algoritmos y de la calidad que se requiera, existen diversos algoritmos de interpolación para realizar el escalado.

Este proceso no se circunscribe únicamente a la etapa de preprocesado de imágenes. Puede ser requerida, en etapas posteriores, una reducción de las dimensiones de la imagen, debido a que sea necesario un procesamiento más rápido de los datos o de una zona concreta de la imagen. En la etapa de representación puede requerirse una modificación de las dimensiones, debido al tamaño y resolución de la pantalla en la que se muestre.

El escalado de imágenes requerirá de un algoritmo de interpolación en los casos en que la relación entre el ancho o alto de la imagen original y el de la imagen escalada no sea un número entero.

Existe un gran número de algoritmos de interpolación, debido a que muchos de ellos son pequeñas mejoras de algoritmos ya existentes. Entre ellos destacamos la interpolación por vecino más cercano (*nearest neighbor*), por ser la técnica más sencilla y rápida, y también la interpolación bilineal y bicúbica que proporcionan una mayor precisión a costa claro, de un mayor consumo de recursos y tiempo de procesado. También daremos algunas nociones sobre el algoritmo denominado interpolación en escalera (*stair interpolation*) y las imágenes vectoriales.

3.6.1 Nearest neighbor

El algoritmo de interpolación más sencillo y rápido empleado en las técnicas de escalado de imagen es el de vecino más cercano (una atribución más correcta seria “algoritmo de selección de pixeles”). Es muy útil cuando la velocidad de procesado es la principal limitación (muy empleado para vistas previas en miniatura, por ejemplo) [25], [26], [27].

El principio de este algoritmo es muy sencillo. Se trata de, una vez calculada la posición que ocupará el nuevo pixel, escoger de la imagen inicial el pixel que ocupa una posición proporcional a la de la imagen final. Podemos suponer su funcionamiento como un interpolador a trozos. Esto va en detrimento de la calidad de la imagen final, al no considerar los demás pixeles cercanos al escogido. Gráficamente lo podemos ver cómo sigue:

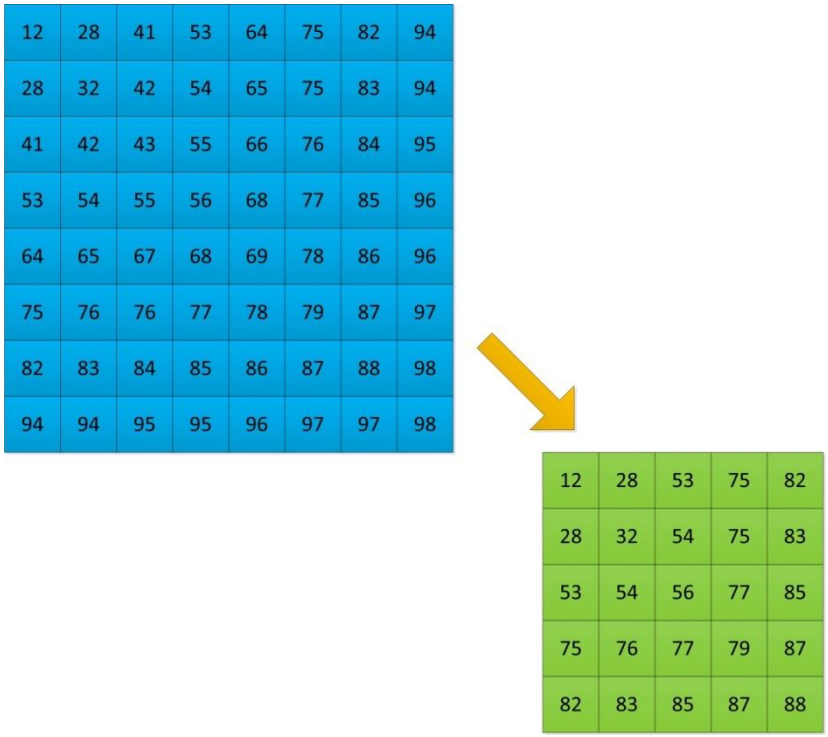


Imagen 3.21 – Matrices de imágenes tras reducción de escala con interpolación nearest neighbor

Aquí tenemos la representación de las matrices de dos imágenes (azul imagen inicial, verde imagen final). El comienzo del algoritmo consiste en la selección del primer pixel (0,0) de la imagen inicial como primer pixel (0,0) de la imagen final. Los siguientes se van escogiendo considerando la relación entre las dimensiones de la imagen inicial y final. Este proceso tiene el mismo funcionamiento que en la interpolación bilineal, como se verá en el Capítulo 4 (expresiones [4.14] a [4.19] y [4.24] a [4.26]). Así pues, se seleccionarán los pixeles de entre las columnas 0, 1, 3, 5 y 6 y de las filas 0, 1, 3, 5 y 6. Por ejemplo, para la posición (1,2) de la imagen final se usará el valor de la posición (1,3) de la imagen inicial, para la posición (3,3) de la imagen final, se usará el valor de la posición (5,5) de la imagen inicial, etc. Esto es para una reducción de escala. Para el caso contrario, ocurriría seria lo siguiente:

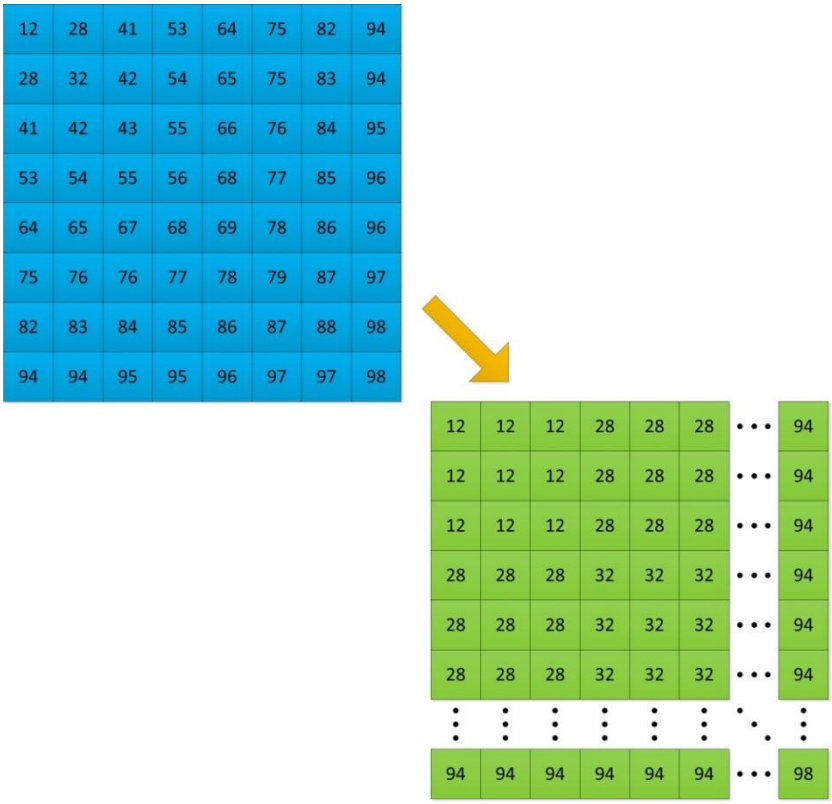


Imagen 3.22 – Matrices de imágenes tras ampliación de escala con interpolación nearest neighbor

Al igual que antes, el algoritmo comenzaría escogiendo el primer pixel de la imagen inicial, pero en este caso lo repetirá tantas veces como indiquen las relaciones entre el ancho y alto de las imágenes inicial y final. Esto es, dado que la relación ahora es 0.3333 (la imagen final, en verde, es tres veces mayor), por ejemplo, para las posiciones formadas por el cuadrado de vértices (0,0), (0,2), (2,0) y (2,2) de la imagen final se escogería el pixel en la posición (0,0.667) de la imagen inicial. Como no existen posiciones decimales, sería el pixel de la posición (0,0).

Si pasamos a analizar los resultados gráficos de este algoritmo (ver imagen 3.27), podemos apreciar que se obtienen imágenes dentadas (para ampliación), así como un claro aumento del tamaño de los pixeles originales de la imagen, y en consecuencia, la introducción de información redundante. Por el contrario, la reducción de escala implica la reducción del número de pixeles y por tanto, una pérdida de información irrecuperable.

Dos técnicas adicionales comúnmente empleadas son la interpolación bilineal y bicúbica. Estos procedimientos examinan los puntos en torno a los datos que faltan para aproximar matemáticamente los valores necesarios, como veremos a continuación.

3.6.2 Interpolación bilineal

El escalado de imágenes mediante el algoritmo de interpolación bilineal se basa en el funcionamiento del algoritmo de vecino más cercano, con la excepción de la interpolación [26], [27], [28], [29], [30], [31], [32].

En este caso, en lugar de copiar directamente los valores de la imagen inicial, la interpolación bilineal se basa en los valores circundantes a la posición del pixel seleccionado para obtener el

valor final. Concretamente, se usan los valores de los cuatro pixeles más cercanos. Así pues, nos podemos referir a esta técnica como vecindad 2x2 o vecinos más cercanos. A continuación se realiza un promedio de estos cuatro pixeles para obtener el dato buscado para la imagen final. Esto se traduce en imágenes de aspecto mucho más suave que las obtenidas por el método de vecino más cercano.

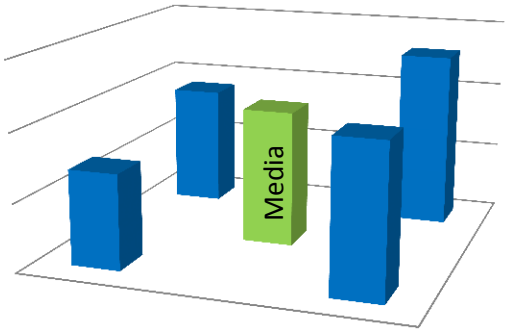


Imagen 3.23 – Interpolación bilineal aplicada a imágenes

Matemáticamente, la interpolación bilineal es una extensión de la interpolación lineal para funciones de dos variables. La idea principal consiste en realizar dos interpolaciones lineales en una dirección (horizontal o vertical) y una tercera en la otra dirección (vertical, horizontal). Si consideramos la siguiente imagen:

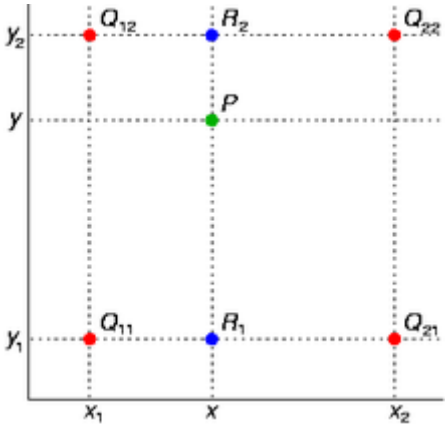


Imagen 3.24 – Interpolación bilineal matemática [33]

Aquí se pretende obtener el valor de P a partir de los cuatro puntos Q_{11}, Q_{12}, Q_{21}, Q_{22}. Para ello se realizarán primero dos interpolaciones lineales horizontales en las que se obtendrán los puntos R_1 y R_2 de la siguiente forma:

$$R_1 = \frac{x_2 - x}{x_2 - x_1} Q_{11} + \frac{x - x_1}{x_2 - x_1} Q_{21} \quad [3.7]$$

$$R_2 = \frac{x_2 - x}{x_2 - x_1} Q_{12} + \frac{x - x_1}{x_2 - x_1} Q_{22} \quad [3.8]$$

A continuación se realizará una tercera interpolación vertical con la que se obtendrá el punto buscado P de la siguiente forma:

$$P = \frac{y_2 - y}{y_2 - y_1} R_1 + \frac{y - y_1}{y_2 - y} R_2 \quad [3.9]$$

Debemos tener en cuenta, que aunque cada una de las interpolaciones que se realizan son lineales, el resultado final es una función cuadrática.

El algoritmo se encargará de calcular los valores de los coeficientes que aparecen en las anteriores expresiones (función de la distancia entre los pixeles y la relación de dimensiones entre la imagen inicial y final).

Para comparar mejor este método con el anterior, recurriremos a las mismas imágenes pero con los valores que se obtendrían mediante la interpolación bilineal (solo veremos el caso de reducción de escala, el por qué y los cálculos se justifican en el Capítulo 4).

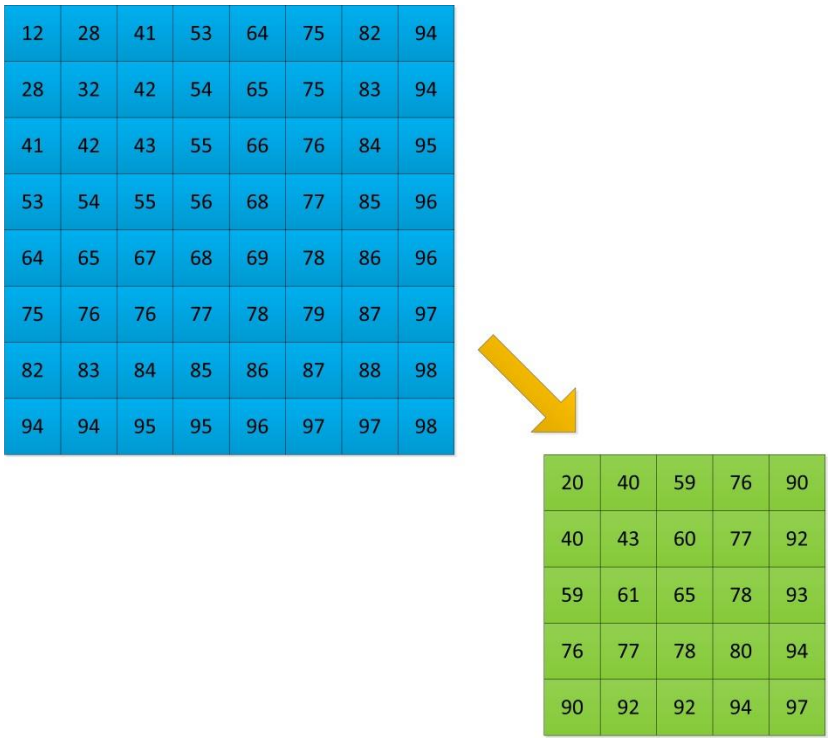


Imagen 3.25 – Matrices de imágenes tras reducción de escala con interpolación bilineal

Los resultados que proporciona la interpolación bilineal (ver imagen 3.28) son mucho mejores que el de la técnica de vecino más cercano. En concreto, para el caso de la ampliación, se mejora en gran medida el efecto de dientes de sierra provocado por la duplicación de los pixeles. Si en cambio hablamos de una reducción de escala, también se obtienen mejoras, aunque menos apreciables que en el caso anterior. Aun así, siguen existiendo defectos como aliasing, visión borrosa, halo de bordes, etc.

El siguiente algoritmo que veremos es el de interpolación bicúbica, que supone una mejora con respecto a la interpolación bilineal.

3.6.3 Interpolación bicúbica

La interpolación bicúbica va un paso más allá de la bilineal al considerar los 4x4 vecinos (pixeles) más cercanos. Dado que estos 16 pixeles no se encuentran a la misma distancia del pixel desconocido, para lograr mayor nitidez se les trata con diferente ponderación en el cálculo [28], [29], [32], [34].

En los casos en que las curvas producidas por la interpolación bilineal sean demasiado inexactas, la interpolación bicúbica es una excelente solución, ya que garantiza la continuidad de las primeras derivadas (gradientes) así como de la derivada cruzada.

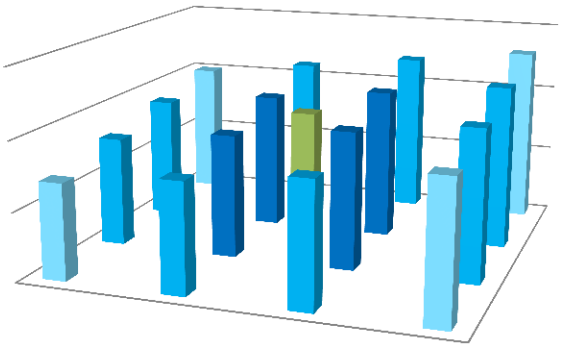


Imagen 3.26 – Interpolación bicúbica aplicada a imágenes

Matemáticamente, la interpolación bicúbica se puede lograr utilizando polinomios de Lagrange, splines cúbicos o algoritmos de convolución cúbica.

La interpolación bicúbica produce imágenes notablemente más nítidas que la bilineal, y es quizás la combinación ideal de tiempo de procesado y calidad obtenida. Por esta razón, es un estándar en muchos programas de edición de imágenes, controladores de impresoras, interpolación en cámaras, etc.

A continuación realizaremos una comparación grafica de los resultados de ampliar y reducir una imagen por cada uno de los tres algoritmos comentados anteriormente mediante Matlab.

Imagen 3.27 – Lena reducida (izquierda) y zoom (derecha) mediante interpolación nearest neighbor

Imagen 3.28 – Lena reducida (izquierda) y zoom (derecha) mediante interpolación bilineal

Imagen 3.29 – Lena reducida (izquierda) y zoom (derecha) mediante interpolación bicúbica

3.6.4 Stairstep interpolation

Consiste en realizar múltiples operaciones de interpolación con pequeños incrementos de escala, usando cualquiera de los tres algoritmos de interpolación que hemos visto anteriormente (preferiblemente bilineal o bicúbica). Dichos incrementos suelen ser del orden de un 10 o 20% de las dimensiones de la imagen inicial hasta llegar a las dimensiones deseadas. La teoría es que al realizar el escalado en pequeños pasos y no directamente a las dimensiones finales el rendimiento de los resultados es superior [35], [36].

Por ultimo veremos un tipo de imágenes que permite la ampliación sin que se produzca ningún tipo de distorsión en estas.

3.6.5 Imágenes vectoriales

Las imagines vectoriales están compuestas por entidades geométricas simples (segmentos, polígonos, arcos, etc). Cada una de estas entidades está definida matemáticamente por un grupo de parámetros (coordenadas inicial y final, grosor y color del contorno, color de relleno, etc.). Por compleja que pueda parecer una imagen, puede reducirse a una colección de entidades geométricas simples [37], [38].

Este formato de imagen es completamente distinto al formato de las imágenes de mapa de bits, que están formadas por pixeles. El interés principal de las imágenes vectoriales es poder modificar su escala sin sufrir la pérdida de calidad que sufren los mapas de bits.

A continuación mostramos una imagen de Lena y su correspondiente imagen vectorizada:

Imagen 3.30 – Lena original y Lena vectorizada

Si procedemos a realizas sucesivas ampliaciones de la imagen vectorizada, obtenemos los siguientes resultados:

Imagen 3.31 – Ampliaciones de escala sobre Lena vectorizada

4. Diseño en software

El desarrollo de hardware es a menudo un proceso lento y costoso. Lo primero está justificado por el tiempo que implica la simulación de códigos muy complejos (un segundo en una simulación puede tardar horas en obtenerse en tiempo real) y por la síntesis del código para obtener el diseño que se implementará en la FPGA donde se comprobará su funcionamiento. Ahora bien, el diseño puede funcionar correctamente en las pruebas de simulación, pero el comportamiento en hardware no se asegura con esto, por lo que también nos encontramos con la dificultad de la depuración, lo cual retrasa aún más la finalización del diseño. A esto hay que sumarle el otro aspecto negativo del desarrollo de hardware, el costo de las herramientas de desarrollo del código y de simulación y la plataforma (FPGA en nuestro caso) para la comprobación del funcionamiento del código.

Así pues, con el propósito de solventar en parte el tiempo de desarrollo del hardware, para el objetivo de este trabajo partimos de un código ya existente desarrollado en lenguaje C, en el que se ha implementado el algoritmo de interpolación bilineal. Este proviene de la librería OpenCV, puesta en marcha por Intel, que provee de funciones para el desarrollo de aplicaciones de visión artificial.

En nuestro caso, la función que usamos se denomina **resize**, a partir de la cual se ha creado una versión propia y optimizada, **imse_Resize**, que ha sido empleada en una aplicación de detección de rostros. Para ello nos hemos ayudado del software Eclipse CDT.

Eclipse es un entorno de desarrollo integrado (IDE). Escrito principalmente en Java, se puede utilizar para desarrollar aplicaciones. Por medio de plug-ins, también puede ser utilizado en otros lenguajes de programación: C, C++, JavaScript, Perl, PHP, Python, etc. Algunas de estas aplicaciones son Eclipse JDT para Java y Scala, Eclipse CDT para C/C++ (en nuestro caso con el paquete Kepler SR2 y el compilador MinGW) y Eclipse PDT para PHP, entre otros.

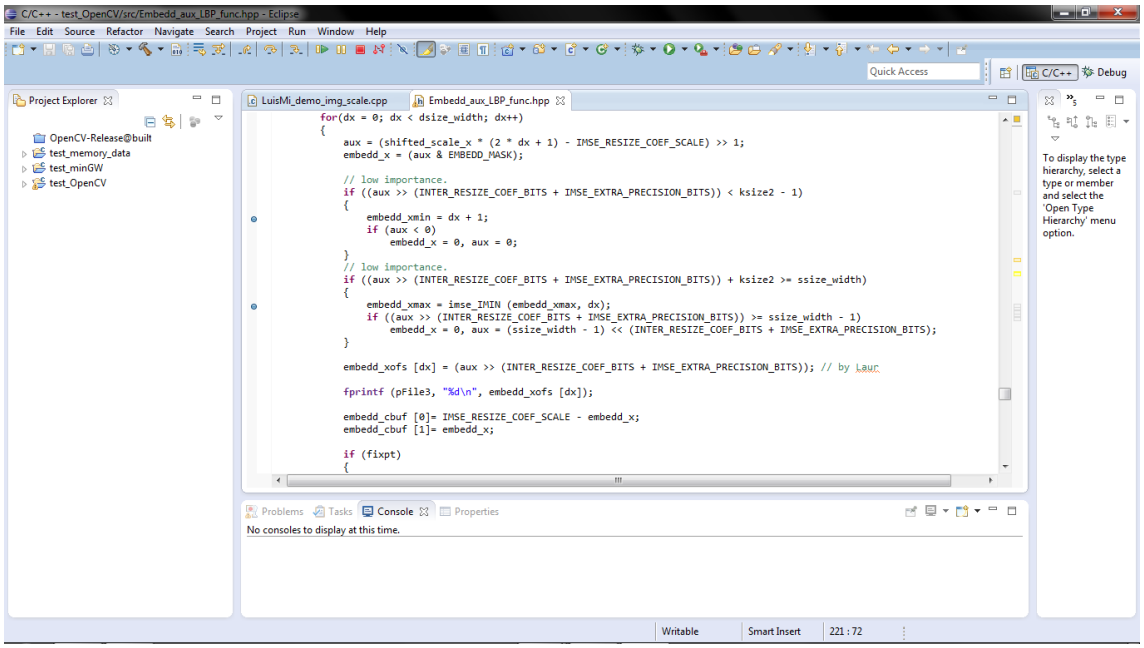


Imagen 4.1 – Entorno Eclipse CDT

Una vez introducido el entorno de programación en C, pasamos a describir el algoritmo en cuestión. Para ello partiremos de su diagrama de flujo.

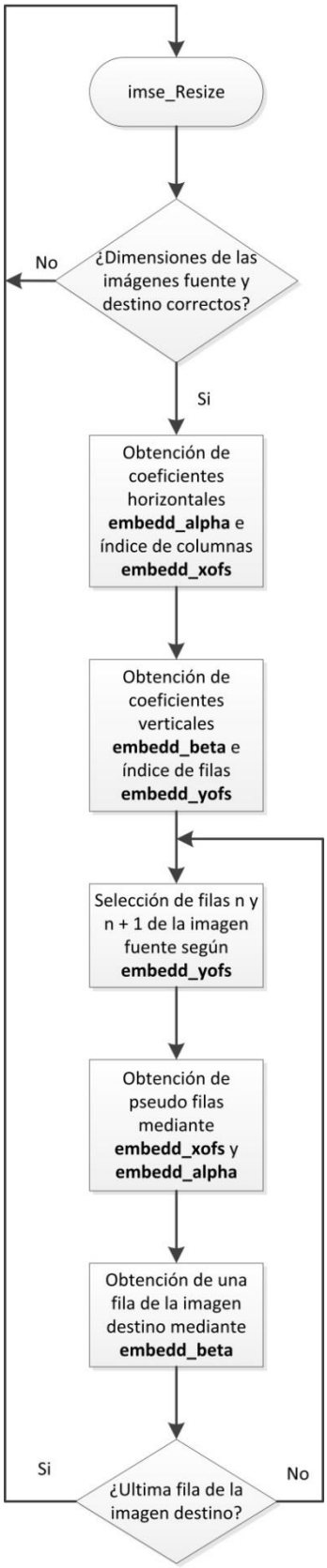


Imagen 4.2 – Diagrama de flujo del algoritmo en software

4.1 Función de interpolación imse_Resize

En la gran mayoría de ocasiones tratamos con imágenes en color. Esto no es más que tres capas en colores rojo, verde y azul que en conjunto producen la imagen a color. En nuestro caso, trabajaremos con imágenes en escala de grises, ya que no resulta necesaria la imagen en color para la aplicación a la que está destinado el escalado de imágenes, además de que resultaría un proceso tres veces más lento. Así pues, el algoritmo está preparado para trabajar con imágenes en gris. Junto al algoritmo de interpolación aquí descrito, se proporciona el código donde se realiza la conversión de imágenes RGB a gris (no entraremos en la descripción de este). Esto no evita que no pueda ser usado en otras aplicaciones para modificar las dimensiones de imágenes en color, tan solo sería necesario aplicar el mismo algoritmo a cada una de las capas de la imagen a color.

Dicho esto, pasamos a describir el funcionamiento del algoritmo de interpolación bilineal.

El prototipo de la función de interpolación es el siguiente:

```
imse_Resize (const void* srcarr, void* dstarr, int method)
```

Recibe como parámetros los punteros de las imágenes fuente y destino, junto con el método de interpolación. Los punteros indican la dirección de un tipo de estructura que contiene parámetros de las imágenes como las dimensiones y la dirección de inicio de estas. El tercer parámetro está relacionado con el método de interpolación y puede tomar uno de los siguientes valores:

```
#define imse_INTER_LINEAR       1
#define imse_INTER_CUBIC        2
#define imse_INTER_AREA         3
```

En nuestro caso será imse_INTER_LINEAR (realmente solo es funcional la que usamos al haber modificado el código original).

Una vez dentro de esta función, se realizará una conversión de las estructuras recibidas y llamaremos a **imse_resize_int_embb**, cuya definición es la siguiente:

```
static void imse_resize_int_embb (const imse_intMat src, imse_intMat dst,
                                  int dsize_width, int dsize_height,
                                  int interpolation)
```

Recibe como parámetros las nuevas estructuras junto con las dimensiones de la imagen destino (se obtienen de la estructura de la imagen destino), además del método de interpolación.

Esta función será la encargada de la obtención de los índices de filas y columnas y de los coeficientes de interpolación.

Antes de esto se realizará una comprobación de las dimensiones de las imágenes fuente y destino, con el objetivo de no continuar con el algoritmo de interpolación si alguna de estas es cero.

4.2 Obtención de los índices de filas y columnas y de los coeficientes de interpolación

En caso de contar con imágenes de dimensiones correctas, pasaremos a obtener los índices de selección de filas y columnas y los coeficientes de interpolación.

Antes de continuar con la descripción del código, expondremos el desarrollo matemático de la interpolación bilineal para dar a conocer algunas consideraciones que se tendrán en cuenta después.

En la imagen 3.24 se pretende obtener el valor de P a partir de los puntos Q_{11}, Q_{12}, Q_{21}, Q_{22}.

Para ello se realizan primero dos interpolaciones lineales en dirección horizontal con las que se obtendrán los puntos R_1 y R_2 de la siguiente forma:

$$R_1 = \frac{x_2 - x}{x_2 - x_1} Q_{11} + \frac{x - x_1}{x_2 - x_1} Q_{21} \quad [4.1]$$

$$R_2 = \frac{x_2 - x}{x_2 - x_1} Q_{12} + \frac{x - x_1}{x_2 - x_1} Q_{22} \quad [4.2]$$

A continuación se realiza una tercera interpolación lineal en dirección vertical con la que se obtendrá el punto buscado P de la siguiente forma:

$$P = \frac{y_2 - y}{y_2 - y_1} R_1 + \frac{y - y_1}{y_2 - y} R_2 \quad [4.3]$$

Dado que trabajamos con imágenes, los puntos con los que interpolamos son pixeles, así que nuestra correspondencia con la anterior imagen sería

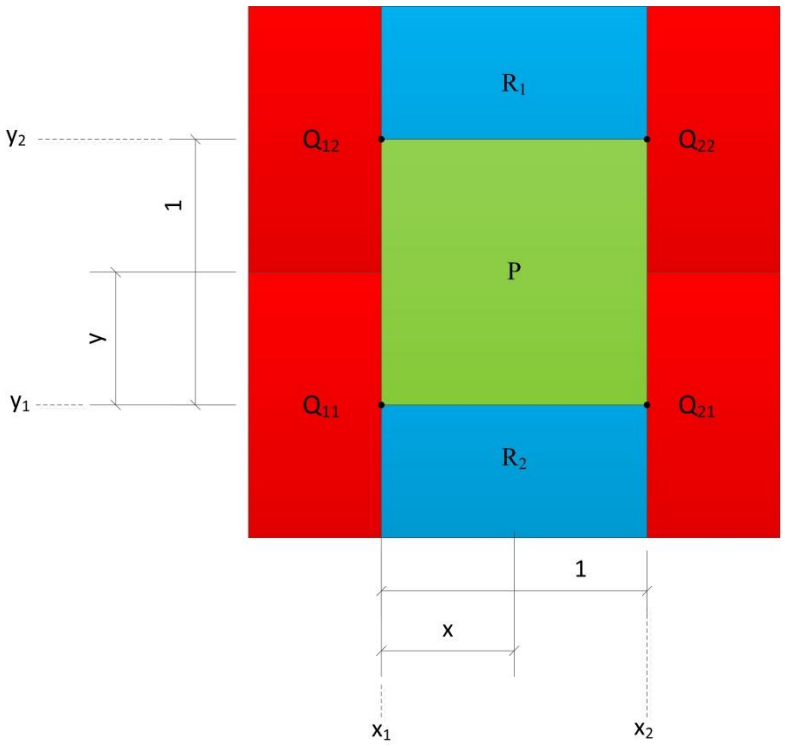


Imagen 4.3 – Explicación gráfica de la interpolación bilineal

En este caso el pixel a obtener está en el centro de los cuatro iniciales. En consecuencia debemos tomar algunas consideraciones en las anteriores expresiones matemáticas, que además facilitan el cálculo. Así pues, la distancia entre dos pixeles consecutivos será 1, medido desde el centro de estos, luego:

$$x_2 - x_1 = 1 \quad [4.4]$$

$$y_2 - y_1 = 1 \quad [4.5]$$

Como no se ha definido un origen, lo consideraremos en (x_1, y_1), con lo que:

$$x_1 = 0 \rightarrow x_2 = 1 \quad [4.6]$$

$$y_1 = 0 \rightarrow y_2 = 1 \quad [4.7]$$

quedando las expresiones iniciales reducidas a:

$$R_1 = (1 - x)Q_{11} + xQ_{21} \quad [4.8]$$

$$R_2 = (1 - x)Q_{12} + xQ_{22} \quad [4.9]$$

$$P = (1 - y)R_1 + yR_2 \quad [4.10]$$

Finalmente, nos queda determinar las distancias “x” e “y”. Dado que el pixel que estamos calculando se encuentra en el centro de los cuatros pixeles iniciales, estas distancias equivaldrían a la mitad de la distancia entre dos pixeles consecutivos, entonces:

$$R_1 = (1 - 0.5)Q_{11} + 0.5Q_{21} \quad [4.11]$$

$$R_2 = (1 - 0.5)Q_{12} + 0.5Q_{22} \quad [4.12]$$

$$P = (1 - 0.5)R_1 + 0.5R_2 \quad [4.13]$$

Una vez conocemos el fundamento matemático de la interpolación bilineal, podemos pasar a describir su funcionamiento en el algoritmo implementado.

El núcleo de la función **imse_resize_int_embb** son dos bucles *for* limitados por las dimensiones de la imagen destino. El primero de ellos obtiene el índice de columnas y los coeficientes de interpolación que se aplican en dirección horizontal. El límite de este bucle es el ancho o número de columnas de la imagen destino. El segundo obtiene el índice de filas y los coeficientes de interpolación que se aplican en dirección vertical. El límite de este bucle es el alto o número de filas de la imagen destino.

Para evitar trabajar con números decimales, y en consecuencia, tener la menor pérdida de precisión, se usa el factor de escala **IMSE_RESIZE_COEF_SCALE**, el cual puede modificarse, siendo su valor 2^{15} en nuestro caso. El valor del exponente de esta potencia de dos se obtiene de la suma de las constantes presentes en el código denominadas **INTER_RESIZE_COEF_BITS** (de valor 11) e **IMSE_EXTRA_PRECISION_BITS** (de valor 4).

El cometido del algoritmo no se trata simplemente de la realización del cálculo de la interpolación, si no que se extiende su función para el escalado de una imagen. Así pues, lo

primero que obtenemos es la relación entre las dimensiones de las imágenes fuente y destino. Dichas relaciones tienen las siguientes expresiones:

$$shifted\_scale\_x = \frac{src\_width}{dst\_width} IMSE\_RESIZE\_COEF\_SCALE \quad [4.14]$$

$$shifted\_scale\_y = \frac{src\_height}{dst\_height} IMSE\_RESIZE\_COEF\_SCALE \quad [4.15]$$

Así pues, **shifted_scale_x** guarda la relación entre el número de columnas de la imagen fuente y destino y **shifted_scale_y** la relación entre el número de filas de la imagen fuente y destino.

Tras esto, da comienzo el primero de los bucles, el cual describimos a continuación.

La distancia "x" a la que se hace referencia en las expresiones [4.8] y [4.9] no siempre valdrá 0.5, si no que variará en función de la relación de dimensiones **shifted_scale_x**. Dicha distancia se camufla en la obtención del índice de columnas y de los coeficientes de interpolación horizontales a partir de las siguientes expresiones ([4.16] a [4.23]):

$$aux = \frac{shifted\_scale\_x(2 \cdot dx + 1) - IMSE\_RESIZE\_COEF\_SCALE}{2} \quad [4.16]$$

$$embedd\_x = aux \,\&\, EMBEDD\_MASK \quad [4.17]$$

con

$$EMBEDD\_MASK = IMSE\_RESIZE\_COEF\_SCALE - 1 \quad [4.18]$$

La expresión [4.16] se corresponde exactamente con la presente en el código. La expresión [4.17] es un enmascaramiento del valor **aux** con **EMBEDD_MASK**, obteniéndose en **embedd_x** un valor en el rango 0 a **IMSE_RESIZE_COEF_SCALE** - 1.

Llegados a este punto del código, nos encontramos con dos sentencias *if*, las cuales solo afectan cuando se realiza una ampliación de escala. Esto, que deriva en complicaciones del código (**embedd_xmin** y **embedd_xmax**) y la observación de casos especiales (**aux** y **embedd_x** toman valores forzados), junto con la justificación de que no se necesita ampliación de escala para la aplicación a la que está destinada este trabajo (sustitución a la función en software del escalado de imágenes en una aplicación de detección de rostros), nos lleva a decidir no implementarlo en nuestro diseño en hardware, dejándolo como trabajo futuro para versiones posteriores. Así pues, no entramos en su descripción, pero si resaltamos sus valores para la reducción de escala, cero para **embedd_xmin** y **dsize_width** para **embedd_xmax**.

Si dividimos la expresión [4.17] por el factor **IMSE_RESIZE_COEF_SCALE,** nos queda lo siguiente:

$$embedd\_xofs = \frac{embedd\_x}{IMSE\_RESIZE\_COEF\_SCALE} \quad [4.19]$$

La parte entera de este resultado nos da el índice de columnas **embedd_xofs**. Este nos determina que pixeles de las filas de la imagen fuente intervien en los cálculos de interpolación.

Si desarrollamos esta expresión nos queda:

$$embedd\_xofs = \frac{\frac{src\_width}{dst\_width}(2 \cdot dx + 1) - 1}{2} = \frac{src\_width}{dst\_width}(dx + 0.5) - 0.5 \quad [4.20]$$

El parámetro **dx** es quien controla la evolución del bucle, limitado al ancho o número de columnas de la imagen destino, **dsize_width**.

Para comprender mejor esto, veamos un ejemplo de los valores que tomaría **embedd_xofs** para un escalado de una imagen de [n, 15] a una de [m, 9].

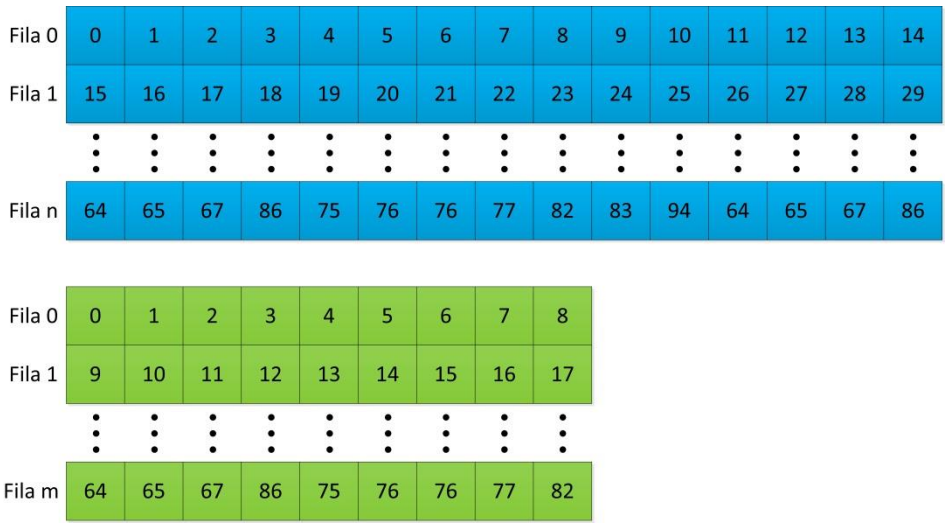


Imagen 4.4 – Escalado de [n, 15] a [m, 9]

$$embedd\_xofs\ [0] = \frac{15}{9}(0 + 0.5) - 0.5 = 0.333 \rightarrow embedd\_xofs\ [0] = 0$$

$$embedd\_xofs\ [3] = \frac{15}{9}(3 + 0.5) - 0.5 = 5.333 \rightarrow embedd\_xofs\ [3] = 5$$

$$embedd\_xofs\ [6] = \frac{15}{9}(6 + 0.5) - 0.5 = 10.333 \rightarrow embedd\_xofs\ [6] = 10$$

$$embedd\_xofs = \{0,2,3,5,7,8,10,12,13\}$$

Nota: el redondeo se debe a que ese sería el resultado obtenido con el cálculo en software.

Por ejemplo, para obtener el pixel 0 de la imagen destino, se emplea la vecindad 2x2 que comienza en el pixel 0 de la imagen fuente, es decir, los pixeles 0, 1, 15 y 16; para el pixel 5 de la imagen destino, los pixeles 8, 9, 23 y 24 de la imagen fuente, etc.

Finalmente nos queda determinar los coeficientes, los cuales se obtienen de forma muy sencilla. Dichos coeficientes serán los mismos para cada fila que procesemos.

En cada iteración del bucle se obtiene un valor para **embedd_xofs**. A cada valor de este le corresponden dos valores de coeficientes, puesto que para cada una de las interpolaciones horizontales se emplean dos pixeles, el indicado por **embedd_xofs** y el consecutivo. Así pues, en cada iteración del bucle se obtendrán dos coeficientes de interpolación, que se corresponden con **embedd_cbuf** como podemos ver a continuación:

$$embedd\_cbuf\ [0] = IMSE\_RESIZE\_COEF\_SCALE - embedd\_x \quad [4.21]$$

$$embedd\_cbuf\ [1] = embedd\_x \quad [4.22]$$

Dado que estos valores son temporales, a continuación se almacenan en **embedd_alpha**, de forma que al finalizar el bucle, este contiene todos los coeficientes que se aplicarán a los pixeles seleccionados por **embedd_xofs**. Dichos valores no se guardan tal cual se obtienen de **embedd_cbuf**, si no que se reduce el factor de escala que se empleó anteriormente. Dicha reducción se corresponde con **IMSE_EXTRA_PRECISION_BITS**, que es parte del exponente de la potencia de dos con la que se obtiene el factor de escala (**INTER_RESIZE_COEF_BITS** + **IMSE_EXTRA_PRECISION_BITS**) y equivale a dividir **embedd_cbuf** como sigue:

$$embedd\_alpha = \frac{embedd\_cbuf}{2^{IMSE\_EXTRA\_PRECISION\_BITS}} \quad [4.23]$$

Para ello se emplea la función **imse_saturate_short_from_int**.

Al igual que hicimos con **embedd_xofs**, procederemos a obtener algunos coeficientes **embedd_alpha** para el ejemplo anterior (imagen 4.4):

$$dx = 0 \rightarrow embedd\_x = \frac{\frac{15}{9}32768(2 \cdot 0 + 1) - 32768}{2} = 10922$$

$$embedd\_cbuf\ [0] = 32768 - 10922 = 21846\,,\ \ embedd\_cbuf\ [1] = 10922$$

$$embedd\_alpha\ [0] = \frac{21846}{16} = 1365,\ \ embedd\_alpha\ [1] = \frac{10922}{16} = 682$$

$$embedd\_alpha = \{1365{,}682{,}0{,}2047{,}341{,}1706, \dots, 341{,}1076\}$$

El segundo bucle tiene la misma funcionalidad que el descrito anteriormente, con la salvedad de que este está limitado por el número de filas de la imagen destino. A continuación se describe brevemente.

Al igual que ocurría con la distancia “x” en las expresiones [4.8] y [4.9], la distancia “y” de la expresión [4.10] también interviene en la obtención del índice de filas y de los coeficientes de interpolación verticales por medio de las siguientes expresiones ([4.24] a [4.29]):

$$auy = \frac{shifted\_scale\_y(2 \cdot dy + 1) - IMSE\_RESIZE\_COEF\_SCALE}{2} \quad [4.24]$$

$$embedd\_y = auy\ \&\ EMBEDD\_MASK \quad [4.25]$$

El termino **dy** es el que controla el bucle, limitado por el alto o número de filas de la imagen destino, **dsize_height.**

Dividiendo la anterior expresión por **IMSE_RESIZE_COEF_SCALE** obtendríamos:

$$embedd\_yofs = \frac{embedd\_y}{IMSE\_RESIZE\_COEF\_SCALE} \quad [4.26]$$

La parte entera del resultado es el índice de filas **embedd_yofs,** el cual escoge las filas de la imagen fuente de las cuales se seleccionarán los pixeles mediante **embedd_xofs** a los que se aplicarán los coeficientes **embedd_alpha.** A las pseudo filas (pixeles azules en la imagen 4.3) así obtenidas se les aplicará otros coeficientes de interpolación para obtener las filas de la imagen destino. Dichos coeficientes se obtienen como sigue:

$$embedd\_cbuf\ [0] = IMSE\_RESIZE\_COEF\_SCALE - embedd\_y \quad [4.27]$$

$$embedd\_cbuf\ [1] = embedd\_y \quad [4.28]$$

Estos se almacenan en **embedd_beta**, de forma que al finalizar el bucle, contiene todos los coeficientes que se aplicarán a cada una de las pseudo filas, obtenidas como vemos en el siguiente apartado. Aquí también realizamos la misma reducción en el factor de escala de dichos valores como se puede apreciar en la siguiente expresión:

$$embedd\_beta = \frac{embedd\_cbuf}{2^{IMSE\_EXTRA\_PRECISION\_BITS}} \quad [4.29]$$

Al finalizar la ejecución del segundo bucle, realizamos la llamada a la función **imse_resizeGeneric**, cuyo prototipo es el siguiente:

```
static void imse_resizeGeneric (const imse_intMat src, imse_intMat dst,
                                 const int* xofs, const void* _alpha,
                                 const int* yofs, const void* _beta, int xmin,
                                 int xmax, int ksize, const int ONE,
                                 const int SHIFT)
```

Recibe como parámetros las estructuras de las imágenes fuente y destino, el índice de selección de columnas junto con los coeficientes de interpolación horizontales, el índice de selección de filas junto con los coeficientes de interpolación verticales, otros dos parámetros (**xmin** y **xmax**) cuyo funcionamiento se obviará al no haber implementado la ampliación de escala en hardware, otro parámetro más relacionado con el tipo de interpolación, **ksize**, de valor dos (debido a que los coeficientes están almacenados de dos en dos) y dos constantes, **ONE** (de valor **IMSE_RESIZE_COEF_SCALE**) y **SHIFT** (de valor **INTER_RESIZE_COEF_BITS** * 2).

4.3 Obtención de las pseudo filas y filas de la imagen destino

Una vez dentro de **imse_resizeGeneric**, un bucle *for* controlado por **dy** recorre el vector **yofs** con el objetivo de seleccionar las filas de la imagen fuente de las que se escogerán los pixeles que intervendrán en los cálculos de interpolación. Así pues, en cada iteración del bucle se calcula la dirección de la fila (**srows**) que indique **yofs**. La función que se encarga de obtener las pseudo filas es **imse_HResizeLinear**. Esta está preparada para obtener con cada uso dos pseudo filas a partir de dos filas (“n” y “n + 1”) de la imagen fuente, pero esto no obliga a

conocer la dirección de ambas, puesto que la segunda es la consecutiva a la ya calculada. Dicha función tiene el siguiente prototipo:

```
static void imse_HResizeLinear (const uint8_t** src, int** dst, int count,
                                const int* xofs, const short* alpha,
                                int swidth, int dwidth, int xmin, int xmax,
                                const int ONE)
```

Recibe como parámetros la dirección de la fila seleccionada anteriormente y la dirección donde se guardarán temporalmente las pseudo filas aquí obtenidas, el parámetro **count** (relacionado con la selección de las filas), el índice de columnas junto con los coeficientes de interpolación horizontales, el ancho o número de columnas de las imágenes fuente y destino, los dos parámetros **xmin** y **xmax** y la constante **ONE**.

Esta función también está controlada por un bucle *for* limitado por el ancho o número de columnas de la imagen destino. Antes de iniciar el bucle, se tienen cuatro punteros, **S0** y **S1** que indican las direcciones de las filas "n" y "n + 1" de la imagen fuente y **D0** y **D1**, que indican la dirección donde se guardarán temporalmente las pseudo filas aquí obtenidas (estas quedarán consecutivas). Una vez dentro del bucle, se pasa a seleccionar los pixeles que indique el índice de columnas **xofs** y los coeficientes de interpolación **alpha** correspondientes, para obtener con ellos los valores **t0** y **t1**, correspondientes a los datos de las pseudo filas, según la expresión:

$$t = dato_n \cdot alpha_n + dato_{n+1} \cdot alpha_{n+1} \quad [4.30]$$

Una vez obtenidas dichas pseudo filas, estas serán empleadas por la función **imse_VResizeLinear** para obtener una fila de la imagen destino. El prototipo de esta función es el siguiente:

```
static void imse_VResizeLinear (const int** src, uint8_t* dst,
                                const short* beta, int width, const int SHIFT)
```

Recibe como parámetros la dirección de la primera de las pseudo filas (la segunda es consecutiva) obtenidas anteriormente y la dirección donde se ubicará la fila de la imagen destino, los coeficientes de interpolación verticales, el ancho o número de columnas de la imagen destino y la constante **SHIFT**.

Esta función, al igual que **imse_HResizeLinear**, está controlada por un bucle *for* limitado por el ancho o número de columnas (**width**) de la imagen destino. Así pues, se necesitan también dos punteros que indiquen la dirección de cada una de las pseudo filas (**S0** y **S1**). En cada iteración del bucle se obtienen cuatro valores (dos de **t0** y otros dos de **t1**) que se corresponden con cuatro pixeles consecutivos de una fila de la imagen destino (así se consiguen menos iteraciones en el bucle), según la siguiente expresión:

$$t = dato_n \cdot beta_n + dato_{n+1} \cdot beta_{n+1} \quad [4.31]$$

Como se comentó en el apartado 4.2, todos los cálculos que se han ido realizando están afectados por el factor de escala **IMSE_RESIZE_COEF_SCALE**, de modo que los valores que se

obtienen con **t0** y **t1** no se encuentran en el rango de valores de un pixel, por lo que se necesita retirar dicho factor de estos valores, mediante la función **imse_saturate_uchar** y la constante **SHIFT**. Aquí se ha de aclarar el valor de la constante **SHIFT**. Recordemos que en las expresiones [4.23] y [4.29] (**embedd_alpha** y **embedd_beta**) ya se redujo dicho factor de escala con **IMSE_EXTRA_PRECISION_BITS**, luego ya solo queda hacer lo mismo con **INTER_RESIZE_COEF_BITS**. Los valores de **t0** y **t1** se han obtenido a partir del producto de los coeficientes **embedd_alpha** y **embedd_beta**, por lo que dichos valores están afectados con $2^{2 \cdot \text{INTER\_RESIZE\_COEF\_BITS}}$, cuyo exponente es el valor que toma **SHIFT**. Tras esto se obtienen los valores de los pixeles de la imagen destino en su rango correcto.

El algoritmo concluirá con la finalización del bucle presente en **imse_resizeGeneric**. Tras esto se realizarán las pertinentes liberaciones de memoria (**imse_Free**) que se necesitó durante la ejecución del algoritmo para cuando vuelva a requerirse.

A continuación vemos un ejemplo gráfico de un escalado desde una imagen de 8x8 a una de 5x5 con los valores que se obtiene de la ejecución del algoritmo descrito. Nos detenemos en el proceso de obtención del pixel (0,0) de la imagen destino y los valores de los índices **embedd_xofs** y **embedd_yofs** y coeficientes **embedd_alpha** y **embedd_beta** (los valores están redondeados como ocurriría con el cálculo en software).

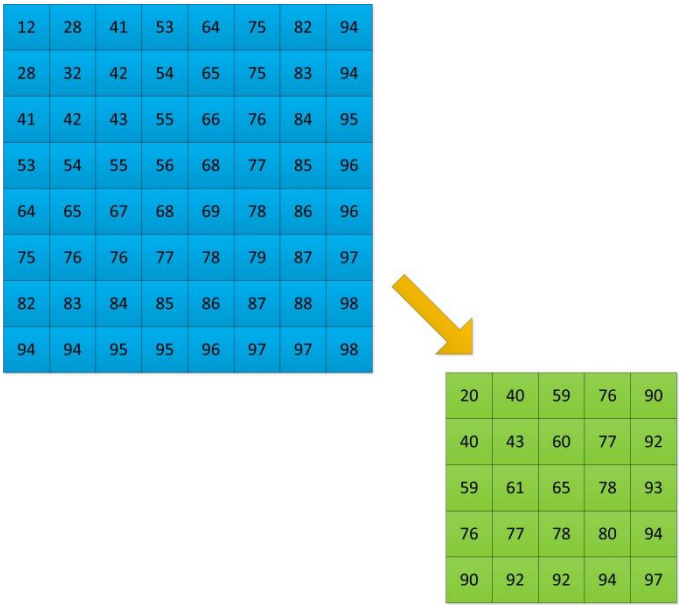


Imagen 4.5 – Escalado de 8x8 a 5x5

$$shifted\_scale\_x = \frac{8}{5}32768 = 52428 \quad [4.32]$$

$$dx = 0 \rightarrow aux = \frac{52428(2 \cdot 0 + 1) - 32768}{2} = 9830 \quad [4.33]$$

$$embedd\_x = 9830 \rightarrow embedd\_xofs\,[dx] = \frac{9830}{32768} = 0 \quad [4.34]$$

$$embedd\_cbuf[0] = 32768 - 9830 = 22938 \quad [4.35]$$

$$embedd\_cbuf\ [1] = 9830 \quad [4.36]$$

$$embedd\_alpha\ [0] = \frac{22938}{16} = 1433 \quad [4.37]$$

$$embedd\_alpha\ [1] = \frac{9830}{16} = 614 \quad [4.38]$$

$$h_0 = 12 \cdot 1433 + 28 \cdot 614 = 34388 \quad [4.39]$$

$$h_1 = 28 \cdot 1433 + 32 \cdot 614 = 59772 \quad [4.40]$$

$$v = 34388 \cdot 1433 + 59772 \cdot 614 = 85978012 \quad [4.41]$$

$$pixel = \frac{85978012 + 2^{22-1}}{2^{22}} = 20 \quad [4.42]$$

$$embedd\_xofs = \{0,1,3,5,6\} \quad [4.43]$$

$$embedd\_alpha = \{1433,614,204,1843,1024,1023,1843,204,614,1433\} \quad [4.44]$$

$$embedd\_yofs = \{0,1,3,5,6\} \quad [4.45]$$

$$embedd\_beta = \{1433,614,204,1843,1024,1023,1843,204,614,1433\} \quad [4.46]$$

5. Diseño en hardware

El propósito de este trabajo es la aceleración del algoritmo de interpolación bilineal anteriormente presentado, empleado para el escalado de imágenes en una aplicación de detección de rostros.

Esto se consigue con la descripción e implementación de dicho algoritmo mediante lenguajes de descripción de circuitos digitales, dando lugar al desarrollo de un módulo IP. En nuestro caso, de entre las numerosas posibilidades disponibles (VHDL, Verilog, ABEL, AHDL, etc), hemos hecho uso del lenguaje de descripción de circuitos digitales VHDL, del cual se dan algunas nociones a continuación.

5.1 VHDL

VHDL es el acrónimo que representa la combinación de VHSIC (*Very High Speed Integrated Circuit*) y HDL (*Hardware Description Language*). Es un estándar denominado ANSI/IEEE 1076-1993 definido por el IEEE (*Institute of Electrical and Electronics Engineers*) [39],[40].

Se trata de un lenguaje de descripción de hardware, es decir, mediante él se puede describir la forma de comportarse de un circuito electrónico. Dicho diseño puede ser llevado a algún dispositivo que dispondrá de sus propios componentes con los que lograr ese comportamiento deseado.

5.1.1 Historia

A mediados de los años setenta se produce una fuerte evolución en los procesos de fabricación de los circuitos integrados. En aquella época, el esfuerzo de diseño se concentraba en los niveles eléctricos para establecer características e interconexiones entre los componentes básicos a nivel de transistor. A medida que pasaban los años, los procesos tecnológicos se hacían más y más complejos. Los problemas de integración iban en aumento y los diseños eran cada vez más difíciles de depurar y de dar mantenimiento. Es entonces cuando diversos grupos de investigadores empiezan a crear y desarrollar los llamados lenguajes de descripción de hardware cada uno con sus peculiaridades. Sin embargo, estos lenguajes nunca alcanzaron el nivel de difusión y consolidación necesarias por motivos distintos.

Alrededor de 1981 el Departamento de Defensa de los Estados Unidos desarrolla un proyecto llamado VHSIC cuyo objetivo era rentabilizar las inversiones en hardware haciendo más sencillo su mantenimiento. Se pretendía con ello resolver el problema de modificar el hardware diseñado en un proyecto para utilizarlo en otro. Era el momento de los HDL´s. en 1983 IBM, Intermetrics y Texas Instruments empezaron a trabajar en el desarrollo de un lenguaje que permitiera la estandarización, facilitando con ello, el mantenimiento de los diseños y la depuración de algoritmos, para ello el IEEE propuso su estándar en 1984. Tras varias versiones, el IEEE publicó en diciembre de 1987 el estándar IEEE 1076-1987. A continuación, se seguiría actualizando, con versiones destacables como IEEE 1164 y la actual denominación, IEEE 1076-1993.

5.1.2 Introducción

Un proyecto de VHDL puede contener muchos ficheros. El código VHDL usualmente se encuentra en ficheros con extensión *.vhd. La sintaxis típica de estos ficheros es:

```
-- Llamadas a librerías

library ieee;

use ieee.std_logic_1164.all;

-- Entidad

entity entity_name is

	generic (generic_name : type := initialization;

		 generic_name : type := initialization);

	port (signal_name : mode type;

	      signal_name : mode type);

end entity_name;

-- Arquitectura

architecture architecture_name of entity_name is

	declaration of signals

	declaration of constants

	declaration of components

	declaration of types

	declaration of functions

	declaration of procedures

	declaration of packages

begin

	concurrent statements : when ... else

	concurrent statements : with ... select ... when

	concurrent statements : block

	sequential statements : process : if ... then ... else

	sequential statements : process : case

	sequential statements : process : loop

	sequential statements : process : wait

	sequential statements : process : next and exit

	sequential statements : process : assert

end architecture_name;
```

Como se ha dicho antes, VHDL sirve para describir un circuito electrónico, pero el mismo circuito puede ser descrito de las siguientes formas:

- Descripción de flujo de datos:
  A la hora de plantearse crear un programa en VHDL no hay que pensar como si fuera un programa típico de ordenador. VHDL es un lenguaje concurrente, como consecuencia las instrucciones se ejecutan todas a la vez.
  La instrucción básica de la ejecución concurrente es la asignación entre señales a través del signo "<=". Para facilitar la asignación de las señales, VHDL incluye elementos de alto nivel como son instrucciones condicionales, de selección, etc., como:

  `when ... else` `with ... select ... when` `block`

- Descripción de comportamiento:
  Como la programación concurrente no siempre es la mejor forma de describir comportamientos, VHDL incorpora la programación serie, la cual se define en bloques indicados con la sentencia process. Su sintaxis es la siguiente:

```
process (sensivity list)
        declaration of variables
begin
        sequential statements
end process;
```

  En un mismo diseño puede haber varios bloques de este tipo, cada uno de ellos corresponderá a una instrucción concurrente. Es decir, internamente la ejecución de las instrucciones de los procesos es secuencial, pero entre los bloques es concurrente. Dentro de los procesos se pueden incluir sentencias secuenciales como:

  `if ... then ... else` `case` `loop` `wait` `next` and `exit` `assert`

- Descripción estructural:
  Las dos descripciones anteriores son las más utilizadas ya que son más cercanas al pensamiento humano. Aunque existe otro tipo de descripción, que permite la realización de diseños jerárquicos. VHDL dispone de diferentes mecanismos para la descripción estructural.
  En VHDL es posible declarar componentes dentro de un diseño mediante la palabra component. Un componente se corresponde con una entidad que ha sido declarada en otro modulo del diseño, o incluso en alguna biblioteca. La declaración de este elemento se realiza en la parte declarativa de la arquitectura del módulo que se está desarrollando. La sintaxis para declarar un componente es muy parecida a la de una entidad:

```
component component_name is
        generic (generic_name : type := initialization;
                 generic_name : type := initialization);
        port (signal_name : mode type;
              signal_name : mode type);
end component;
```

El flujo de diseño de un sistema en VHDL es el siguiente:

- División del diseño principal en módulos separados:
  La modularidad es uno de los conceptos principales de todo diseño. Normalmente se diferencia entre dos metodologías de diseño: top-down y bottom-up. La metodología top-down consiste en que un diseño complejo se divide en diseños más sencillos que se puedan diseñar (o describir) más fácilmente. La metodología bottom-up consiste en construir un diseño complejo a partir de módulos más simples ya diseñados.
- Simulación funcional:
  Comprobaremos que lo escrito en el punto anterior realmente funciona como queremos, si no lo hace tendremos que modificarlo. En este tipo de simulación se comprueba que el código ejecuta correctamente lo que se pretende.
- Síntesis:
  En este paso se adapta el diseño anterior (que sabemos que funciona) a un hardware en concreto, ya sea una FPGA o un ASIC. Hay sentencias del lenguaje que no son sintetizables, como por ejemplo divisiones o exponenciales con números no constantes (no pueden ser transformadas a circuitos digitales). Durante la síntesis se tiene en cuenta la estructura interna del dispositivo, y se definen restricciones, como la asignación de pines. El sintetizador optimiza las expresiones lógicas con objeto de que ocupen menor área, o bien son eliminadas las expresiones lógicas que no son usadas por el circuito.
- Simulación post síntesis:
  En este tipo de simulación se comprueba que el sintetizador ha realizado correctamente la síntesis del circuito, al transformar el código HDL en bloques lógicos conectados entre sí. Este paso es necesario, ya que a veces, los sintetizadores producen resultados de síntesis incorrectos, o bien realizan simplificaciones del circuito al optimizarlo.
- Ubicación y enrutamiento:
  El proceso de ubicación consiste en situar los bloques digitales obtenidos en la síntesis de forma óptima, de forma que aquellos bloques que se encuentran muy interconectados entre si se sitúen próximamente. El proceso de enrutamiento consiste en interconectar adecuadamente los bloques entre sí, intentando minimizar retardos de propagación para maximizar la frecuencia de funcionamiento del dispositivo.
- Anotación final:
  Una vez ha sido completado el proceso de ubicación y enrutamiento, se extraen los retardos de los bloques y sus interconexiones, con objeto de poder realizar una simulación temporal (simulación post layout). Estos retardos son anotados en un

fichero SDF (*Standard Delay Format*) que asocia a cada bloque o interconexión un retardo mínimo/típico/máximo.

- Simulación temporal:
  A pesar de la simulación anterior, puede que el diseño no funcione cuando se programa, una de las causas puede ser por los retardos internos del chip. Con esta simulación se puede comprobar, y si hay errores se tiene que volver a uno de los anteriores pasos.
- Programación en el dispositivo:
  Se implementa el diseño en el dispositivo final y se comprueba el resultado.

En VHDL es posible describir modelos para la simulación. Estos modelos, conocidos como bancos de pruebas o *testbench*, no tienen ninguna restricción, necesitando únicamente un intérprete de las instrucciones VHDL. En cambio, en la síntesis se imponen restricciones como pueden ser aquellas que tienen que ver con las del tiempo, ya que no es posible aplicar retardos a la hora de diseñar un circuito.

El retraso es uno de los elementos más importantes de la simulación, puesto que el comportamiento de un circuito puede cambiar dependiendo del cambio de las diferentes señales. Cuando se realiza una asignación se produce de forma inmediata, puesto que no se ha especificado ningún retraso. Este comportamiento puede ser alterado mediante la opción after. También es posible introducir una espera entre dos secuencias mediante la palabra reservada wait.

Una de las partes más importantes en el diseño de cualquier sistema son las pruebas para la verificación del funcionamiento de un sistema. Con las metodologías tradicionales la verificación solo era posible tras su implementación física, lo que se traducía en un alto riesgo y coste adicional. Lo más sencillo es excitar las entradas para ver si las salidas responden según lo esperado, en una herramienta de simulación. Un banco de pruebas es una entidad sin puertos cuya estructura contiene un componente que corresponde al circuito que se desea simular y la excitación de las diferentes señales de entrada a dicho componente, para poder abarcar un mayor número de casos de prueba. Las siguientes líneas muestran la sintaxis de un banco de pruebas:

```
library ieee;

use ieee.std_logic_1164.all;

entity testbench_name is

end testbench_name;

architecture testbench of testbench_name is

begin

end testbench;
```

A continuación se muestran las diferentes estrategias que se pueden seguir para la realización de un banco de pruebas.

- Método tabular

  Para verificar la funcionalidad de un diseño se debe elaborar una tabla con las entradas y las respuestas que se esperan a dichas entradas. Todo ello se debe relacionar mediante código VHDL.
- Uso de ficheros (vectores de test)

  En el caso anterior, los casos de prueba y el código de simulación permanecían juntos, pero es posible separarlos de forma que, por un lado se encuentran las pruebas y por otro el código. Esto es posible ya que VHDL dispone de paquetes de entrada/salida para la lectura/escritura en ficheros de texto.
- Metodología algorítmica

  Existe otro tipo de test, los cuales se basan en realizar algoritmos para cubrir el mayor número de casos posibles.

5.2 ISE Design Suite, ISim y ModelSim

Para el desarrollo y comprobación del funcionamiento del código que se ha implementado, nos hemos servido de distintas herramientas software. En concreto un entorno de síntesis del diseño VHDL conocido como ISE Design Suite de Xilinx y dos entornos de simulación, uno de ellos es ISim, también perteneciente a Xilinx, para las primera etapas de desarrollo del código (desarrollo del núcleo de interpolación), y ModelSim de Mentor Graphics, para las etapas finales de implementación del código (interfaz de comunicación con el entorno del procesador), en las que ISim resultaba incompleto para la simulación requerida.

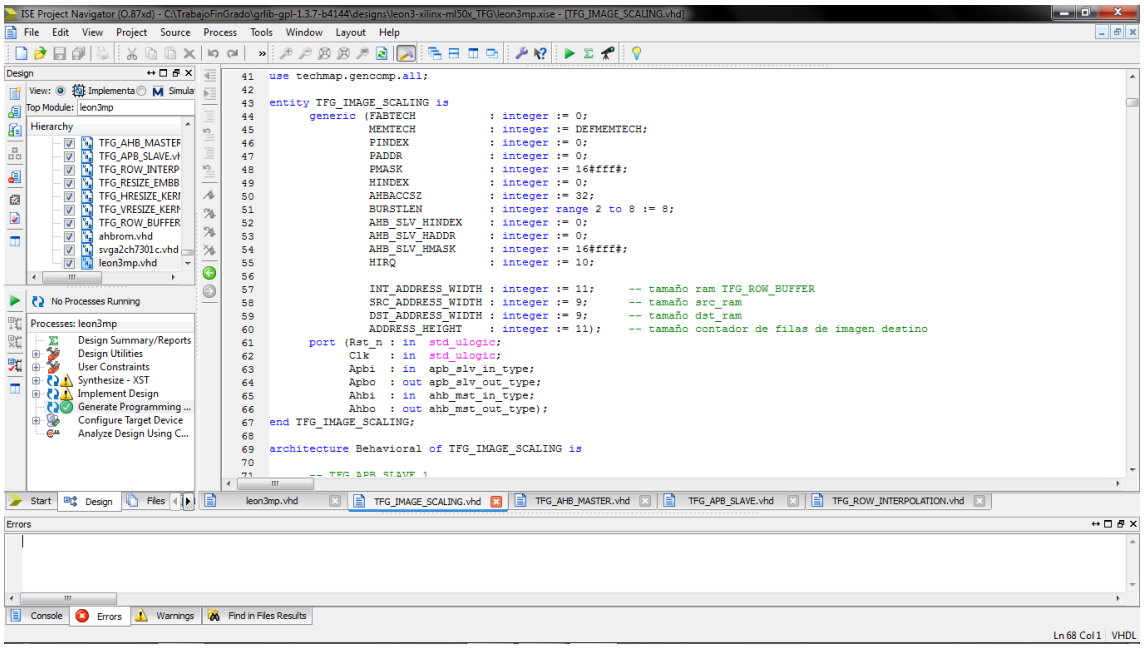


Imagen 5.1 – Entorno ISE Design Suite

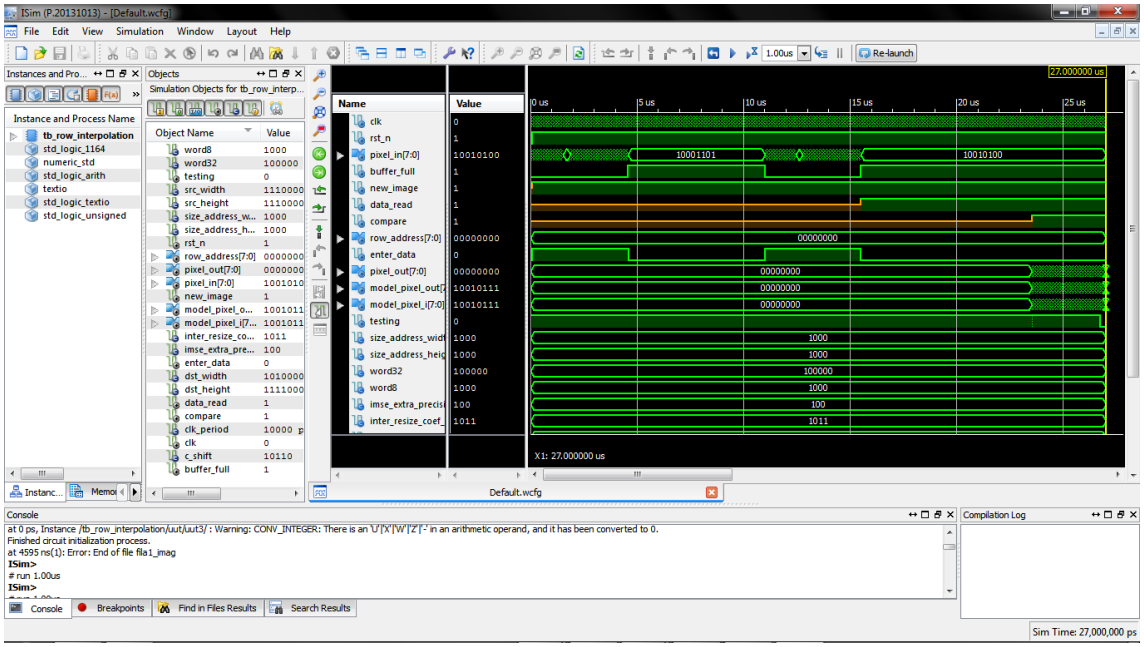


Imagen 5.2 – Entorno ISim

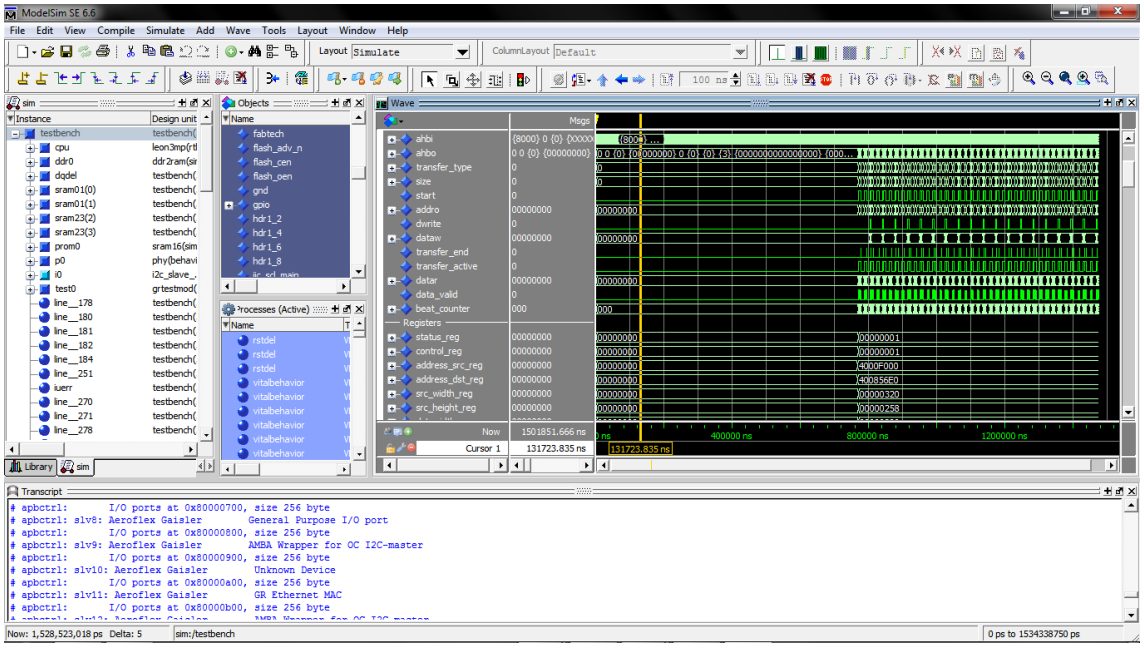


Imagen 5.3 – Entorno ModelSim

5.3 Virtex 5 FPGA Evaluation Platform ML505

Una vez concluida la etapa de desarrollo y simulación con efectividad, debemos pasar a portar el diseño a un hardware donde poder comprobar finalmente el correcto o no funcionamiento. Para ello se ha optado por una FPGA producida por Xilinx, concretamente la plataforma de desarrollo y evaluación *Virtex 5 FPGA Evaluation Platform ML505* [41].

Las características de la plataforma son las siguientes:

- Xilinx Virtex 5 FPGA XC5LX50T-1FFG1136 (ML505)

- Dos Xilinx XCF32P Platform Flash PROMs (32 Mb cada una) para almacenar las configuraciones de los dispositivos.
- Controlador de configuración Xilinx System ACE CompactFlash con conector Type I CompactFlash.
- Xilinx XC95144XL COLD para "lógica de pegamento".
- 256 MB DDR2 SODIMM de 64 bits de ancho, compatible con el soporte EDK para IP y controladores de software.
- Temporización:
    a) Chip generador del reloj del sistema programable.
    b) Un conector de oscilador para reloj de 3.3 V.
    c) Reloj externo vía SMA (dos pares diferenciales)
    - Interruptores DIP de propósito general (8), LEDs (8), pulsadores y un codificador rotatorio.
    - Conector de expansión con 32 terminales simples de E/S, 16 terminales LVDS de pares diferenciales y bus de expansión I^2C.
    - Códec de audio estéreo AC97 con conectores jack de entrada, salida, auriculares de 50 mW, micrófono, audio digital SPDIF y transductor piezoeléctrico.
    - Puerto serie RS 232, DB9 y conector para segundo puesto serie.
    - Display LCD de dos líneas y 16 caracteres.
    - Dispositivo EEPROM de 8 kb por I^2C.
    - Conector PS/2 para ratón y teclado.
    - Entrada/salida de video:
        a) Entrada de video (VGA).
        b) Conector DVI de salida de video (soporta VGA, conector incluido).
- ZBT SRAM síncrona, 9 Mb en bus de datos de 32 bits con cuatro bits de paridad.
- Intel P30 StrataFlash chip flash lineal (32 MB).
- Serial Peripheral Interface (SPI) flash (2 MB).
- Transceptor Ethernet PHY trivelocidad (10/100/1000)y conector RJ-45 con soporte para interfaces Ethernet PHY MII, GMII, RGMII, y SGMII.
- Chip de interfaz USB con puertos host y periféricos.
- Batería de litio recargable para mantener las claves de cifrado de la FPGA.
- Puerto de configuración JTAG para usar con Parallel Cable III, Parallel Cable IV o el cable de descarga de la plataforma USB.
- Fuentes de alimentación en placa con todas las tensiones necesarias.
- Chip de monitorización de temperatura y tensión con controlador de ventilador.
- Adaptador de 5V a 6A AC.
- Indicador LED de alimentación.
- Interfaces MII, GMII, RGMII y SGMII Ethernet PHY.
- GTP/GTX: SFP (1000Base-X).
- GTP/GTX: SMA (pares diferenciales RX y TX).
- GTP/GTX: SGMII.
- GTP/GTX: Conector PCI Express (PCIe).
- GTP/GTX: SATA (conexiones host dobles) con cable de bucle invertido.
- GTP/GTX: Clock synthesis ICs.
- Puerto de pruebas Mictor.

- Puerto de depuración BDM.
- Puerto sensible al tacto.
- Monitor de sistema.

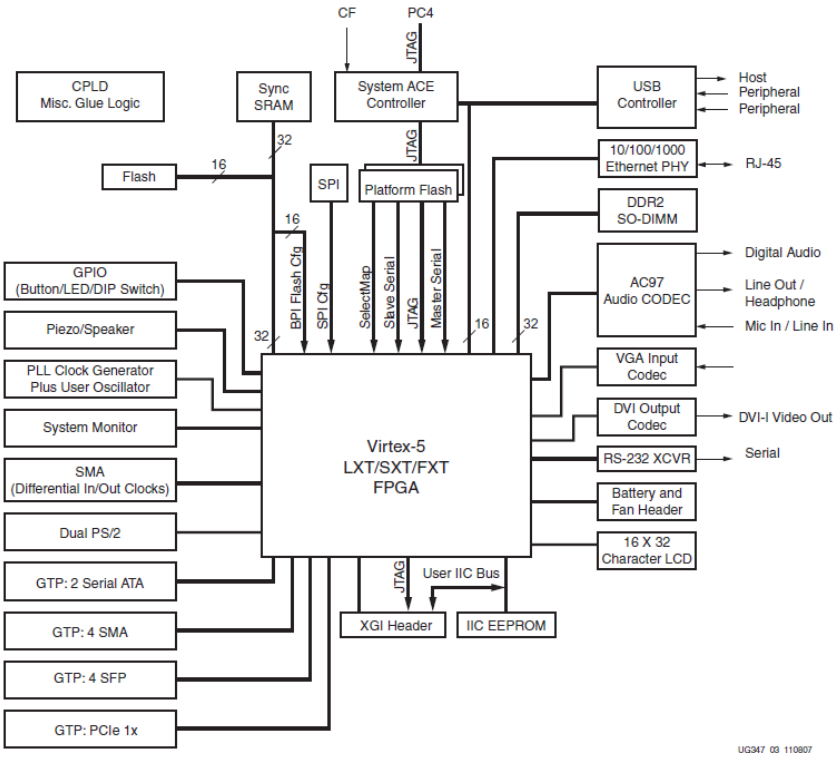


Imagen 5.4 – Características Virtex 5 FPGA Evaluation Platform ML505

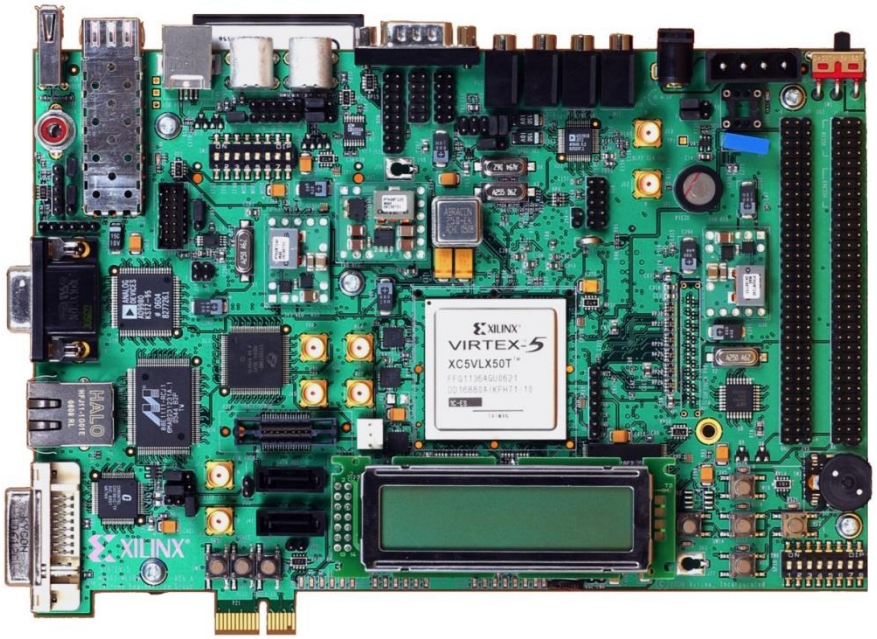


Imagen 5.5 – Vista frontal Virtex 5 FPGA Evaluation Platform [42]

Imagen 5.6 – Vista trasera Virtex 5 FPGA Evaluation Platform LM505 [43]

5.4 Procesador LEON3

El objetivo del presente trabajo consiste en el desarrollo de un módulo IP, y para que este pueda ser usado, requiere de un procesador que lo haga funcionar como esclavo en un sistema *SoC* (*System on Chip*). Para tal cometido se ha empleado el procesador LEON3, desarrollado por Aeroflex Gaisler [44], [45].

El LEON3 es un modelo de un procesador de 32 bits compatible con la arquitectura SPARC V8. El modelo es altamente configurable y especialmente adecuado para diseños *SoC* con soporte para configuraciones de multiprocesamiento. El procesador es totalmente sintetizable y se pueden implementar hasta 16 núcleos CPU en multiprocesamiento asimétrico (ASM) o multiprocesamiento síncrono (SMP).

El código fuente completo está disponible bajo licencia GNU GPL, lo que permite el uso gratuito e ilimitado para la investigación y la educación. También está disponible bajo licencia comercial de bajo coste, lo que le permite ser utilizado en cualquier aplicación comercial.

Las características del procesador LEON3 son las siguientes:

- Conjunto de instrucciones SPARC V8 con extensión V8e.
- Pipeline avanzado de 7 etapas.
- Unidades hardware de multiplicación, división y MAC.
- IEEE-754 FPU de alto rendimiento.
- Caché separada de instrucciones y datos (arquitectura Hardvard) con espionaje.
- Cachés configurables: 1 – 4 formas, 1 – 256 Kbytes/forma. Aleatorio, LRR o LRU.
- RAM local de instrucciones y datos, 1 – 256 Kbytes.
- SPARC Reference MMU (SRMMU) con TLB configurable.
- Interfaz de bus AMBA 2.0 AHB.
- Soporte avanzado de depuración en chip con buffer de instrucciones y datos.

- Soporte multiprocesador simétrico (SMP).
- Modo apagado y reducción de consume del reloj.
- Diseño robusto y totalmente síncrono activo por flanco de reloj.
- Hasta 125 MHz en FPGA y 400 MHz en tecnologias ASIC de 0.13 µm.
- Tolerancia a fallos y version SEU de prueba para aplicaciones espaciales.
- Ampliamente configurable.
- Amplia gama de herramientas software: compiladores, kernels, simuladores and monitores de depuración.
- Alto rendimiento: 1.4 DMIPS/MHz, 1.8 CoreMark/MHz (gcc -4.1.2)

El procesador LEON3 se distribuye como parte de la biblioteca GRLIB IP, lo que permite una integración sencilla en los diseños *SoC*. GRLIB también incluye una amplia gama de módulos periféricos.

El procesador LEON3 es totalmente parametrizable a través del uso de genéricos VHDL. Es posible crear instancias de varios núcleos de procesador en el mismo diseño con diferentes configuraciones. Los diseños de las plantillas de LEON3 se pueden configurar mediante una herramienta grafica integrada denominada *Xconfig* (se puede ejecutar con el comando *make xconfig*). Esto permite a los nuevos usuarios definir rápidamente una configuración personalizada. La herramienta de configuración no solo configura el procesador, sino también otros periféricos en el chip, tales como controladores de memoria o interfaces de red.

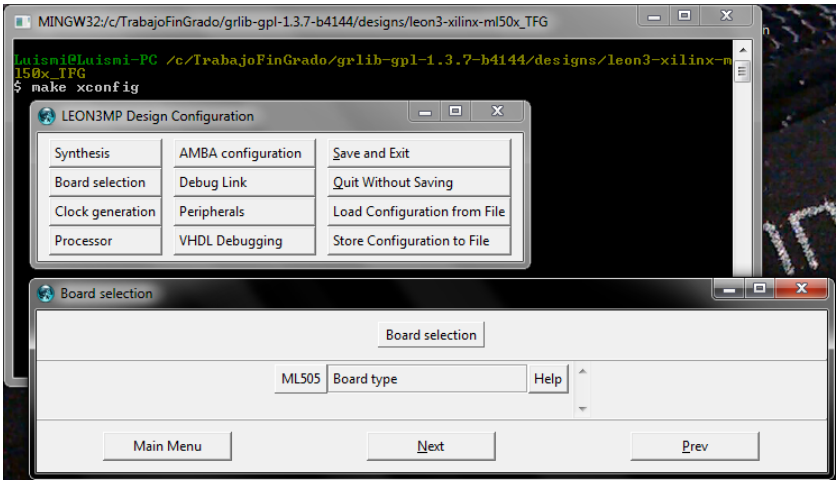


Imagen 5.7 – Selección de placa para el procesador LEON3

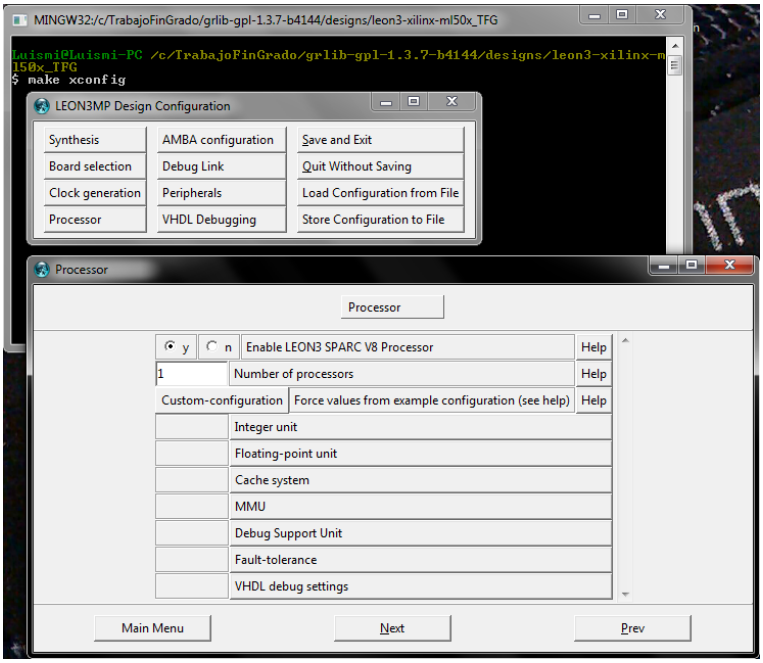


Imagen 5.8 – Configuración del procesador LEON3

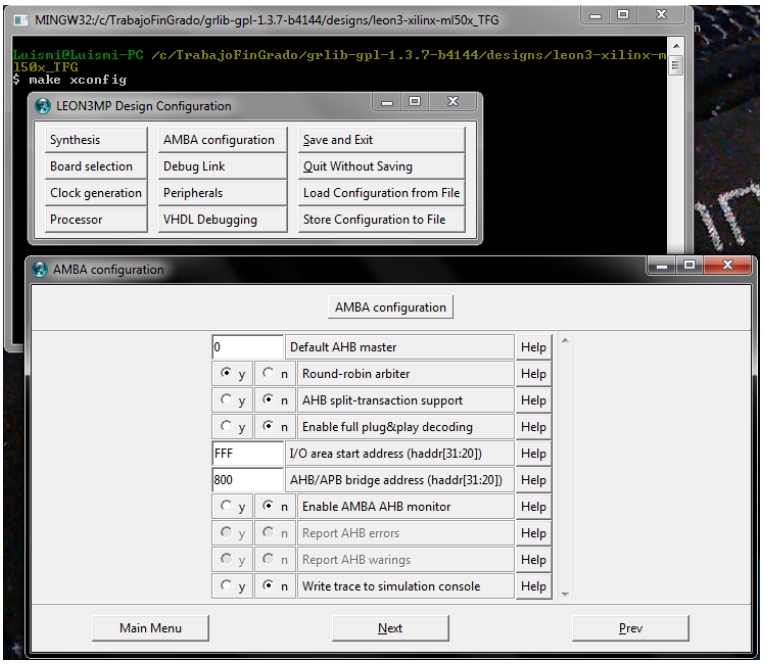


Imagen 5.9 – Configuración del AMBA bus en el procesador LEON3

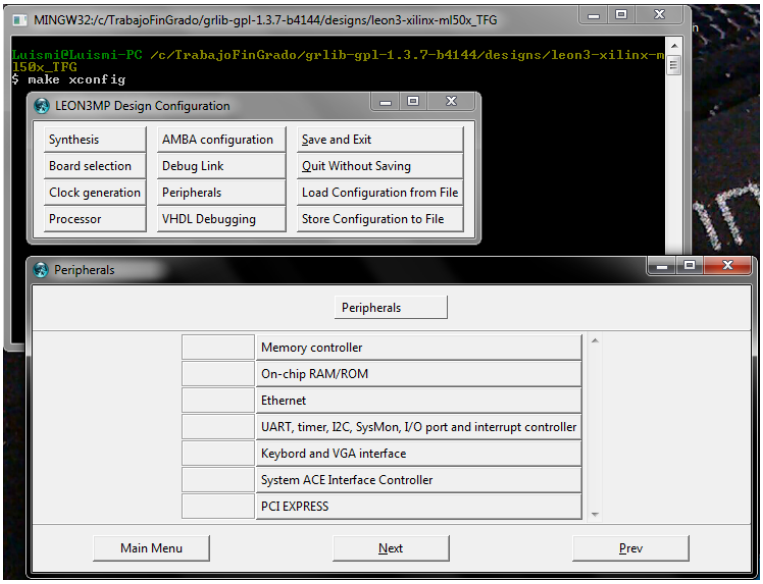


Imagen 5.10 – Configuración de los periféricos del procesador LEON3

El procesador LEON3 se puede sintetizar con herramientas de síntesis comunes como Synplify, Synopsys DC y Cadence RC. El núcleo alcanza hasta 125 MHz en FPGA y 400 MHz en tecnologías ASIC de 0.13 µm. La zona del núcleo requiere solo 20 a 25 Kgates o 35000 LUT, dependiendo de la configuración. El procesador LEON3 también se puede sintetizar con Xilinx XST y Quartus II, ya sea a través de secuencias de comandos o mediante el uso de las correspondientes interfaces gráficas.

Para simplificar el desarrollo de software, Aeroflex Gaisler proporciona BCC, un compilador cruzado libre de C/C++ basado en gcc y la biblioteca en C Newlib. La depuración se realiza generalmente utilizando el depurador GDB y una interfaz gráfica como DDD (*Data Display Debugger*) o, en nuestro caso, Eclipse.

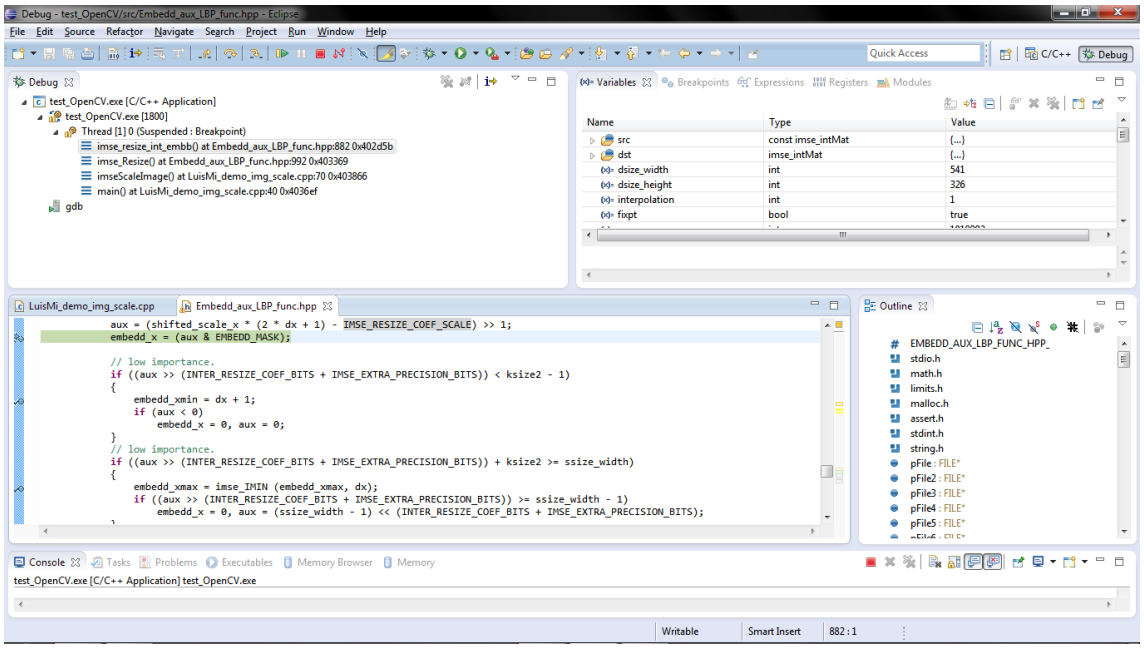


Imagen 5.11 – Entorno de depuración de Eclipse

Es posible llevar a cabo una depuración simbólica, ya sea en un simulador o mediante el uso de hardware real. Aeroflex Gaisler ofrece Tsim, un simulador de LEON3 de alto rendimiento que se puede conectar a GDB y emular un sistema LEON3 a más de 30 MIPS. La interfaz de monitor GRMON es la unidad de apoyo para la depuración del procesador LEON3 (DSU, *Debug Support Unit*), que incluye una amplia gama de funciones de depuración, así como una entrada GDB.

Un diseño típico basado en el procesador LEON3 consiste en un sistema con un núcleo del procesador LEON3 y un conjunto de módulos IP conectados a través de los buses AHB y APB.

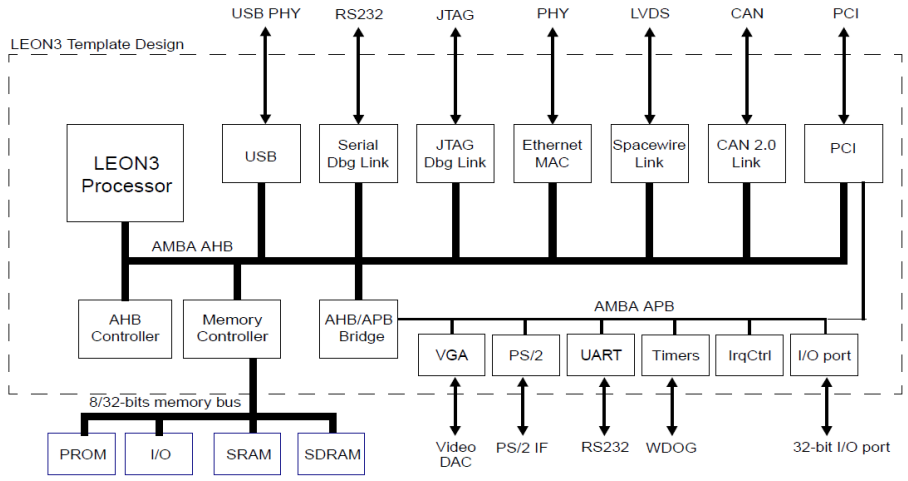


Imagen 5.12 – Configuración típica con un procesador LEON3

5.4.1 Arquitectura del procesador LEON3

El núcleo del LEON3 tiene las siguientes características principales: pipeline de 7 niveles con arquitectura Harvard, cachés separadas de instrucciones y datos (I-Cache y D-Cache), unidad de gestión de memoria (SRMMU), multiplicador y divisor hardware (HW MUL/DIV), soporte de depuración en chip y extensiones para multiprocesador [46].

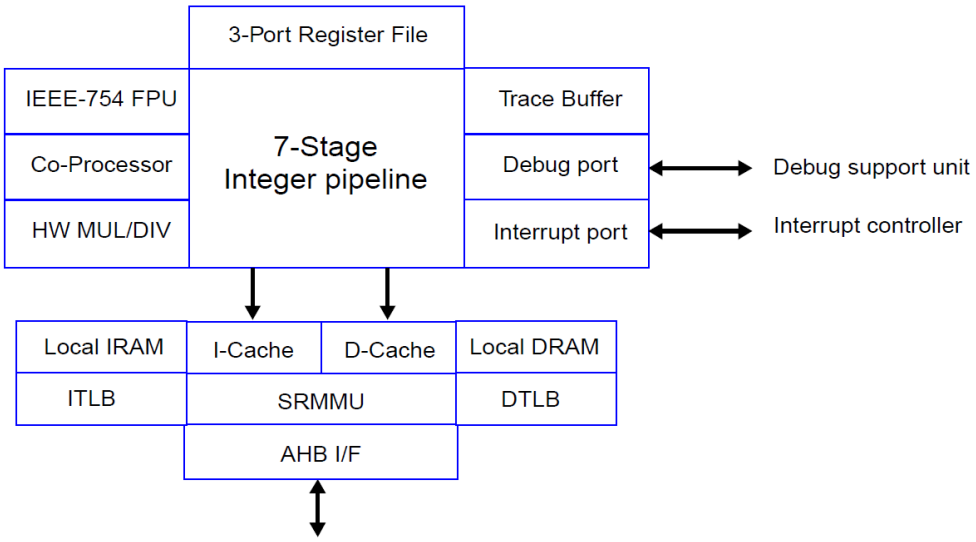


Imagen 5.13 – Diagrama de bloques básicos del procesador LEON3

La Integer Unit (unidad entera) implementa la parte entera del conjunto de instrucciones de la arquitectura SPARC V8. Su implementación está centrada en el alto rendimiento y la baja complejidad, incluyendo las instrucciones hardware de multiplicación y división.

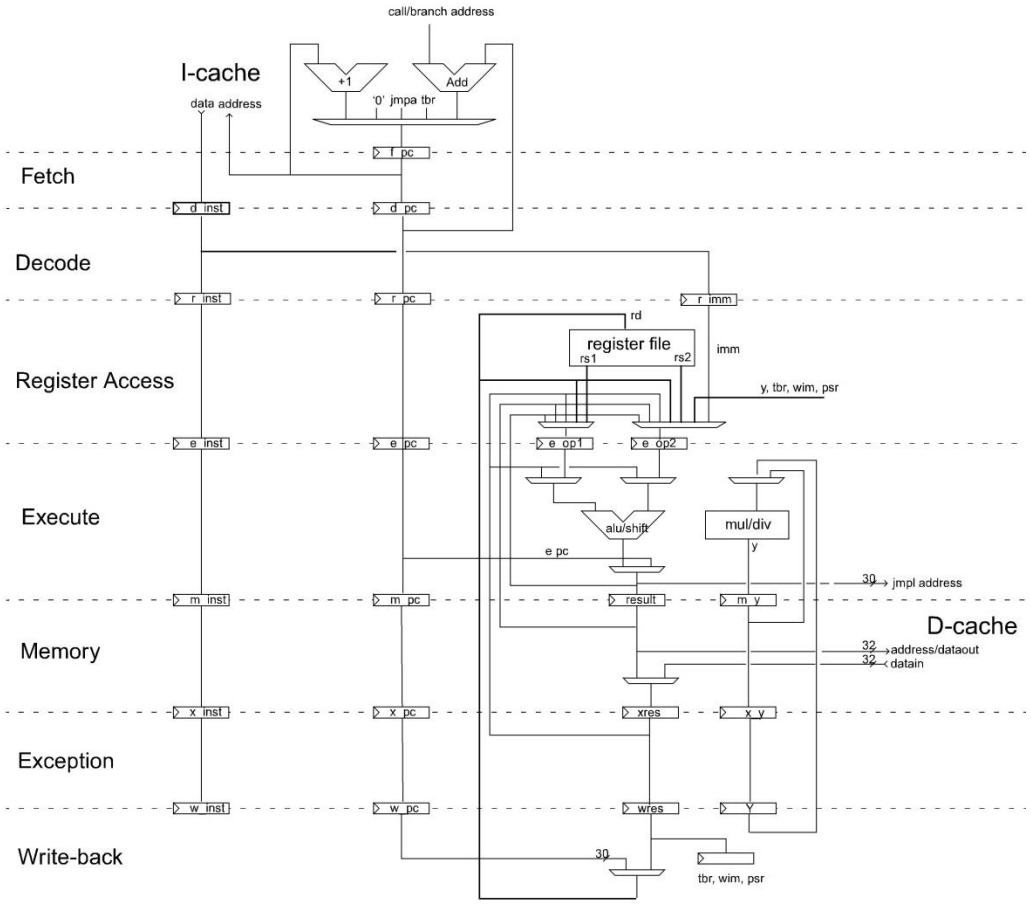


Imagen 5.14 – Diagrama de bloques de la Integer Unit

La Integer Unit del procesador LEON3 implementa una arquitectura Harvard con buses separados de instrucciones y datos, conectados a dos controladores de caché separadas. Esto deriva en un sistema de caché altamente configurable.

El núcleo del procesador LEON3 puede ser opcionalmente configurado con una unidad de gestión de la memoria (SRMMU) compatible con la arquitectura SPARC V8. La MMU proporciona el mapeo entre los espacios de direcciones virtuales y la memoria física.

La Integer Unit ofrece interfaces para dos coprocesadores (opcionales), la unidad de coma flotante (para la FPU de alto rendimiento, GRFPU) y un coprocesador definido por el usuario (para la GRFPU Lite). Estos se ejecutan en paralelo con la Integer Unit. La GRFPU funciona con operandos de simple y doble precisión e implementa todas las operaciones de la SPARC V8 FPU. La GRFPU Lite es una versión reducida de la GRFPU adecuada para implementaciones en FPGA con recursos lógicos limitados.

5.5 AMBA bus

La idea de escoger el procesador LEON3 para realizar la implementación de nuestro IP surge a partir de las características de este, partiendo de la total libertad de utilización de su código, y finalizando en la arquitectura de interconexión con la que este ha sido desarrollado, el AMBA bus (*Advanced Microcontroller Bus Architecture*), en concreto la versión 2.0 [47].

La especificación AMBA define un estándar para la comunicación en chip de microcontroladores de alto rendimiento. Se definen tres buses distintos dentro de la especificación AMBA bus 2.0:

- AMBA AHB: bus de sistema de alto rendimiento para módulos de alta frecuencia de reloj. Actúa como bus central de sistemas de alto rendimiento. Soporta la conexión eficiente de procesadores, memorias internas (*on-chip*) e interfaces externas.
- AMBA ASB: bus de sistema para módulos de alto rendimiento. Es un bus alternativo al AHB bus, adecuado para cuando no se requieren las características de alto rendimiento del AHB bus. También soporta la conexión eficiente de procesadores, memorias internas e interfaces externas.
- AMBA APB: bus para periféricos de bajo consumo. Está optimizado para un consumo mínimo de energía y la reducción de la complejidad de la interfaz que da soporte a los periféricos. Puede ser usado en conjunto con cualquiera de los buses de sistema anteriores.

La especificación AMBA tiene como objetivo satisfacer cuatro requisitos fundamentales:

- Facilitar el desarrollo inicial de microcontroladores con uno o más CPU o procesadores de señal.
- Intentar ser independiente de la tecnología para asegurar su funcionamiento en distintos sistemas, tanto estándar como desarrollados a medida.
- Promover el diseño modular del sistema para mejorar la independencia del procesador.
- Reducir al mínimo la infraestructura de silicio requerida para soportar las comunicaciones internas y externas al chip.

Un microcontrolador (o procesador en nuestro caso) basado en AMBA consiste en un sistema con un bus central de alto rendimiento (AMBA AHB o AMBA ASB), capaz de soportar el ancho de banda de la memoria externa, la CPU, el chip de memoria y otros dispositivos (DMA) conectados a él . Proporciona una interfaz de alto rendimiento entre los elementos que están implicados en la mayoría de las transferencias. También implementa un puente para la comunicación con el bus de ancho de banda inferior APB, donde se encuentran la mayoría de los dispositivos periféricos en el sistema, como podemos apreciar en la siguiente imagen.

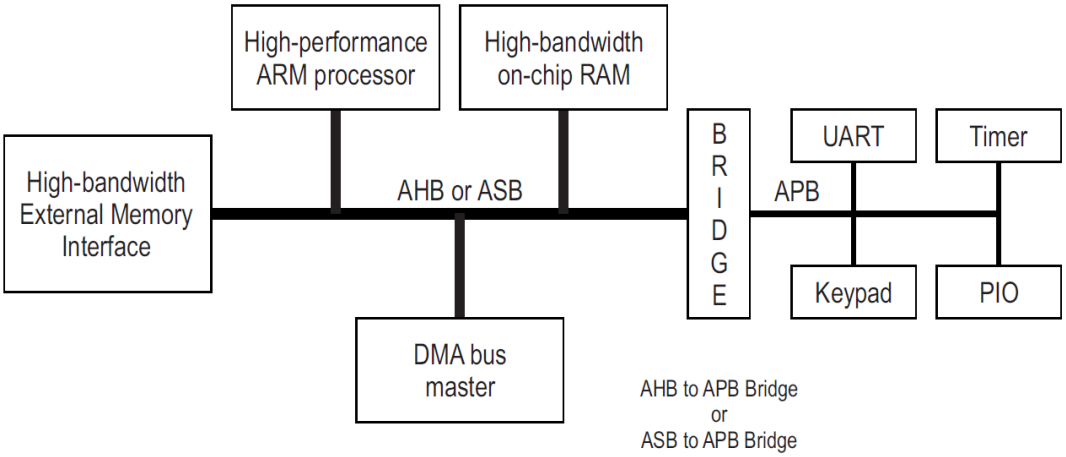


Imagen 5.15 – Configuración típica de un microcontrolador basado en AMBA

A continuación describimos las dos interfaces que se han implementado en nuestro IP.

5.5.1 AMBA AHB

El AHB bus es una nueva generación de AMBA bus que está destinada a hacer frente a las exigencias de los diseños sintetizables de alto rendimiento. Es un bus de sistema que soporta múltiples maestros de bus y proporciona un funcionamiento de alto ancho de banda.

El AMBA AHB implementa las características necesarias para un alto rendimiento en sistemas de reloj de alta frecuencia incluyendo:

- Transferencias ráfaga.
- Transacciones divididas.
- Ciclo único de entrega del bus al maestro.
- Funcionamiento por flanco de un único reloj.
- Implementación sin triestado.
- Configuraciones de datos de bus más amplia (64/128 bits).

Se puede tender un puente entre este nivel superior de bus y el APB bus de manera eficiente para asegurar que los diseños existentes pueden integrarse fácilmente.

Un diseño AMBA AHB puede contener uno o más maestros de bus, por lo general un sistema contendría al menos el procesador y la interfaz de prueba. Sin embargo, también es común tener un acceso directo a memoria (DMA) o procesador de señal digital (DSP) incluido en el bus principal.

La interfaz de memoria externa, el puente APB y cualquier memoria interna son los esclavos AHB más comunes. Cualquier otro periférico en el sistema también podría incluirse como un esclavo AHB. Sin embargo, los periféricos de bajo ancho de banda residen normalmente en el APB bus.

Un diseño tipico de un sistema AMBA AHB contiene los siguientes componentes:

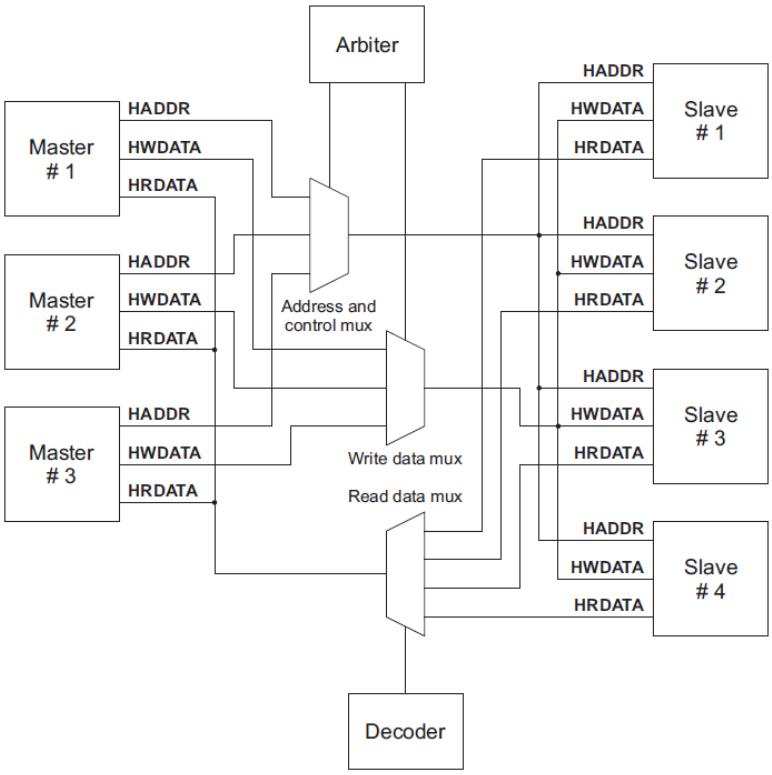


Imagen 5.16 – Componentes de un sistema AMBA AHB

- AHB master: el maestro es capaz de iniciar las operaciones de lectura y escritura mediantes su dirección e información de control. Solo se permite a un maestro utilizar el bus en un momento dado.

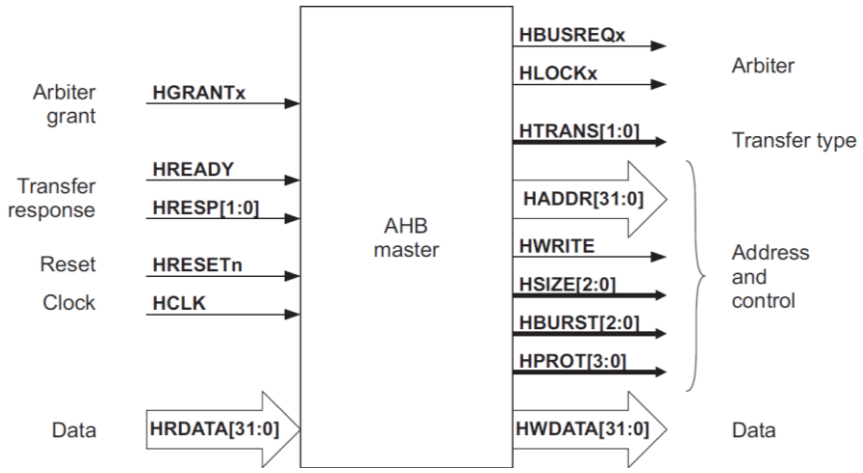


Imagen 5.17 – Interfaz AHB master

- AHB slave: el esclavo responde a una operación de lectura o escritura requerida por el maestro. Las señales del esclavo indican al maestro el éxito, fracaso o espera de la transferencia de datos.

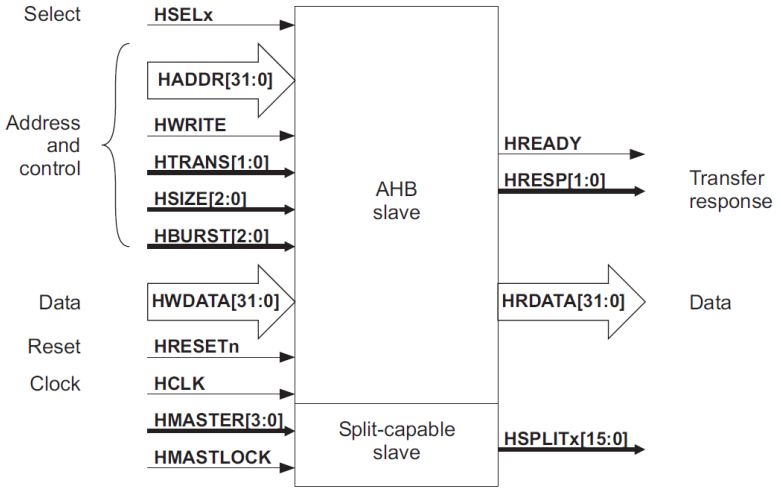


Imagen 5.18 – Interfaz AHB slave

- AHB arbiter: el árbitro del bus asegura que solo se permite a un maestro iniciar la transferencia de datos. A pesar de que el protocolo de arbitraje es fijo, se puede implementar cualquier algoritmo de arbitraje, como máxima prioridad o acceso equitativo, en función de los requisitos de la aplicación.

  El AHB bus incluye un único árbitro, aunque este será trivial en sistemas con único maestro.

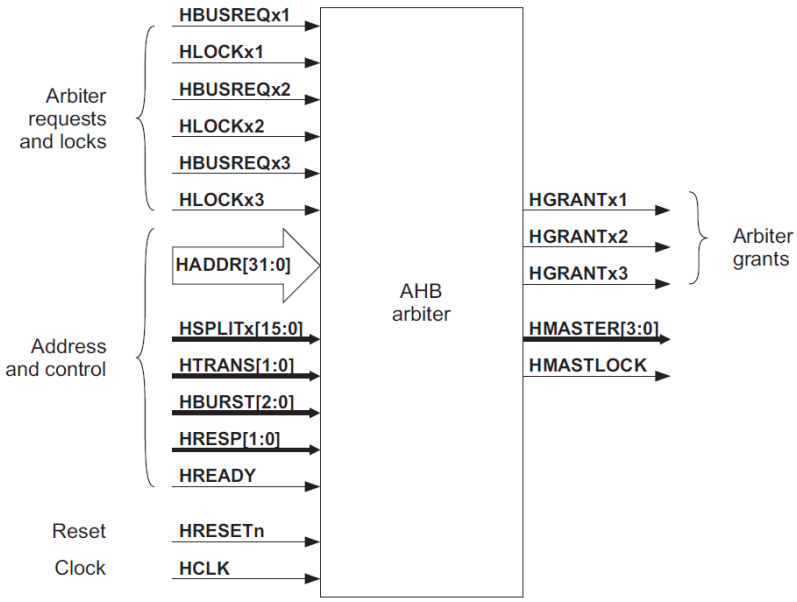


Imagen 5.19 – Interfaz AHB arbiter

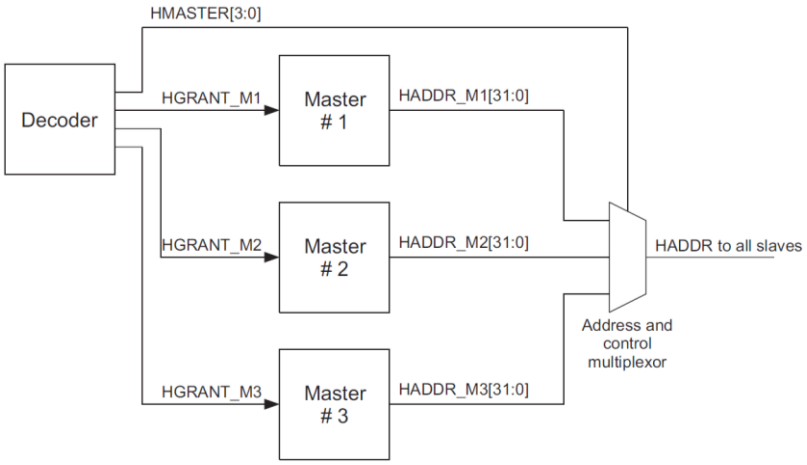


Imagen 5.20 – Señales de concesión del bus al master

- AHB decoder: el decodificador se usa para decodificar la dirección de cada transferencia y proporciona una señal de selección para el esclavo que está involucrado en la transferencia.
  Se requiere un solo decodificador centralizado en todas las implementaciones AHB.

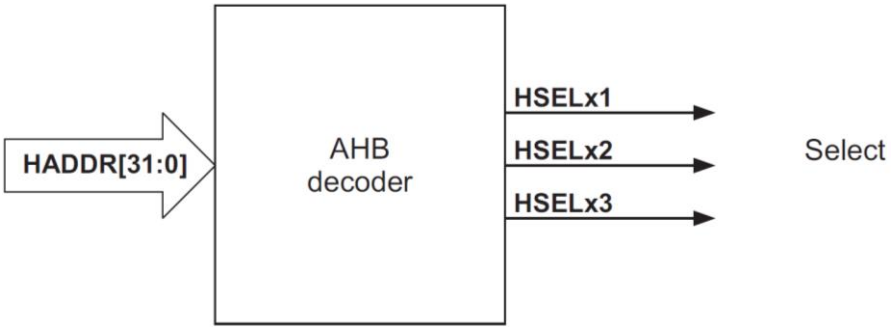


Imagen 5.21 – Interfaz AHB decoder

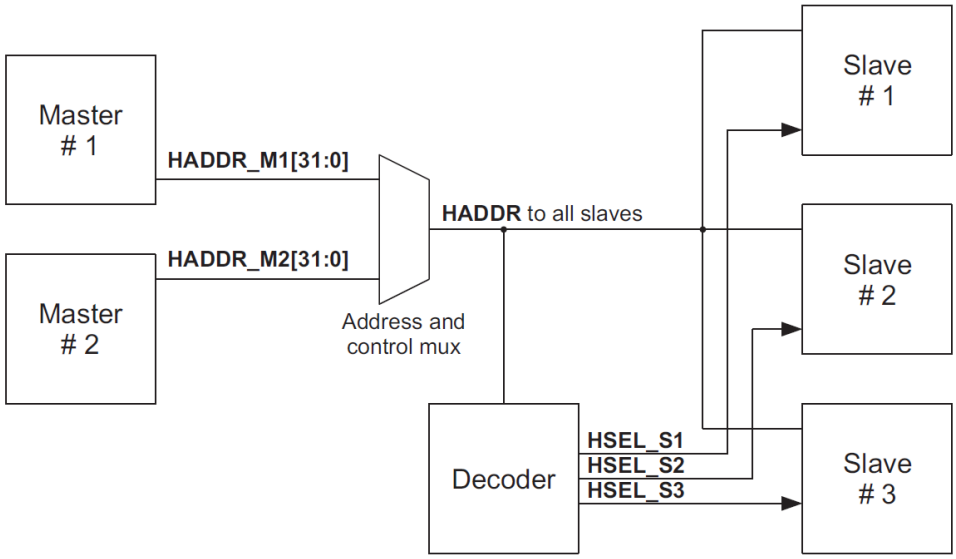


Imagen 5.22 – Señales de selección de esclavo

A continuación describimos las señales que componen el AHB bus.

Señal	Descripción
HCLK	Este reloj sincroniza todas las transferencias del bus. Todas las señales están relacionadas con el flanco de subida de HCLK.
HRESETn	La señal de reset del bus es activa en nivel bajo y se utiliza para reiniciar el sistema y el bus.
HADDR [31:0]	Bus de dirección de 32 bits del sistema.
HTRANS [1:0]	Indica el tipo de transferencia actual, que puede ser secuencial, asíncrona, ausente u ocupado.
HWRITE	Cuando está en nivel alto indica que es una transferencia de escritura y cuando está en nivel bajo es una transferencia de lectura.
HSIZE [2:0]	Indica el tamaño máximo de la transferencia, que es típicamente un byte (8 bits), *halfword* (media palabra, 16 bits) o una palabra (32 bits). El protocolo permite la transferencia de tamaños más grandes hasta un máximo de 1024 bits.
HBURST [2:0]	Indica si la transferencia forma parte de una ráfaga. Soporta cuatro, ocho y dieciséis disparos.
HPROT [3:0]	Las señales de control de protección proporcionan información sobre un acceso al bus y están destinadas principalmente para su uso por cualquier módulo que desee aplicar un cierto nivel de protección.
HWDATA [31:0]	El bus de escritura de datos se utiliza para transferir datos del maestro a los esclavos del bus durante las operaciones de escritura. Se recomienda un ancho mínimo del bus de datos de 32 bits. Sin embargo, este es fácilmente extensible para permitir la operación con mayor ancho de banda.
HSELx	Cada esclavo AHB tiene su señal de selección propia y esta indica que la transferencia actual está destinada al esclavo seleccionado. Es simplemente una decodificación combinacional del bus de direcciones.
HRDATA [31:0]	El bus de lectura de datos se utiliza para transferir datos de los esclavos al maestro del bus durante las operaciones de lectura. Se recomienda un ancho mínimo del bus de datos de 32 bits. Sin embargo, este es fácilmente extensible para permitir la operación con mayor ancho de banda.
HREADY	Cuando se encuentra en nivel alto indica que la transferencia ha terminado en el bus. Puede ser llevada a nivel bajo para extender una transferencia. Nota: los esclavos en el bus requieren una señal de HREADY tanto de entrada como de salida.
HRESP [1:0]	Proporciona información adicional sobre el estado de una transferencia. Proporciona cuatro respuestas diferentes: correcta, error, reintentar y detenida.
HBUSREQx	Señal del maestro x del bus para el árbitro del bus que indica que este requiere el bus. Hay una señal de HBUSREQx para cada maestro del bus, con hasta un máximo de 16 maestros de bus.
HLOCKx	Cuando esta señal se encuentra en nivel alto indica que el maestro requiere bloquear el acceso al bus y ningún otro maestro puede acceder al bus hasta que esta señal este en nivel bajo.
HGRANTx	Esta señal indica que el maestro x del bus es actualmente el maestro con mayor prioridad. La propiedad de las señales de dirección/control cambian con el final de una transferencia cuando HREADY está en nivel alto, por lo que un maestro tiene acceso al bus cuando HREADY y HGRANTx están en nivel alto.
HMASTER [3:0]	Esta señal del árbitro indica que un maestro del bus está realizando una transferencia y es utilizada por los esclavos que soportan las transferencias *SPLIT* para determinar que maestro está intentando acceder. La sincronización de HMASTER está alineada con la sincronización de las señales de dirección y control.
HMASTLOCK	Indica que el maestro actual está realizando una secuencia de bloqueo de las transferencias. Esta señal tiene la misma sincronización que HMASTER.
HSPLITx [15:0]	Este bus de 16 bits es utilizado por un esclavo para indicar al árbitro que el maestro del bus puede volver a intentar una transacción detenida.

Tabla 5.1 – Señales del AMBA AHB

5.5.2 AMBA APB

El APB bus es parte de la jerarquía de buses AMBA y esta optimizado para un mínimo consumo de energía y la reducción de la complejidad de la interfaz AHB.

El AMBA APB aparece como un bus local secundario que se encapsula como un dispositivo esclavo del AHB bus. El APB bus proporciona una extensión de baja potencia al bus del sistema basándose directamente en señales del AHB bus.

El puente APB aparece como un módulo esclavo que se encarga del *handshake* (establecimiento de la comunicación) del bus y del control de la señal de retraso en representación de los periféricos del bus local.

El AMBA APB debe ser usado para interconectar todos los periféricos que tienen un bajo ancho de banda y no requieren el alto rendimiento de una interfaz de bus pipeline.

Todas las transiciones de señales están relacionadas con el flanco de subida del reloj. Esto asegura que los periféricos del APB bus pueden integrarse fácilmente en cualquier flujo de diseño y que sea más sencillo interconectar con la interfaz AHB.

Una implementación AMBA APB contiene típicamente un solo puente APB que se requiere para convertir las transferencias AHB en un formato adecuado para los dispositivos esclavos del APB bus. El puente ofrece conexión a todas las señales de direcciones, datos y control, así como proporcionar un segundo nivel de decodificación para generar las señales de selección de los periféricos esclavos del APB bus.

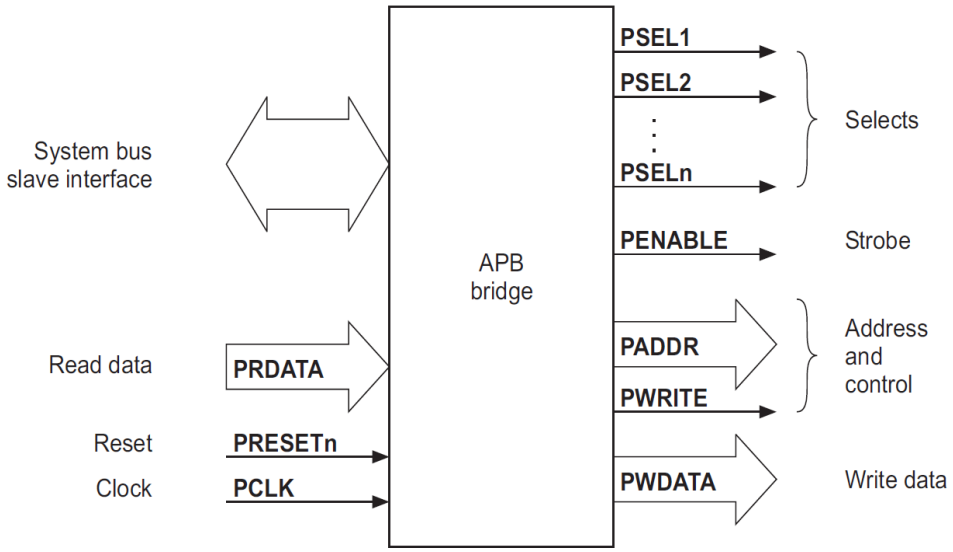


Imagen 5.23 – Interfaz APB bridge

Todos los demás módulos del APB bus son esclavos. Los esclavos del APB bus tienen la siguiente especificación de interfaz:

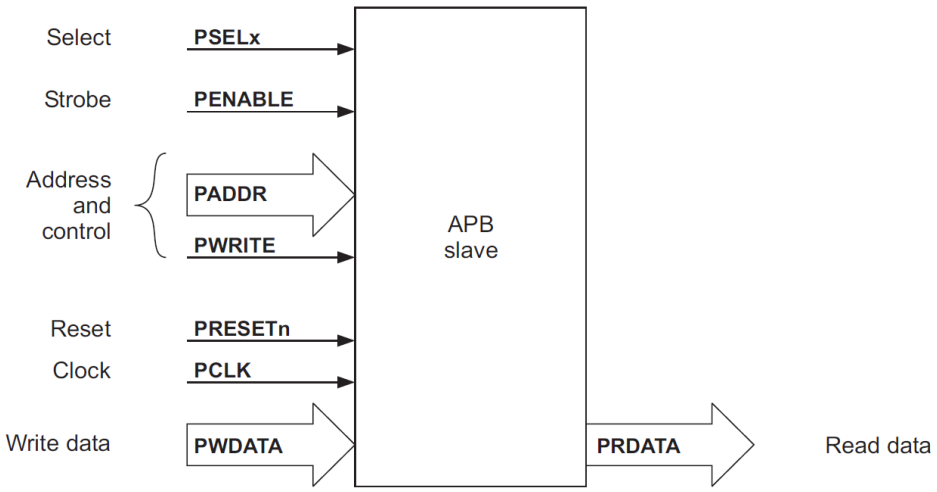


Imagen 5.24 – Interfaz APB slave

- Dirección y control valido en todo el acceso (sin pipeline).
- Interfaz de consumo cero cuando no existe actividad en el bus de periféricos.
- La temporización puede ser proporcionada por un decodificador con sincronización estroboscópica.
- Escritura de datos válidos para todo el acceso.

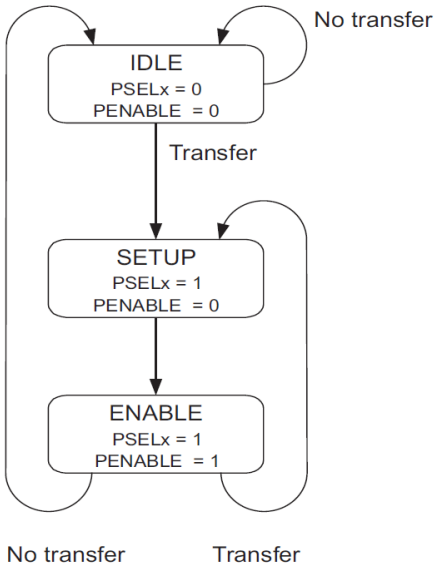


Imagen 5.25 – Diagrama de estados de actividad del bus de periféricos

A continuación describimos las señales que componen el APB bus.

Señal	Descripción
PCLK	El flanco de subida de PCLK se utiliza para sincronizar todas las transferencias en el APB bus.
PRESETn	La señal de reinicio del APB bus es activa en nivel bajo y normalmente se conecta a la señal de reset del bus del sistema.
PADDR [31:0]	El bus de direcciones APB, que puede ser de hasta 32 bits de ancho, es controlado por el puente del bus de periféricos.
PSELx	Señal del decodificador secundario, dentro del puente del bus de periféricos, conectada a cada esclavo del bus de periféricos. Indica que se ha seleccionado el dispositivo esclavo y se requiere una transferencia de datos. Hay una señal de PSELx para cada esclavo del bus.
PENABLE	Esta señal se usa para sincronizar todos los accesos al bus de periféricos. Se utiliza para indicar el segundo ciclo de una transferencia en el APB bus. El flanco de subida de PENABLE se produce en medio de la transferencia.
PWRITE	Cuando esta señal está en nivel alto indica un acceso de escritura y cuando está en nivel bajo un acceso de lectura.
PRDATA [31:0]	El bus de lectura de datos está controlado por el esclavo seleccionado durante los ciclos de lectura (cuando PWRITE está en nivel bajo). El bus de lectura de datos puede tener un ancho máximo de hasta 32 bits.
PWDATA [31:0]	El bus de escritura de datos está controlado por el puente del bus de periféricos durante los ciclos de escritura (cuando PWRITE está en nivel alto). El bus de escritura de datos puede tener un ancho máximo de hasta 32 bits.

Tabla 5.2 – Señales del AMBA APB

Una vez descritos los programas que nos han ayudado con la descripción del código y las características del procesador empleado para el desarrollo de nuestra aplicación, pasamos a describir el código implementado. Empezamos para ello con la descripción estructural del módulo IP y a continuación indagamos más en la sintaxis del código, describiendo este desde un punto de vista de comportamiento casi a nivel de flujo de datos.

5.6 Descripción a nivel de componentes

La descripción a nivel de componentes consiste en lo que se conoce en el entorno de la programación VHDL como descripción estructural, la cual da un punto de vista de la jerarquía seguida en la implementación del algoritmo. Así pues, se tiene una descripción del código que parte del nombre de cada uno de los ficheros el VHDL que comprende trabajo hasta (y cierto límite) los demás elementos que se consideren oportunos y que se han detectado en la síntesis del código realizada por ISE, como pueden ser registros, memorias, multiplexores…

Antes de pasar a describir los componentes, dedicamos un momento a relacionar los distintos elementos que aparecerán de aquí en adelante.

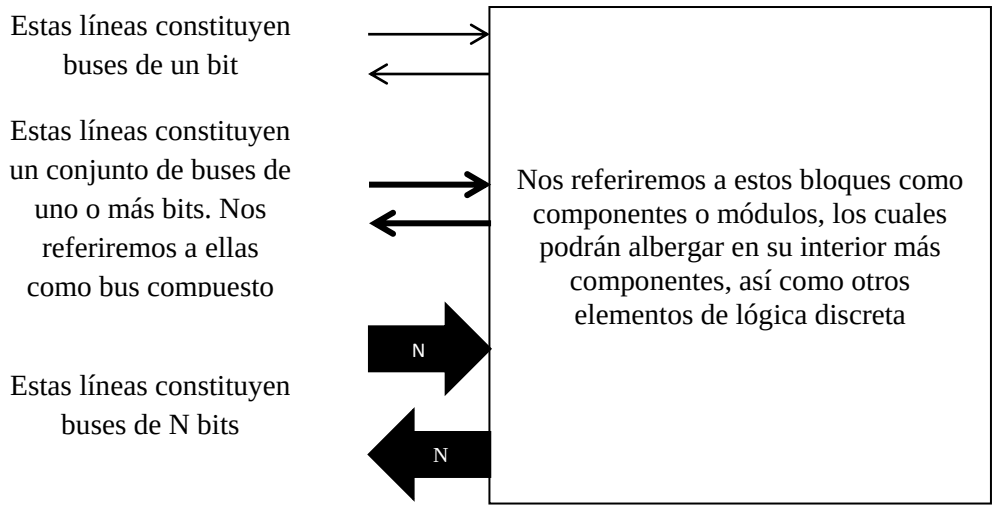


Imagen 5.26 – Interfaz genérica de componentes

5.6.1 leon3mp

Este es el módulo superior del proyecto creado para ISE. En él aparecen las instanciaciones de todos y cada uno de los componentes conectados al procesador. Aquí pues, está la instanciación del módulo IP desarrollado en este trabajo, denominado TFG_IMAGE_SCALING. En el siguiente apartado vemos la descripción de esta entidad y a continuación los demás componentes y elementos que se ha implementado dentro de este. No es cometido de este trabajo describir la entidad del procesador.

5.6.2 TFG_IMAGE_SCALING

Es aquí donde comienza el desarrollo del módulo IP de escalado de imágenes. Este módulo contiene las líneas de reloj y reset para el correcto funcionamiento de los componentes internos y las interfaces necesarias para la comunicación con los distintos buses de la arquitectura AMBA bus. Así pues, la función más importante de este consiste en realizar los accesos a la memoria externa donde se encontrará la imagen fuente y donde también se ubicará la imagen destino, además de controlar el arranque y detención del algoritmo de escalado. La descripción de la interfaz de TFG_IMAGE_SCALING es la siguiente:

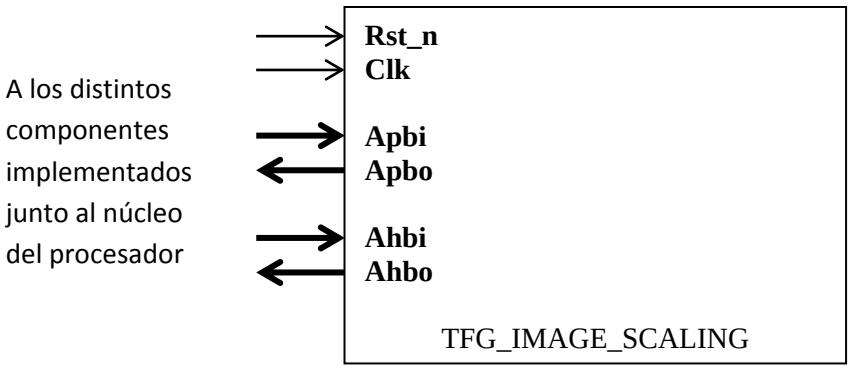


Imagen 5.27 – Interfaz de TFG_IMAGE_SCALING

A continuación, pasamos a describir la interfaz de nuestra entidad principal. Comenzamos para ello con los genéricos.

Genérico	Descripción
FABTECH	Genérico relacionado con la tecnología del fabricante del hardware.
MEMTECH	Genérico relacionado con la tecnología del fabricante del hardware.
PINDEX	Descripción en apartado 5.6.4.
PADDR	Descripción en apartado 5.6.4.
PMASK	Descripción en apartado 5.6.4.
HINDEX	Descripción en apartado 5.6.3.
AHBACCSZ	Genérico relacionado con el ancho del AHB bus.
BURSTLEN	Genérico relacionado con las transferencias de datos en memoria.
AHB_SLAVE_INDEX	Genérico relacionado con la configuración de los AHB slave.
AHB_SLAVE_HADDR	Genérico relacionado con la configuración de los AHB slave.
AHB_SLAVE_HMASK	Genérico relacionado con la configuración de los AHB slave.
HIRQ	Descripción en apartado 5.6.4.
INT_ADDRESS_WIDTH	Este genérico, junto con los tres siguientes, permiten parametrizar nuestro diseño para ajustarlo a las dimensiones de las imágenes con las que vaya a ser empleado. Sirve para determinar el tamaño de la memoria implementada en TFG_ROW_BUFFER, que albergará una pseudo fila del tamaño de la imagen destino, cuyos datos tienen un tamaño de 32 bits. Su valor se obtiene como el exponente de la potencia de dos superior al ancho o número de columnas de la imagen destino.
SRC_ADDRESS_WIDTH	Determina el tamaño de la memoria empleada en TFG_IMAGE_SCALING para albergar una fila de la imagen fuente, denominada **src_ram**. Su valor se obtiene como el exponente de la potencia de dos igual o inmediatamente superior al ancho o número de columnas de la imagen fuente dividido por cuatro (valor que se podrá ver incrementado, como se verá con más detalle en el apartado 5.7.4), debido a que en este caso, **src_ram** albergará valores de 32 bits, lo que equivale a 4 bytes o refiriéndonos a imágenes, 4 pixeles, ya que los accesos a la memoria que alberga la imagen fuente son de 32 bits.
DST_ADDRESS_WIDTH	Determina el tamaño de la memoria empleada en TFG_IMAGE_SCALING para albergar una fila de la imagen destino, denominada **dst_ram**. Su valor se obtiene como el exponente de la potencia de dos igual o inmediatamente superior al alto o número de filas la imagen destino dividido por cuatro (valor que se podrá ver incrementado, como se verá con más detalle en el apartado 5.7.8), debido a que en este caso, **dst_ram** albergará valores de 32 bits, lo que equivale a 4 bytes o refiriéndonos a imágenes, 4 pixeles, ya que los accesos a la memoria donde se colocará la imagen destino son de 32 bits.
ADDRESS_HEIGHT	Determina el tamaño del contador que controla el algoritmo de escalado, denominado **row_cnt**. Su valor se obtiene como el exponente de la potencia de dos igual o inmediatamente superior al alto o número de filas de la imagen destino.

Tabla 5.3 – Genéricos de TFG_IMAGE_SCALING

Una vez descritos los genéricos, pasamos a la parte de los puertos, con la cual nuestro IP se comunica con el procesador y la memoria externa y demás componentes implementados en este.

Señal	Descripción
Rst_n	Línea de reinicio de la CPU. Se encuentra directamente cableado a un pin de la FPGA mediante el pulsador denominado CPU reset.
Clk	Línea de reloj de la CPU. En nuestro caso está programado para una frecuencia de 60 MHz.
Apbi	Descripción en apartado 5.6.4.
Apbo	Descripción en apartado 5.6.4.
Ahbi	Descripción en apartado 5.6.3.
Ahbo	Descripción en apartado 5.6.3.

Tabla 5.4 – Señales de TFG_IMAGE_SCALING

5.6.3 TFG_AHB_MASTER

Este módulo, perteneciente a la librería S.H.O.R.E.S [48], contiene la lógica que permite realizar los accesos al AHB bus para comunicarse con la memoria externa. Se han efectuado algunas mejoras y correcciones en su código.

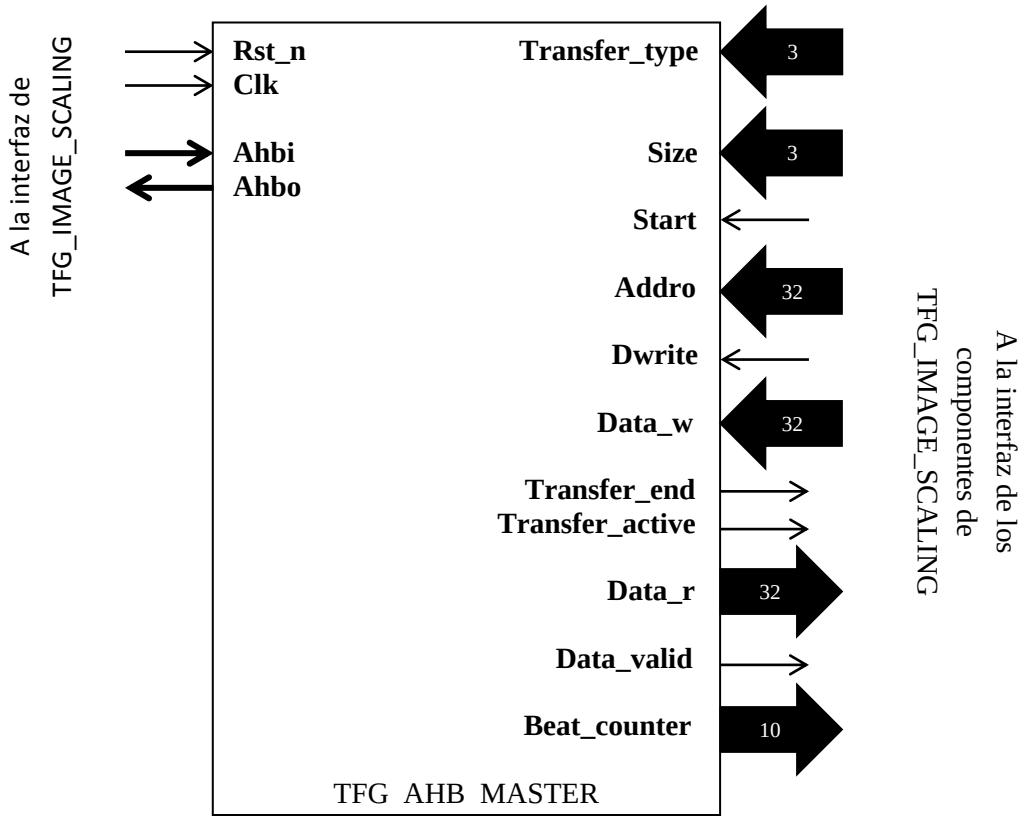


Imagen 5.28 – Interfaz de TFG_AHB_MASTER

A continuación, pasamos a describir la interfaz de esta entidad. Comenzamos para ello con los genéricos.

Genéricos	Descripción
HINDEX	Índice de selección del AHB master.
VENID	Información referente al plug&play, identificador del proveedor.
DEVID	Información referente al plug&play, identificador del dispositivo.
VERSION	Información referente al plug&play, versión del dispositivo.

Tabla 5.5 – Genéricos de TFG_AHB_MASTER

Una vez descritos los genéricos, pasamos a la parte de los puertos.

Señal	Descripción
Rst_n	Descripción en apartado 5.6.2
Clk	Descripción en apartado 5.6.2

Ahbi	Bus compuesto mediante el cual el master accede al AHB bus para realizar una lectura de los datos presentes en este. Las señales que lo componen están presentes en la definición ahb_mst_in_type y son las siguientes:
	
	`type ahb_mst_in_type is record`
	`  hgrant : std_logic_vector(0 to NAHBMST-1);`
	`  hready : std_ulogic;`
	`  hresp : std_logic_vector(1 downto 0);`
	`  hrdata : std_logic_vector(AHBDW-1 downto 0);`
	`  hirq : std_logic_vector(NAHBIRQ-1 downto 0);`
	`  testen : std_ulogic;`
	`  testrst : std_ulogic;`
	`  scanen : std_ulogic;`
	`  testoen : std_ulogic;`
	`  testin : std_logic_vector(NTESTINBITS-1 downto 0);`
	`end record;`

Ahbo	Bus compuesto mediante el cual el master accede al AHB bus para realizar una escritura. Las señales que lo componen están presentes en la definición ahb_mst_out_type y son las siguientes:
	
	`type ahb_mst_out_type is record`
	`  hbusreq : std_ulogic;`
	`  hlock : std_ulogic;`
	`  htrans : std_logic_vector(1 downto 0);`
	`  haddr : std_logic_vector(31 downto 0);`
	`  hwrite : std_ulogic;`
	`  hsize : std_logic_vector(2 downto 0);`
	`  hburst : std_logic_vector(2 downto 0);`
	`  hprot : std_logic_vector(3 downto 0);`
	`  hwdata : std_logic_vector(AHBDW-1 downto 0);`
	`  hirq : std_logic_vector(NAHBIRQ-1 downto 0);`
	`  hconfig : ahb_config_type;`
	`  hindex : integer range 0 to NAHBMST-1;`
	`end record;`

Transfer_type	Bus de 3 bits que permite configurar el tipo de transferencia que el master realizará con el AHB bus en cuanto al número de datos que serán transmitidos (**Beat_counter**). Su valor se carga en **Ahbo.hburst**.
Size	Bus de 3 bits que determina el tamaño de los datos que serán transmitidos (byte, halfword o word). Su valor se carga en **Ahbo.hsize**.
Start	Línea que determina el inicio de una transferencia. Su valor se carga en **Ahbo.hbusreq** y **Ahbo.hlock** y se mantiene en nivel alto mientras dura la transferencia.
Addro	Bus de 32 bits que contiene la dirección de lectura/escritura de datos entre la memoria y los elementos conectados al AHB bus. Su valor se carga en **Ahbo.haddr**.
Dwrite	Línea de lectura/escritura. Si se encuentra en nivel alto se está realizando una escritura en el AHB bus. Su valor se carga en **Ahbo.hwrite**.

Data_w	Bus de 32 bits que contiene el valor que se pretender enviar por el AHB bus. Su valor se carga en **Ahbo.hwdata**.
Transfer_end	Línea que determina el fin de una transferencia. Su valor se genera internamente en TFG_AHB_MASTER y dura un solo ciclo de reloj.
Transfer_active	Línea que confirma que la transferencia esta activa. Se mantiene en nivel alto tras activar **Start** y detectarse **Ahbi.hready** en nivel bajo hasta que finaliza la transferencia con **Transfer_end**.
Datar	Bus de 32 bits que contiene el valor que se lee de AHB bus. Toma su valor de **Ahbi.hrdata (31 downto 0)**.
Data_valid	Línea que confirma que el dato presente en el AHB bus es válido. Estará en nivel alto siempre que **Transfer_active** y **Ahbi.hready** estén en nivel alto.
Beat_counter	Bus de 10 bits que cuenta el número de datos enviados/recibidos por el AHB bus. Su valor se incrementa en uno en cada ciclo de reloj de **Clk** mientras **Data_valid** se encuentre en nivel alto.

Tabla 5.6 – Señales de TFG_AHB_MASTER

5.6.4 TFG_APB_SLAVE

Este módulo, perteneciente a la librería S.H.O.R.E.S [48], contiene la lógica que permite realizar los accesos al APB bus para configurar el módulo IP desde el software que funciona en el procesador. Se han efectuado algunas mejoras y correcciones en su código.

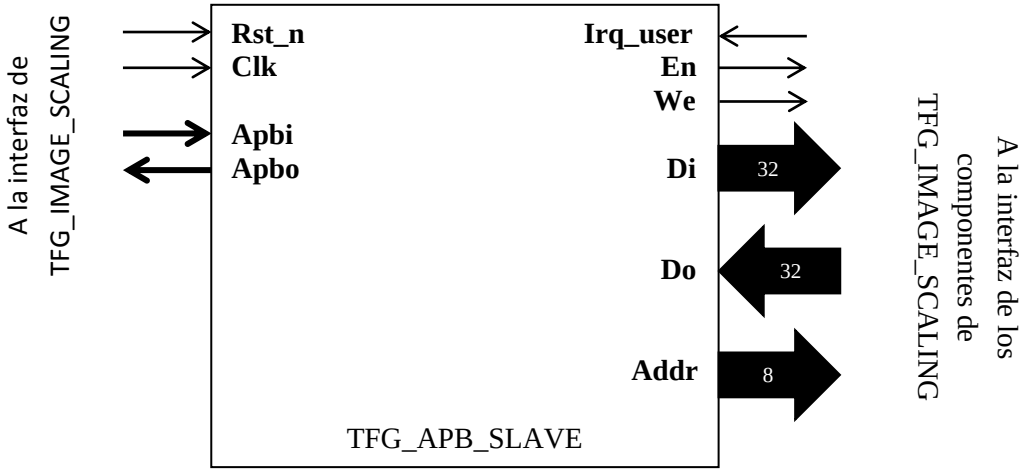


Imagen 5.29 – Interfaz de TFG_APB_SLAVE

A continuación, pasamos a describir la interfaz de esta entidad. Comenzamos para ello con los genéricos.

Genéricos	Descripción
NAHBIRQ	Número máximo de interrupciones.
PINDEX	Índice de selección del APB slave.
PADDR	Bus de direcciones del APB bus.
PMASK	Máscara empleada para obtener la dirección **Addr**.
HIRQ	Numero de línea de interrupción de TFG_APB_SLAVE.

Tabla 5.7 – Genéricos de TFG_APB_SLAVE

Una vez descritos los genéricos, pasamos a la parte de los puertos.

Señal	Descripción
Rst_n	Descripción en apartado 5.6.2.
Clk	Descripción en apartado 5.6.2.

Apbi	Bus compuesto mediante el cual el esclavo accede al APB bus para realizar una lectura de los datos presentes en este. Las señales que lo componen están presentes en la definición apb_slv_in_type y son las siguientes:
	
	`type apb_slv_in_type is record`
	`  psel : std_logic_vector(0 to NAPBSLV-1);`
	`  penable : std_ulogic;`
	`  paddr : std_logic_vector(31 downto 0);`
	`  pwrite : std_ulogic;`
	`  pwdata : std_logic_vector(31 downto 0);`
	`  pirq : std_logic_vector(NAHBIRQ-1 downto 0);`
	`  testen : std_ulogic;`
	`  testrst : std_ulogic;`
	`  scanen : std_ulogic;`
	`  testoen : std_ulogic;`
	`  testin : std_logic_vector(NTESTINBITS-1 downto 0);`
	`end record;`

Apbo	Bus compuesto mediante el cual el esclavo accede al APB bus para realizar una escritura. Las señales que lo componen están presentes en la definición ahb_mst_out_type y son las siguientes:
	
	`type apb_slv_out_type is record`
	`  prdata : std_logic_vector(31 downto 0);`
	`  pirq : std_logic_vector(NAHBIRQ-1 downto 0);`
	`  pconfig : apb_config_type;`
	`  pindex : integer range 0 to NAPBSLV -1;`
	`end record;`

Irq_user	Línea de interrupción de TFG_APB_SLAVE.
En	Línea de esclavo seleccionado. Si se encuentra en nivel alto, el esclavo en cuestión tendrá acceso al APB bus. Toma su valor de **Apbi.psel** cuando esta apunta al índice de nuestro esclavo.
We	Línea de lectura/escritura. Si se encuentra en nivel alto, significa que hay un dato en el APB bus disponible para el esclavo seleccionado. Esto se da cuando **Apbi.psel** apunta a nuestro esclavo y **Apbi.pwrite** y **Apbi.penable** se encuentran en nivel alto.
Di	Bus de 32 bits que contiene el dato que se ha leído del APB bus. Toma su valor de **Apbi.pwdata**.
Do	Bus de 32 bits que contiene el dato que se pretende mandar por el APB bus. Toma su valor de uno de los registros presentes en TFG_IMAGE_SCALING y lo carga en **Apbo.prdata**.
Addr	Bus de 8 bits que contiene la dirección del registro de TFG_IMAGE_SCALING en el cual se va a escribir el dato **Di** o del que se va a leer un dato con **Do**. Toma su valor de **Apbi.paddr** de una forma específica. **Apbi.paddr** es un bus de 32 bits, y la asignación de **Addr** se hace con los bits 9 downto 2, de forma que se tienen accesos de tamaño word. Nuestro IP tiene asignada la dirección 0x80000A00, y el anterior y posterior módulos IP implementados están mapeados en 0x80000900 y 0x80000B00 correspondientemente. Así pues, el tamaño de memoria asignado a nuestro IP es de 256 bytes (los ocho bits 9 downto 2). El hecho por el cual realizamos accesos de tamaño word se debe a que tratamos nuestra memoria como registros de 32 bits, con lo cual, los dos bits menos significativos de **Apbi.paddr** no son útiles puesto que estamos contando de 4 en 4 bytes para poder acceder de un registro al siguiente.

Tabla 5.8 – Señales de TFG_APB_SLAVE

5.6.5 Banco de registros

Conjunto de registros de 32 bits presentes en el módulo TFG_IMAGE_SCALLING, configurables desde el software, que provee de los siguientes datos e información al algoritmo y los componentes presentes en nuestro módulo IP.

Señal	Descripción
status_REG	Registro de solo lectura que nos permite saber en qué estado de funcionamiento se encuentra el algoritmo (funcionando si el bit cero del registro está en nivel alto).
control_REG	Registro de lectura/escritura que nos permite poner en funcionamiento el algoritmo de interpolación. Si el bit cero del registro se encuentra en nivel alto, dará comienzo un nuevo ciclo de escalado. Para iniciar otro ciclo de escalado será necesario colocar antes este bit a nivel bajo y después otra vez en nivel alto.
address_src_REG	Registro de lectura/escritura que contiene la dirección de la imagen fuente.
address_dst_REG	Registro de lectura/escritura que contiene la dirección de la imagen destino.
src_width_REG	Registro de lectura/escritura que contiene el ancho o número de columnas de la imagen fuente.
src_height_REG	Registro de lectura/escritura que contiene el alto o número de filas de la imagen fuente.
dst_width_REG	Registro de lectura/escritura que contiene el ancho o número de columnas de la imagen destino.
dst_height_REG	Registro de lectura/escritura que contiene el alto o número de filas de la imagen destino.
shifted_scale_alpha_REG	Registro de lectura/escritura que contiene el resultado de la expresión [4.14].
shifted_scale_beta_REG	Registro de lectura/escritura que contiene el resultado de la expresión [4.15].

Tabla 5.9 – Banco de registros

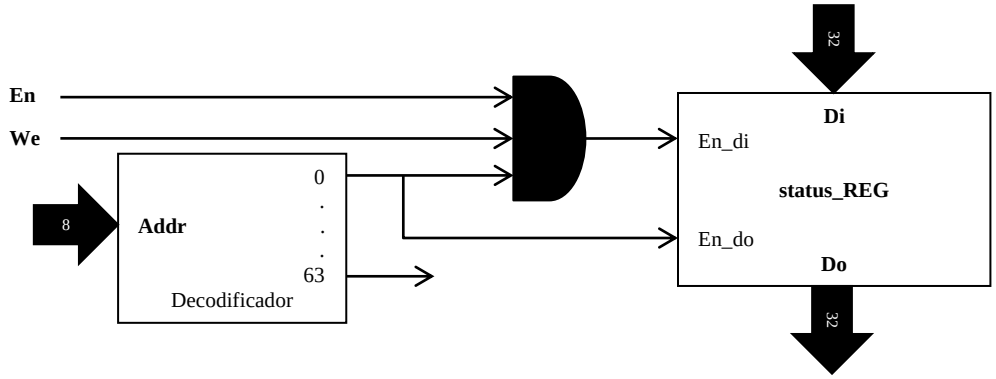


Imagen 5.30 – Interfaz y control de registros

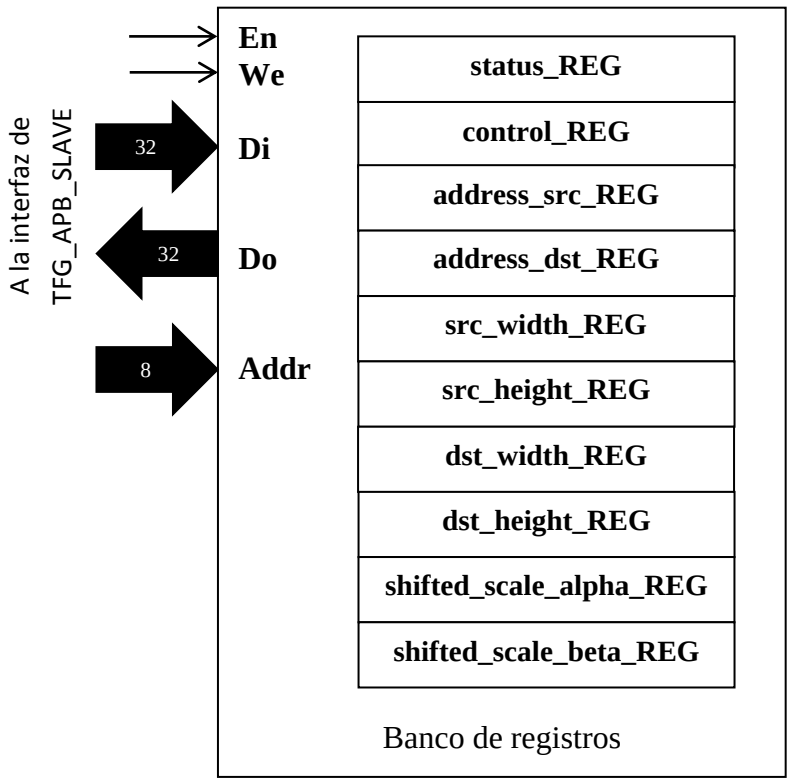


Imagen 5.31 – Interfaz del banco de registros

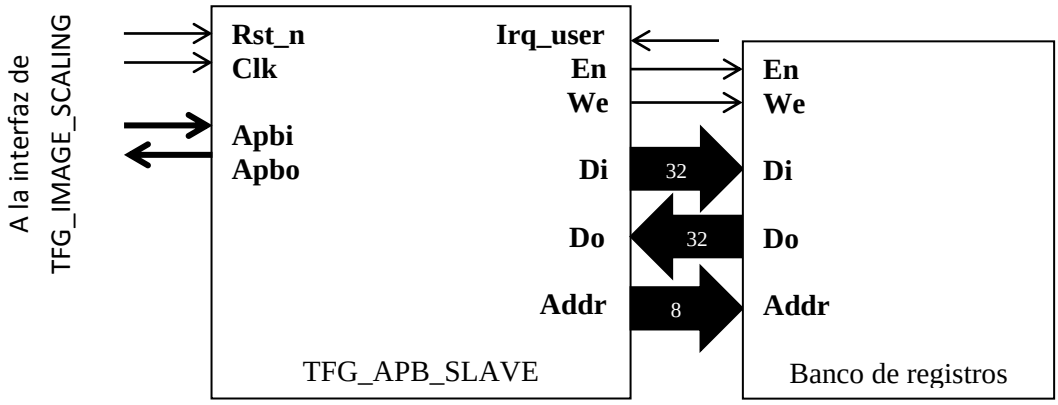


Imagen 5.32 – Conexión de TFG_APB_SLAVE con el banco de registros

5.6.6 TFG_ROW_INTERPOLATION

Este componente implementa una máquina de estados sincronizada con la presente en TFG_IMAGE_SCALING. Es el corazón del diseño, donde se centra el algoritmo de interpolación. Contiene a su vez otros cuatro componentes más, tres de ellos se encargan de realizar los cálculos oportunos para obtener los pixeles de la fila que se esté procesando y el cuarto es una memoria que guarda datos intermedios que se obtienen durante los cálculos.

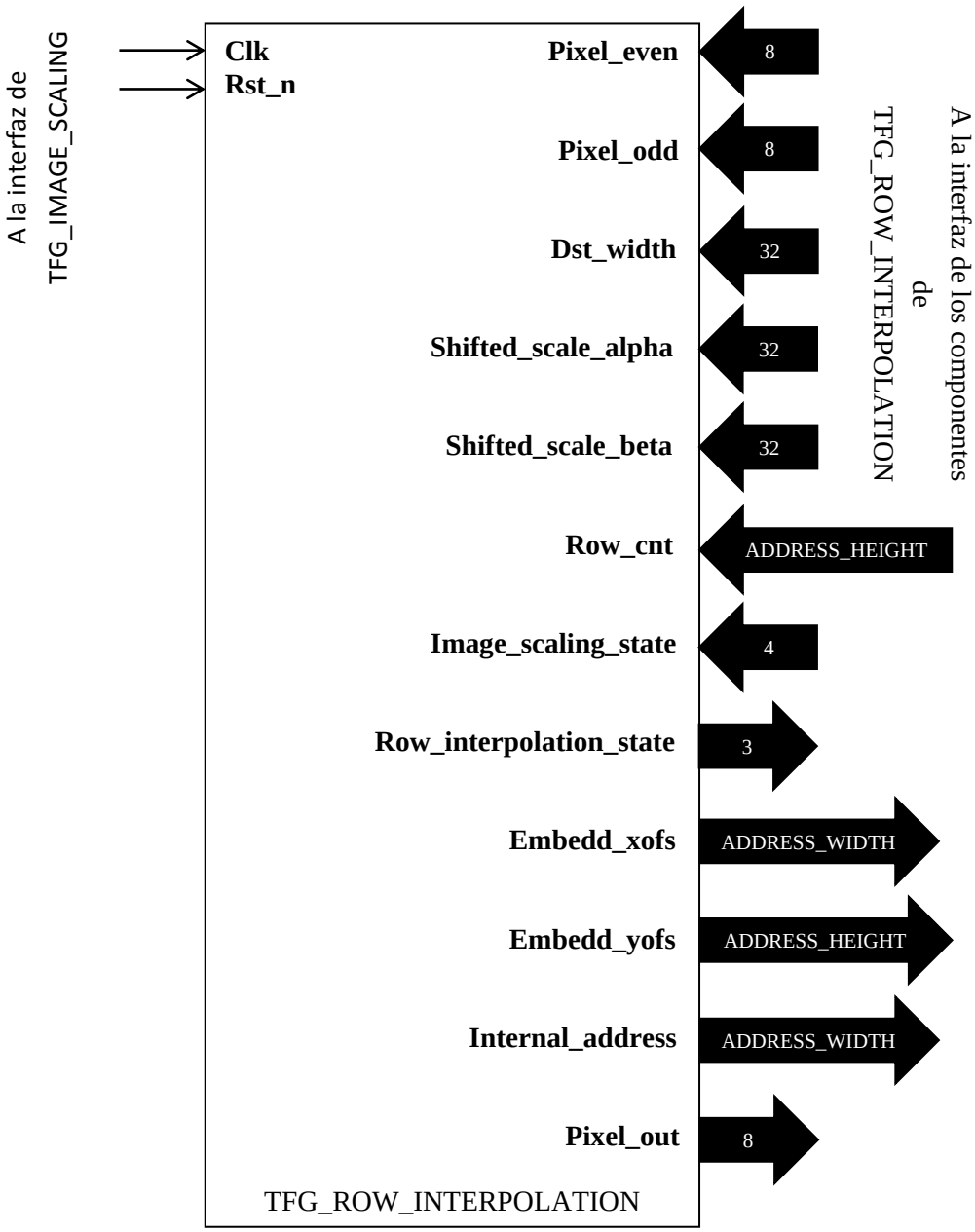


Imagen 5.33 – Interfaz de TFG_ROW_INTERPOLATION

A continuación, pasamos a describir la interfaz de esta entidad. Comenzamos para ello con los genéricos.

Genérico	Descripción
ADDRESS_WIDTH	Toma su valor de **INT_ADDRESS_WIDTH**, descrito en el apartado 5.6.2.
ADDRESS_HEIGHT	Toma su valor de **ADDRESS_HEIGHT**, descrito en el apartado 5.6.2.
IMSE_EXTRA_PRECISION_BITS	Descripción en apartado 5.6.7.
INTER_RESIZE_COEF_BITS	Descripción en apartado 5.6.7.
SHIFT	Descripción en apartado 5.6.9.

Tabla 5.10 – Genéricos de TFG_ROW_INTERPOLATION

Una vez descritos los genéricos, pasamos a la parte de los puertos.

Señal	Descripción
Rst_n	Descripción en apartado 5.6.2.
Clk	Descripción en apartado 5.6.2.
Pixel_even	Bus de 8 bits que contiene el valor del pixel seleccionado por el índice de columnas **Embedd_xofs**.
Pixel_odd	Bus de 8 bits que contiene el valor del pixel siguiente al seleccionado por el índice de columnas **Embedd_xofs**.
Dst_width	Bus de 32 bits que contiene el ancho o número de columnas de la imagen destino. Toma su valor del registro **dst_width_REG**.
Shifted_scale_alpha	Bus de 32 bits que toma su valor del registro **shifted_scale_alpha_REG**.
Shifted_scale_beta	Bus de 32 bits que toma su valor del registro **shifted_scale_beta_REG**.
Row_cnt	Bus de **ADDRESS_HEIGHT** bits que indica la fila de la imagen destino que está siendo calculada. Su valor se genera en TFG_IMAGE_SCALING.
Image_scaling_state	Bus de 4 bits que indica el valor del estado de la máquina de estados presente en TFG_IMAGE_SCALING.
Row_interpolation_state	Bus de 3 bits que indica el valor del estado de la máquina de estados presente en TFG_ROW_INTERPOLATION.
Embedd_xofs	Bus de **ADDRESS_WIDTH** bits que contiene la posición del pixel a emplear en las expresiones de interpolación.
Embedd_yofs	Bus de **ADDRESS_ HEIGHT** bits que indica las filas de las que se obtendrán los pixeles con **Embedd_xofs**.
Internal_address	Bus de **ADDRESS_ HEIGHT** bits que indica la posición del pixel de una fila de la imagen destino que está siendo calculado.
Pixel_out	Bus de 8 bits que contiene el valor del pixel de una fila de la imagen destino en la posición **Internal_address**.

Tabla 5.11 – Señales de TFG_ROW_INTERPOLATION

5.6.7 TFG_RESIZE_EMBB

Este módulo se encarga de obtener los índices de selección de las filas/columnas y los coeficientes que intervienen en los cálculos de interpolación.

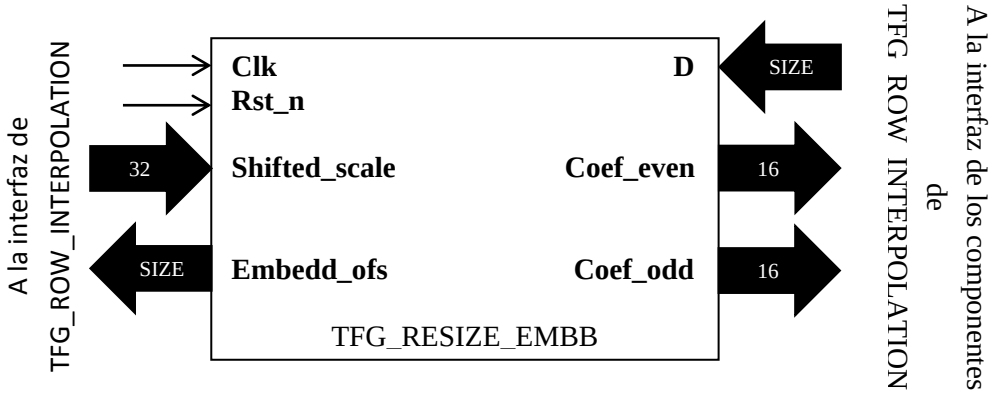


Imagen 5.34 – Interfaz de TFG_RESIZE_EMBB

A continuación, pasamos a describir la interfaz de esta entidad. Comenzamos para ello con los genéricos.

Genérico	Descripción
SIZE	Determina el ancho de los buses **D** y **Embedd_ofs**. En la implementación realizada en el diseño de nuestro IP se han realizado dos instanciaciones de este componente, donde este genérico toma los valores **ADDRESS_WIDTH** y **ADDRESS_HEIGHT**.
IMSE_EXTRA_PRECISION_BITS	Este genérico, junto con el siguiente, son valores tomados directamente desde el algoritmo software de interpolación bilineal. En concreto, **INTER_RESIZE_COEF_BITS** (igual a 11) e **IMSE_EXTRA_PRECISION_BITS** (igual a 4) conforman el valor del exponente de la potencia de dos con el cual se obtiene el factor de escala **IMSE_RESIZE_COEF_SCALE**.
INTER_RESIZE_COEF_BITS	Descrito anteriormente.

Tabla 5.12 – Genéricos de TFG_RESIZE_EMBB

Una vez descritos los genéricos, pasamos a la parte de los puertos.

Señal	Descripción
Rst_n	Descripción en apartado 5.6.2.
Clk	Descripción en apartado 5.6.2.
D	Bus de **SIZE** bits conectado a un contador ascendente, que junto a **Shifted_scale**, se encarga de generar el índice de selección **Embedd_ofs** y los coeficientes **Coef_even** y **Coef_odd**.
Shifted_scale	Bus de 32 bits que contiene el resultado de la expresión [4.14]. Según la instanciación a la que nos refiramos, toma su valor de **Shifted_scale_alpha** o **Shifted_scale_beta**.
Embedd_ofs	Bus de **SIZE** bits que contiene la posición de las columnas/filas que intervienen en los cálculos de interpolación, obtenido según la expresión que corresponda, [4.19] o [4.26].
Coef_even	Bus de 16 bits que contiene el coeficiente de interpolación que se aplica a la columna/fila indicada por **Embedd_ofs**, obtenido con las expresiones [4.21] a [4.23] o [4.27] a [4.29].
Coef_odd	Bus de 16 bits que contiene el coeficiente de interpolación que se aplica a la columna/fila siguiente a la indicada por **Embedd_ofs**, obtenido según las expresiones [4.21] a [4.23] o [4.27] a [4.29].

Tabla 5.13 – Señales de TFG_RESIZE_EMBB

5.6.8 TFG_HRESIZE_KERNEL

Este módulo se encarga de obtener los datos de las pseudo filas, que después pasan a TFG_VRESIZE_KERNEL para obtener la fila de la imagen destino.

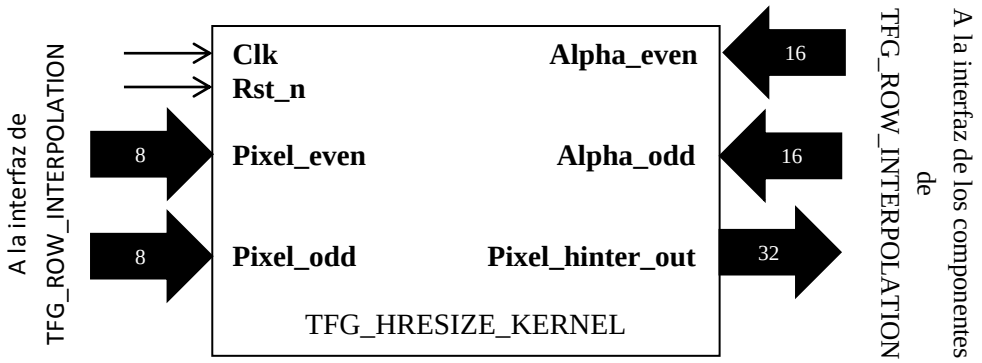


Imagen 5.35 – Interfaz de TFG_HRESIZE_KERNEL

A continuación, pasamos a describir la interfaz de esta entidad. Como no contiene genéricos, pasamos directamente a la descripción de sus puertos.

Señal	Descripción
Rst_n	Descripción en apartado 5.6.2.
Clk	Descripción en apartado 5.6.2.
Pixel_even	Bus de 8 bits que contiene el valor del pixel seleccionado por el índice de columnas **Embedd_xofs**.
Pixel_odd	Bus de 8 bits que contiene el valor del pixel siguiente al seleccionado por el índice de columnas **Embedd_xofs**.
Alpha_even	Bus de 16 bits que contiene el coeficiente de interpolación **Coef_even** obtenido con TFG_RESIZE_EMBB que se aplicará a **Pixel_even**.
Alpha_odd	Bus de 16 bits que contiene el coeficiente de interpolación **Coef_odd** obtenido con TFG_RESIZE_EMBB que se aplicará a **Pixel_odd**.
Pixel_hinter_out	Bus de 32 bits que contiene el resultado de la expresión [4.30].

Tabla 5.14 – Señales de TFG_HRESIZE_KERNEL

5.6.9 TFG_VRESIZE_KERNEL

Este módulo se encarga de obtener los datos de cada una de las filas de la imagen destino.

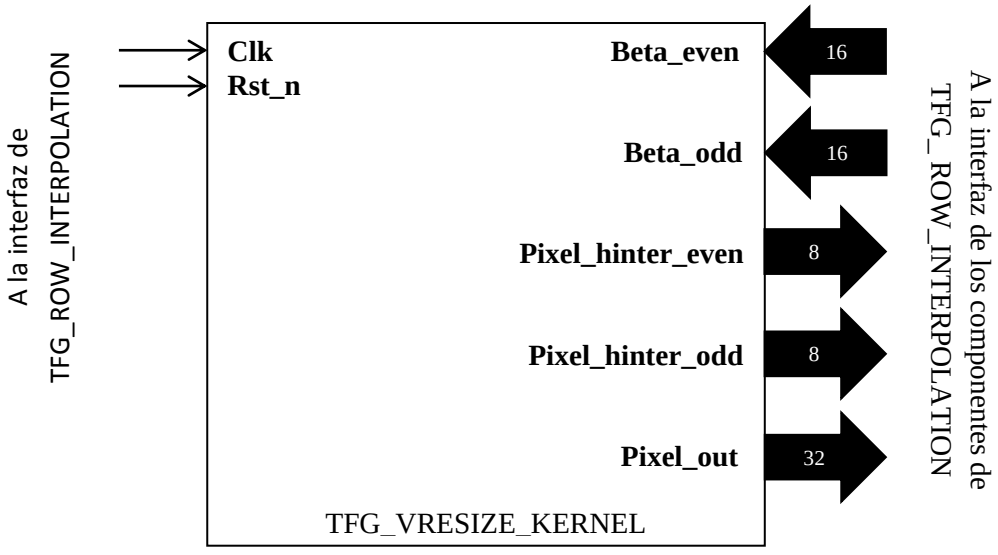


Imagen 5.36 – Interfaz de TFG_VRESIZE_KERNEL

A continuación, pasamos a describir la interfaz de esta entidad. Comenzamos para ello con los genéricos.

Genérico	Descripción
SHIFT	Este genérico es un valor tomado directamente desde el algoritmo de interpolación en software. Es el doble de **INTER_RESIZE_COEF_BITS** y se usa como factor de escala inversa para obtener el valor real de los pixeles que se han calculado con el factor de escala **INTER_RESIZE_COEF_SCALE**.

Tabla 5.15 – Genéricos de TFG_VRESIZE_KERNEL

Una vez descritos los genéricos, pasamos a la parte de los puertos.

Señal	Descripción
Rst_n	Descripción en apartado 5.6.2.
Clk	Descripción en apartado 5.6.2
Pixel_hinter_even	Bus de 32 bits que contiene el valor del dato "n" de la primera pseudo fila obtenida mediante TFG_HRESIZE_KERNEL.
Pixel_hinter_odd	Bus de 32 bits que contiene el valor del dato "n" de la segunda pseudo fila obtenida mediante TFG_HRESIZE_KERNEL.
Beta_even	Bus de 16 bits que contiene el coeficiente de interpolación **Coef_even** obtenido con TFG_RESIZE_EMBB que se aplica a **Pixel_hinter_even**.
Beta_odd	Bus de 16 bits que contiene el coeficiente de interpolación **Coef_odd** obtenido con TFG_RESIZE_EMBB que se aplica a **Pixel_hinter_odd**.
Pixel_out	Bus de 8 bits que contiene el valor de un pixel de la imagen destino, resultado de la expresión [4.31].

Tabla 5.16 – Señales de TFG_VRESIZE_KERNEL

5.6.10 TFG_ROW_BUFFER

Este módulo constituye un bloque de memoria que alberga una de las pseudo filas a partir de las cuales se obtiene una fila de la imagen destino. El tamaño de dicha memoria es pues del ancho o número de columnas de la imagen destino.

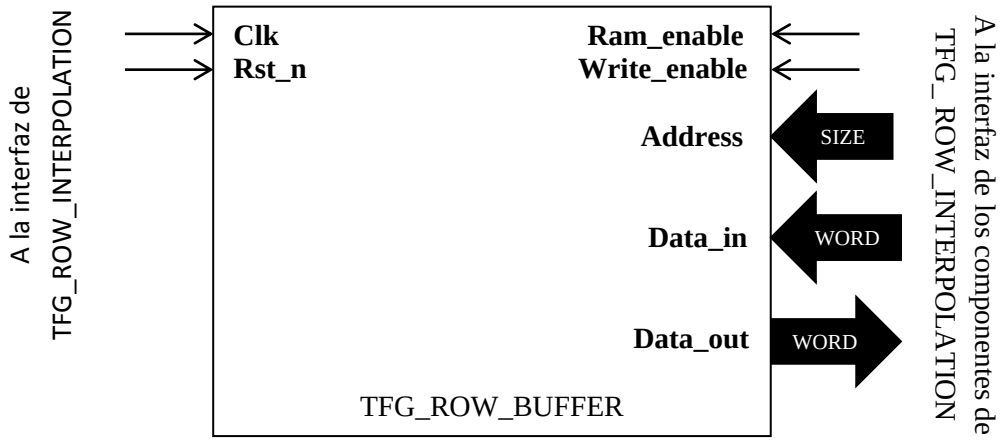


Imagen 5.37 – Interfaz de TFG_ROW_BUFFER

A continuación, pasamos a describir la interfaz de esta entidad. Comenzamos para ello con los genéricos.

Genérico	Descripción
SIZE	Determina el tamaño que tiene la memoria. Toma su valor de **ADDRESS_WIDTH**.
WORD	Determina el tamaño de los datos que se guardan en la memoria. Para nuestro diseño se requiere guardar los datos **Pixel_hinter_out** procedentes de TFG_HRESIZE_KERNEL de tamaño word, luego su valor es 32.

Tabla 5.17 – Genéricos de TFG_ROW_BUFFER

Una vez descritos los genéricos, pasamos a la parte de los puertos.

Señal	Descripción
Rst_n	Descripción en apartado 5.6.2.
Clk	Descripción en apartado 5.6.2.
Ram_enable	Línea de activación de la memoria. Cuando está en nivel alto se puede realizar una lectura del dato guardado en la memoria en la dirección **Address**.
Write_enable	Línea de escritura activa. Cuando está en nivel alto (junto con **Ram_enable**) se puede realizar una escritura en la memoria sobre la dirección **Address**.
Address	Bus de **SIZE** bits que contiene la dirección de lectura/escritura de datos en la memoria.
Data_in	Bus de **WORD** bits que contiene el dato que se pretende almacenar en la memoria en la dirección **Address**.
Data_out	Bus de **WORD** bits que contiene el dato que se obtenga al realizar una lectura sobre la memoria en la dirección **Address**.

Tabla 5.18 – Señales de TFG_ROW_BUFFER

5.7 Descripción de comportamiento

Una vez estudiados los distintos módulos más importantes que componen nuestro IP, pasamos a describir la funcionalidad de este desde otro punto de vista, casi a nivel de flujo de datos. Para ello, nos centramos en las máquinas de estado implementadas en TFG_IMAGE_SCALING y TFG_ROW_INTERPOLATION, cuyas funciones principales son los accesos a la memoria externa y la obtención de una fila de la imagen destino.

Como hemos introducido, TFG_IMAGE_SCALING se encarga principalmente de realizar los accesos a la memoria externa para leer y escribir los datos de las imágenes fuente y destino. Pero también es quien controla el comienzo y finalización del algoritmo de escalado. Para ello se vale de una máquina de estados implementada como una máquina de Moore compuesta por 16 estados numerados de $\mathbf{S_0}$ a $\mathbf{S_{15}}$ como se puede apreciar en la siguiente imagen:

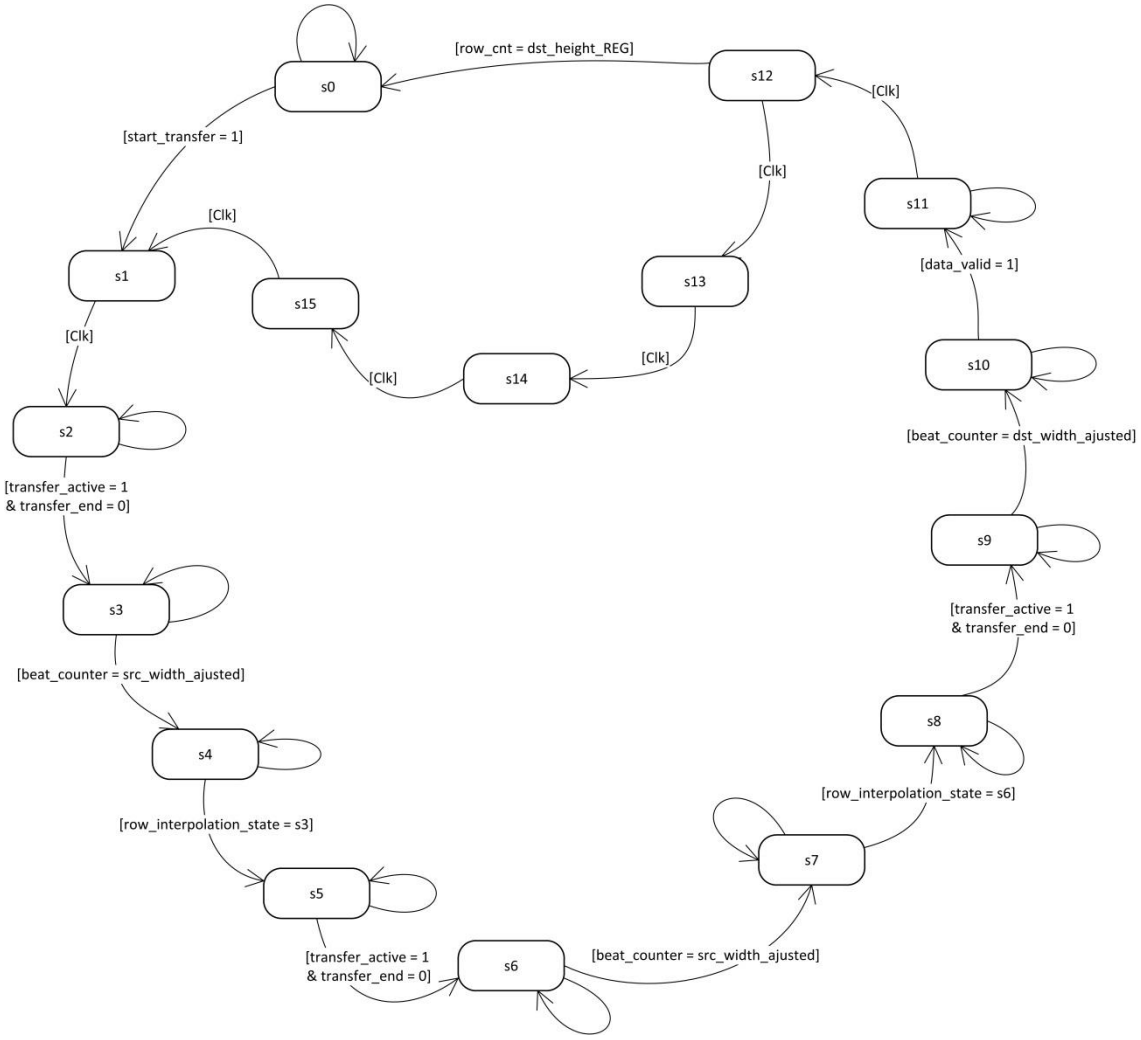


Imagen 5.38 – Maquina de estados de TFG_IMAGE_SCALING

Dado que esta máquina de estados se encuentra sincronizada con la presente en TFG_ROW_INTERPOLATION (condiciones de transición **row_interpolation_state** e **image_scaling_state**) hacemos también una pequeña introducción a esta para a continuación, pasar a describir el algoritmo por completo.

Al igual que ocurre en TFG_IMAGE_SCALING, en TFG_ROW_INTREPOLATION se ha implementado una segunda máquina de estados también descrita como máquina de Moore, cuya función principal es la obtención de una fila de la imagen destino (podríamos decir que es el kernel o núcleo del algoritmo de interpolación). En consecuencia, es la encargada de calcular los índices de selección de columnas/filas que serven a la primera máquina de estados para proporcionar los datos necesarios a TFG_ROW_INTERPOLATION y los coeficientes de interpolación para obtener los datos de las filas de la imagen destino. Dicha maquina está compuesta por 8 estados numerados de $\mathbf{S_0}$ a $\mathbf{S_7}$ como se puede apreciar en la siguiente imagen:

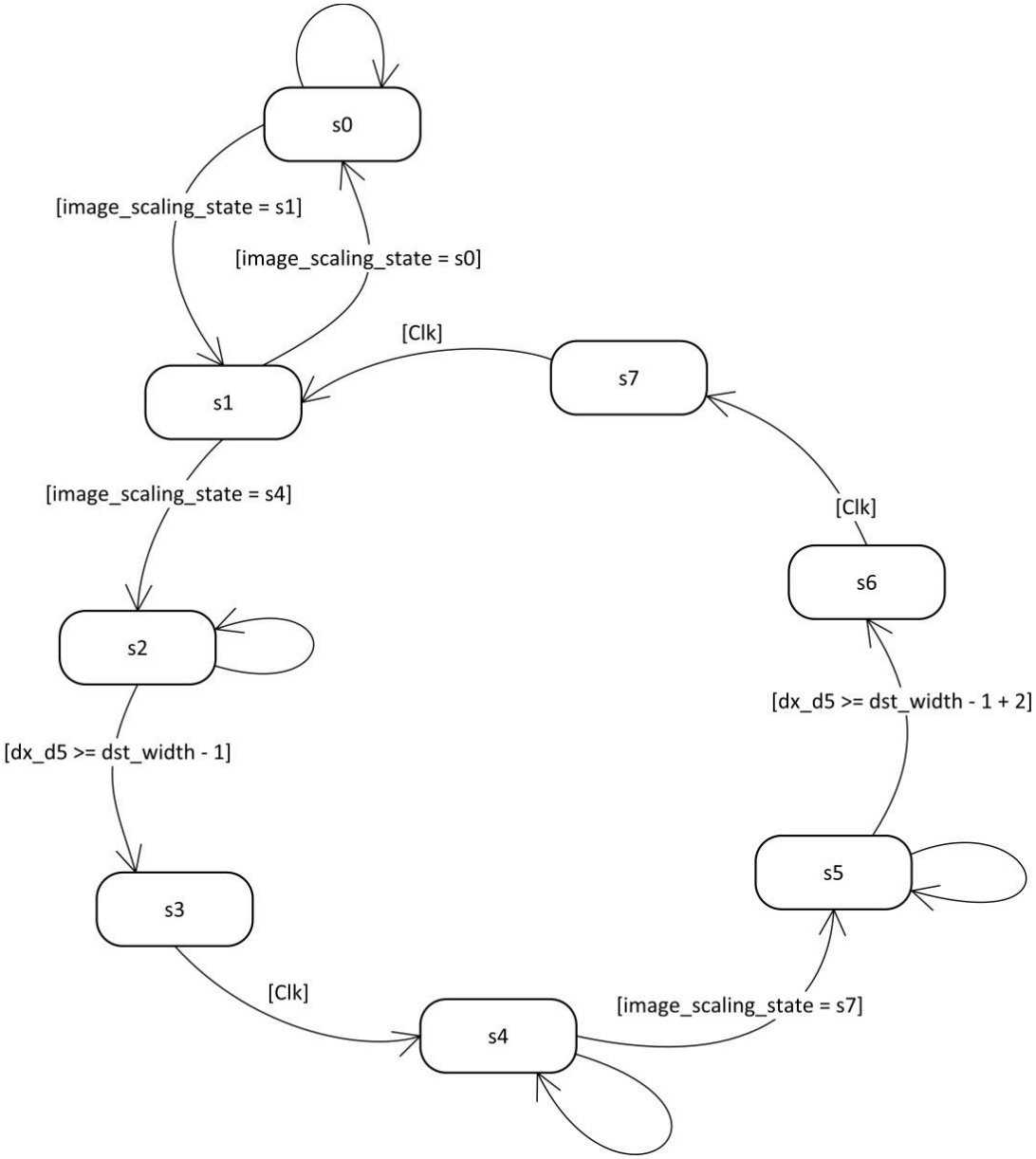


Imagen 5.39 – Maquina de estados de TFG_ROW_INTERPOLATION

Antes de pasar a describir cada uno de los estados de ambas máquinas, nos detenemos a conocer la descripción de estas:

```
-- TFG_IMAGE_SCALING state machine

type state_TYPE is (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14,
s15);

signal current_state : state_TYPE := s0;

-- TFG_ROW_INTERPOLATION state machine

type state_TYPE is (s0, s1, s2, s3, s4, s5, s6, s7);

signal current_state : state_TYPE := s0;
```

Como se puede apreciar en las definiciones, ambas máquinas de estado comienzan en $\mathbf{S_0}$ (**current_state** inicializado con el estado $\mathbf{S_0}$).

Ambas máquinas tienen implementado un reset asíncrono, de forma que si se detecta un nivel bajo en **Rst_n**, retornan al estado $\mathbf{S_0}$. Veamos su descripción:

```
state_PROCESS : process (Clk, Rst_n)

begin

        if (Rst_n = '0') then

                current_state <= s0;

        elsif (Clk = '1' and Clk'event) then

                case current_state is

                        when s0 => ...

                                .
                                .
                                .

                end case;

        end if;

end process;
```

Esto, junto con la definición vista de las máquinas de estados, nos asegura que cada vez que arrancamos el sistema desde cero o se realice un reset manual, ambas se encuentran en el estado $\mathbf{S_0}$ a la espera de sus condiciones de transición.

Antes de continuar vemos el proceso a seguir para realizar las simulaciones que se muestran en las imágenes presentes en los siguientes apartados.

Lo primero consiste en describir el código C que funcionará en el procesador (test_IP.c, ver Anexo A.2) a partir del cual se obtiene un archivo (test_IP) que es el que se carga directamente en la FPGA.

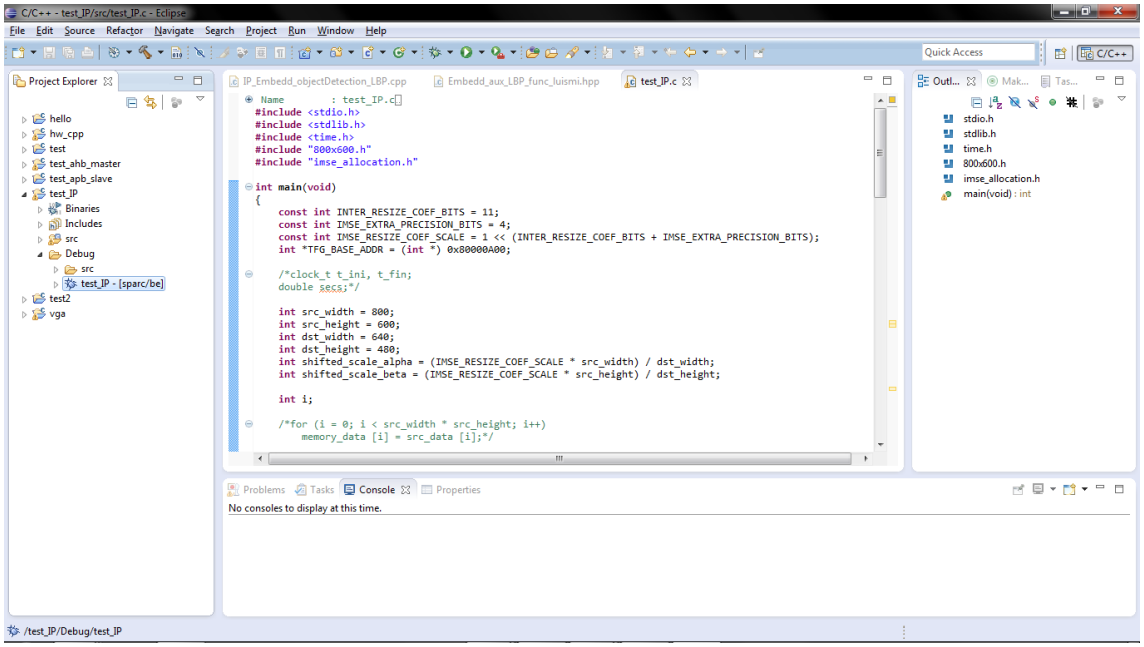


Imagen 5.40 – Código C y archivo test_IP

Dado que este archivo no es compatible con el simulador, se ha de realizar una conversión mediante el comando *sparc-elf-objcopy -O srec test_IP tfg_ram.srec* en el terminal de MinGW. Este archivo (tfg_ram.srec) se guarda en el directorio del proyecto. Tras esto tenemos que preparar el código VHDL para arrancar el simulador. Para ello empleamos los comandos *make vsim*, que compila el código y *vsim testbench*, que arranca el simulador. Una vez dentro de ModelSim, seleccionamos las señales y tecleamos el comando *run -all*. La simulación se detiene mediante el icono de *stop*.

En la siguiente imagen podemos ver como al comienzo de la simulación, los estados iniciales de ambas maquinas son **S_0** (el **current_state** en verde corresponde a la máquina de TFG_IMAGE_SCALING y el azul a la de TFG_ROW_INTERPOLATION).

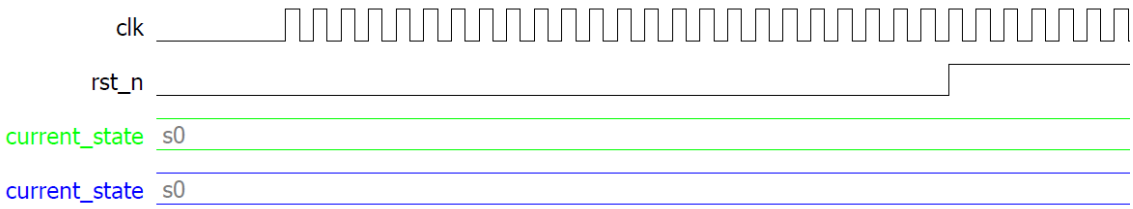


Imagen 5.41 – Inicio de simulación

De aquí en adelante (a fin de facilitar la lectura) nos referimos a la máquina de estados implementada en TFG_IMAGE_SCALING como **image_machine** y a la implementada en TFG_ROW_INTERPOLATION como **row_machine**. En cuanto a las imágenes de las simulaciones, las señales que aparecen en verde (con distintos tonos para diferenciar por algún criterio) se corresponden con las de **image_machine** y demás componentes de TFG_IMAGE_SCALING y en azul (con distintos tonos para diferenciar por algún criterio) las correspondientes a **row_machine** y los distintos componentes implementados en TFG_ROW_INTERPOLATION.

Con el objetivo de dar un primer enfoque del funcionamiento del algoritmo, realizamos una pequeña introducción de cada uno de los estados de ambas máquinas y mostramos el diagrama de flujo del algoritmo en hardware.

Estado de **image_machine**	Duración	Descripción
S_0 (reset)	Múltiples ciclos de reloj	Es el estado inicial de **image_machine**. Cuando se inicia el sistema, se genera un reset o finaliza un ciclo de escalado, **image_machine** está en **S_0**.
S_1 (inicialización)	Un ciclo de reloj	A partir de aquí se da inicio a nuevo un ciclo de escalado y también comienza la sincronización con **row_machine**.
S_2 (acceso al AHB bus)	Múltiples ciclos de reloj	Se inicia una transferencia por el AHB bus. La duración de **S_2** está relacionada con la latencia de acceso al bus (tomar control de este).
S_3 (inicio de transferencia de fila even)	Múltiples ciclos de reloj	Una vez obtenido el control del bus, se inicia la transferencia. En este caso es una lectura de una fila (even) de la imagen fuente, que se almacena en **src_ram**. Antes de que se comiencen a recibir datos existe una latencia de acceso a la memoria externa de 100 ns.
S_4 (entrega de pixeles a **row_machine**)	Múltiples ciclos de reloj	Recibidos los datos de la fila even, mediante un algoritmo decodificador se procede a entregar los datos correspondientes a **row_machine**, que se

		encarga de obtener la pseudo fila even.
S_5 (acceso al AHB bus)	Múltiples ciclos de reloj	Se inicia una transferencia por el AHB bus. La duración de **S_5** está relacionada con la latencia de acceso al bus (tomar control de este)
S_6 (inicio de transferencia de fila odd)	Múltiples ciclos de reloj	Una vez obtenido el control del bus, se inicia la transferencia. En este caso es una lectura de una fila (odd) de la imagen fuente, que se almacena en **src_ram**. Antes de que se comiencen a recibir datos existe una latencia de acceso a la memoria externa de 100 ns.
S_7 (entrega de pixeles a **row_machine**)	Múltiples ciclos de reloj	Recibidos los datos de la fila odd, mediante un algoritmo decodificador se procede a entregar los datos correspondientes a **row_machine**, que se encarga de obtener la pseudo fila odd y la fila correspondiente de la imagen destino, que se almacena en **dst_ram**.
S_8 (acceso al AHB bus)	Múltiples ciclos de reloj	Se inicia una transferencia por el AHB bus. La duración de **S_8** está relacionada con la latencia de acceso al bus (tomar control de este)
S_9 (inicio de transferencia de fila destino)	Múltiples ciclos de reloj	Una vez obtenido el control del bus, se inicia la transferencia. En este caso es la escritura de la fila de la imagen destino almacenada en **dst_ram**. Antes de que se comiencen a enviar datos existe una latencia de acceso a la memoria externa de 100 ns.
S_{10} (finalización de transferencia de fila destino)	Uno/múltiples ciclos de reloj	Al tratarse la transferencia de una escritura, debemos asegurarnos de que el último dato se envió correctamente.
S_{11} (actualización de filas procesadas)	Un ciclo de reloj	Obtenida una fila de la imagen destino, se incrementa el contador que controla el algoritmo de escalado.
S_{12} (comprobación de funcionamiento)	Un ciclo de reloj	Después de incrementar el contador, comprobaremos si la última fila obtenida era la final de la imagen destino. Si era la última fila volveremos a **S_0** a la espera de un nuevo ciclo de escalado, si no se obtendrán las direcciones de las siguientes filas.
S_{13} (actualización de direcciones)	Un ciclo de reloj	Obtiene la dirección de escritura de la siguiente fila de la imagen destino.
S_{14} (actualización de direcciones)	Un ciclo de reloj	Obtiene la dirección de lectura.
S_{15} (actualización de direcciones)	Un ciclo de reloj	A partir de la dirección del estado anterior, se obtienen las direcciones de lectura de la fila even y odd de la imagen fuente.

Tabla 5.19 – Resumen de estados de image_machine

Estado de **row_machine**	Duración	Descripción
S_0 (estado de reset)	Múltiples ciclos de reloj	Es el estado inicial de **row_machine**. Cuando se inicia el sistema, se genera un reset o finaliza un ciclo de escalado, **row_machine** vuelve a **S_0**.
S_1 (estado de espera para fila even)	Múltiples ciclos de reloj	Espera a que **src_ram** se cargue con la fila even de la imagen fuente.
S_2 (estado de obtención de pseudo fila even)	Múltiples ciclos de reloj	Obtiene la pseudo fila even a partir de los datos de la fila almacenada en **src_ram**.
S_3 (estado de reset del contador)	Un ciclo de reloj	Reset del contador que funciona durante **S_2**.
S_4 (estado de espera para fila odd)	Múltiples ciclos de reloj	Espera a que **src_ram** se cargue con la fila odd de la imagen fuente.
S_5 (estado de obtención de pseudo fila odd y fila destino)	Múltiples ciclos de reloj	Obtiene la pseudo fila odd a partir de los datos de la fila almacenada en **src_ram** y la fila correspondiente de la imagen destino.
S_6 (estado de sincronización)	Un ciclo de reloj	Debido a la entrega de los datos de la fila de la imagen destino a **dst_ram**, se necesita un ciclo de sincronización para finalizar correctamente dicho proceso.
S_7 (estado de reset del contador)	Un ciclo de reloj	Reset del contador que funciona durante **S_5**.

Tabla 5.20 – Resumen de estados de row_machine

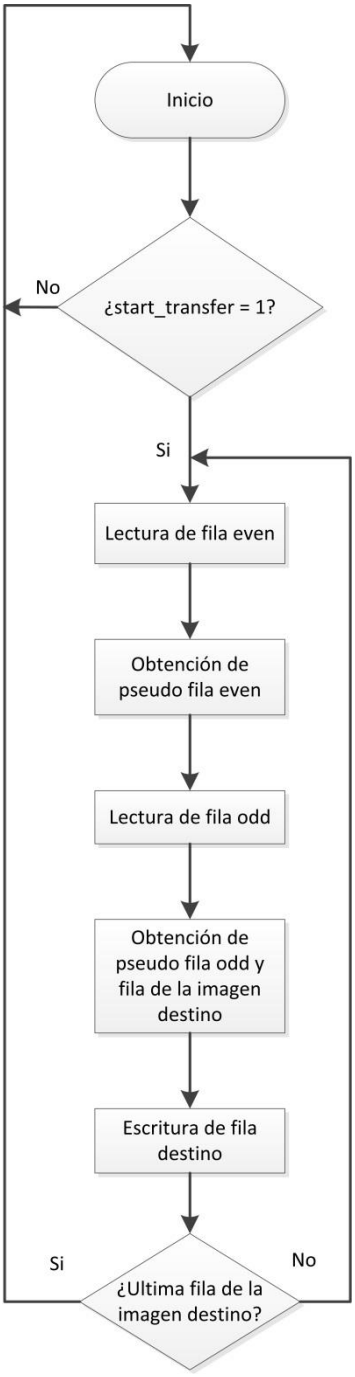


Imagen 5.42 – Diagrama de flujo del algoritmo en hardware

A continuación nos centramos en cada uno de los estados de ambas máquinas. Para ello comenzamos en el estado **S_0** de **image_machine** hasta completar la descripción de todos ellos y procedemos de igual forma con **row_machine**. Las imágenes de la simulación que aquí se presentan corresponden a un escalado desde una imagen fuente de 800x600 a una imagen destino de 640x480.

5.7.1 Image_machine: S_0 (reset)

Como hemos descrito en la introducción, este es el estado en el que se encuentra **image_machine** cada vez que se inicializa el sistema o se genera un reset y cada vez que finaliza un ciclo de escalado. Es decir, todos los componentes y elementos están en reposo

(reseteados), aunque esto no quiere decir que no estén excitados (inicializados a algún valor concreto), como es el caso de **status_REG**, que está a cero para indicar al software que está disponible para iniciar un ciclo de escalado e **image_scaling_state**, que también valde cero, correspondiéndose con el estado en el que se encuentra **image_machine**, para establecer la sincronización con **row_machine**.

Así pues, el comienzo de un nuevo ciclo de escalado viene marcado por **image_machine** al detectar un nivel alto en **start_transfer** (líneas 710 a 715).

start_transfer es una señal que dura un solo ciclo de reloj **Clk**. Esto es posible gracias al bit cero del registro **control_REG** y el proceso de detección de eventos descrito en el código (líneas 810 a 816).

Su funcionamiento se puede ver en la siguiente imagen.

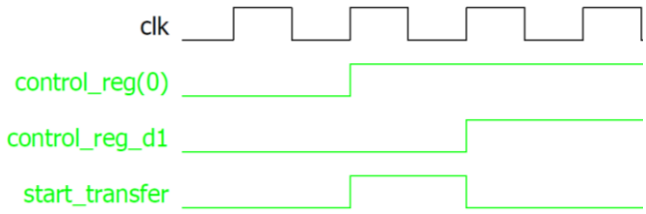


Imagen 5.43 – Detector de eventos para start_transfer

El valor de **control_REG** se modifica desde el software que funcionará en el procesador a través del esclavo TFG_APB_SLAVE (verde oscuro en la siguiente imagen), y de entre todos los registros aquí configurables, este es el último en ser modificado.

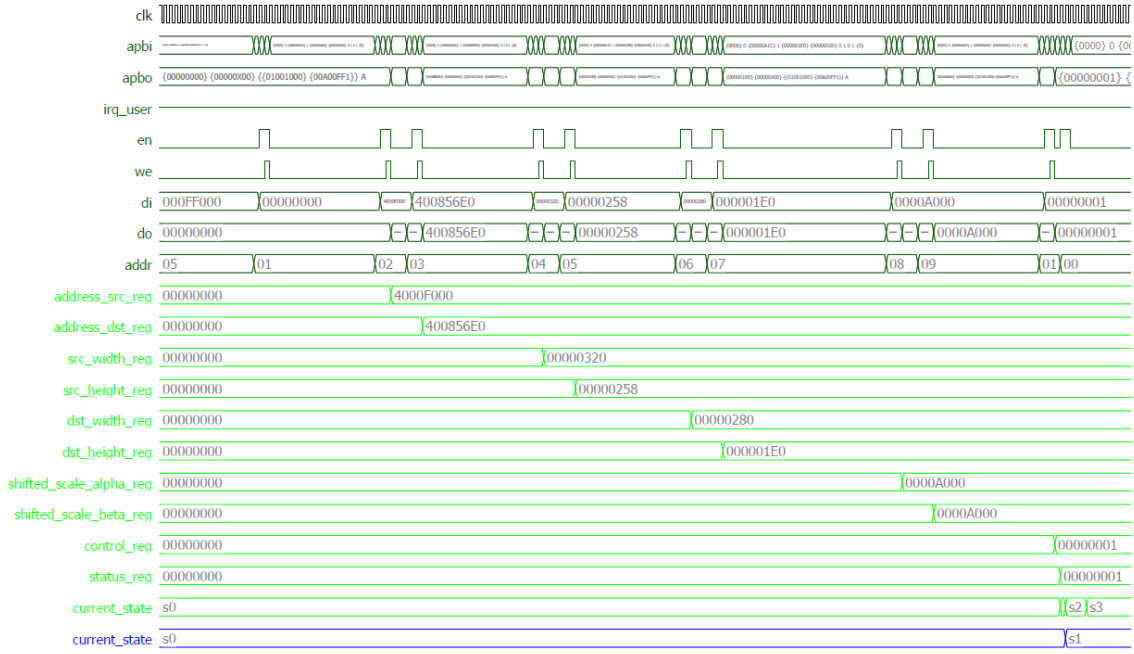


Imagen 5.44 – Configuración del bloque de registros

Nota: **status_REG** es un registro de solo lectura.

Puesto que el esclavo solo se usa para configurar dichos registros, pasamos a describir su comportamiento a partir de la siguiente imagen.

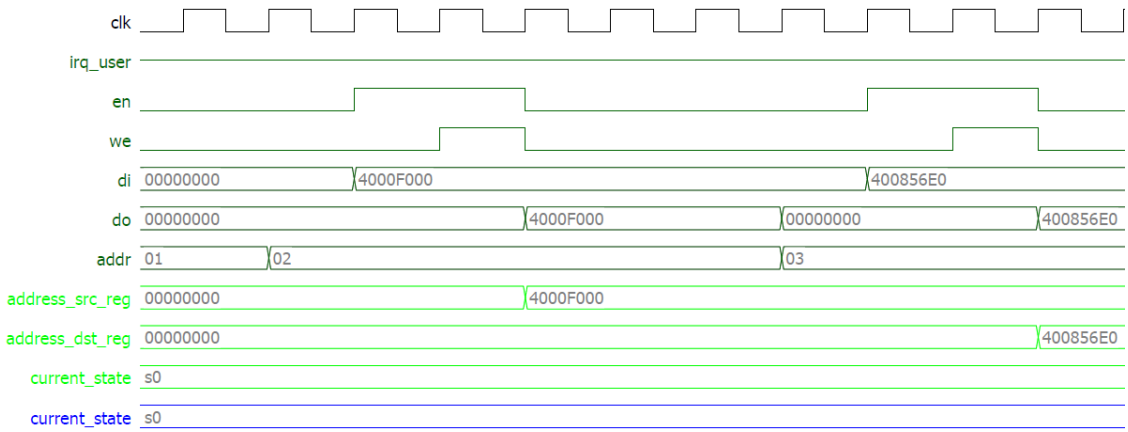


Imagen 5.45 – Funcionamiento de TFG_APB_SLAVE

Su funcionamiento es muy sencillo al tratarse de un esclavo, pues solo puede actuar cuando se le requiere. Así pues, para poder leer el dato **di** presente en el ABP bus debe espera a que **en** y **we** estén en nivel alto, momento en el que dicho valor pasa al registro que indique **addr**. Cuando se le requiera para realizar una escritura del contenido de los registros en el APB bus no se necesita realizar ninguna acción, puesto que el valor de dichos registros siempre está disponible, para que según indique **addr** se cargue el valor del registro en **do**. Esto se puede ver en el código en las líneas 275 a 337.

Dado que **start_transfer** dura un solo ciclo de reloj (para evitar que automáticamente se inicie un nuevo ciclo de escalado de imagen tras finalizar el que estaba en proceso), se debe poner a cero el bit cero de **control_REG**, y a continuación a uno para poder comenzar un nuevo ciclo de escalado.

Tras esto, **image_machine** se encontra en $\mathbf{S_1}$ mientras que **row_machine** sigue en $\mathbf{S_0}$, como se puede apreciar en la siguiente imagen.

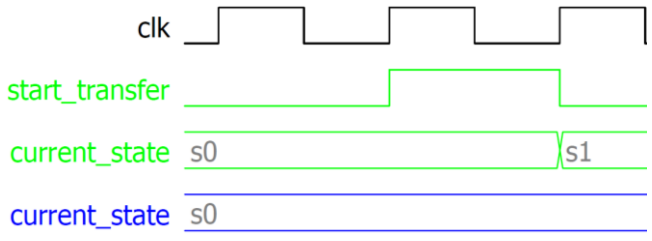


Imagen 5.46 – Transicion de S_0 a S_1 de image_machine

Antes de continuar con la descripción de los demás estados de **image_machine**, vemos el esquema de conexión de los distintos componentes implementados en TFG_IMAGE_SCALING (descritos en los apartados 5.6.3 a 5.6.6) para tener una idea del flujo de datos que se describe en los siguientes apartados.

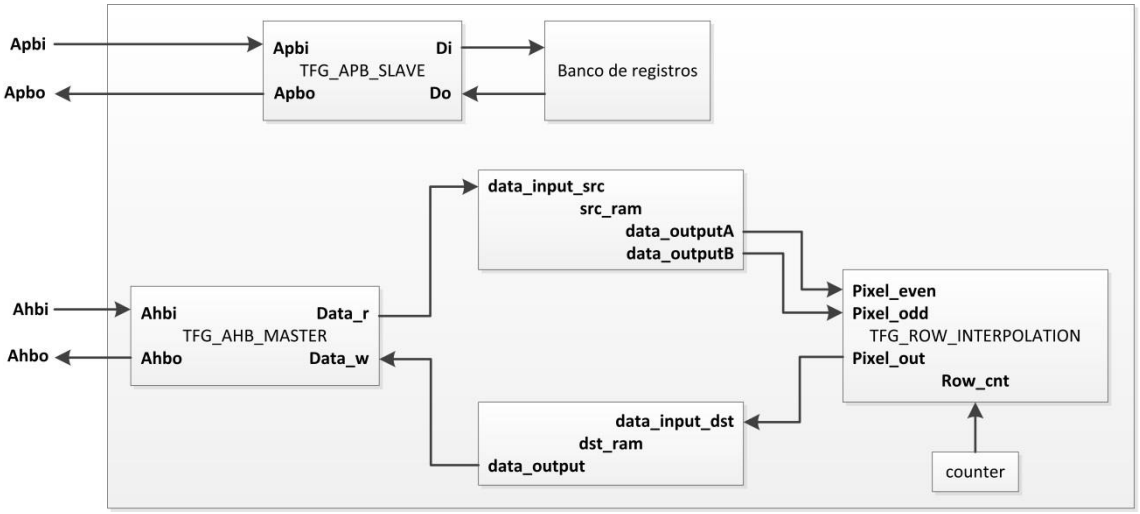


Imagen 5.47 – Pseudoflujo de datos de componentes de TFG_IMAGE_SCALING

5.7.2 Image_machine: S_1 (inicialización)

Es aquí donde comienza la sincronización con **row_machine**. También asumimos que a partir de $\mathbf{S_1}$ se ha iniciado un nuevo ciclo de escalado, por lo que el bit cero de **status_REG** se mantiene a uno hasta que finaliza y volvemos al estado $\mathbf{S_0}$ de **image_machine**. Este estado tiene la misma funcionalidad que $\mathbf{S_0}$.

Con el siguiente ciclo de reloj, **image_machine** pasa a $\mathbf{S_2}$ y **row_machine** a $\mathbf{S_1}$, como se puede observar en la siguiente imagen.

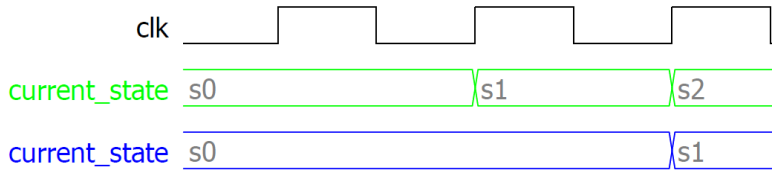


Imagen 5.48 – Transicion de S_1 a S_2 de image_machine

5.7.3 Image_machine: S_2 (acceso al AHB bus)

Cuando **image_machine** se encuentra en el estado $\mathbf{S_2}$, se inicia un ciclo de lectura de datos de la memoria externa. Así pues, se realiza aquí la configuración de dicha transferencia.

Como se puede ver en el código (líneas 411 a 417), establecemos una transferencia con un número indeterminado de datos (**transfer_type** = 1) a recibir de 32 bits (**size** = 2). La dirección de lectura se corresponde con **row_address_read_even** (dirección de lectura de la fila even de la imagen fuente). Dicho valor se obtiene según vemos en el código en las líneas 839 a 859.

Calculamos primero **row_address_read** a partir de la dirección de inicio de la imagen fuente (**address_src_REG**) añadiendo el offset que se obtiene con el producto del ancho o número de columnas de la imagen fuente (**src_width_REG**) por el índice de filas **embedd_yofs**. A continuación obtenemos **row_address_read_even** a partir del valor de **row_address_read**.

Dado que hemos optado por una transferencia de datos de 32 bits, nos vemos obligado a acceder a la memoria externa con direcciones múltiplos de 4. Es por ello que en **addro** se carga el valor de los 30 bits más significativos de **row_address_read_even** y los dos menos

significativos son cero. Si por el contrario hubiéramos optado por transferencias tipo byte, se cargaría el valor completo de **row_address_read_even**.

Tras determinar **addro**, configurado **dwrite** en modo lectura (nivel bajo) y con **start** en nivel alto, hay una latencia de acceso al AHB bus, de modo que cuando esta finaliza y tengamos el control del AHB bus, se inicia la transferencia. Esto se determina mediante **transfer_active** (pasa a nivel alto) y **transfer_end** (debe estar en nivel bajo), momento en el que **image_machine** pasa de **S_2** a **S_3**.

En la siguiente imagen podemos observar esta trama.

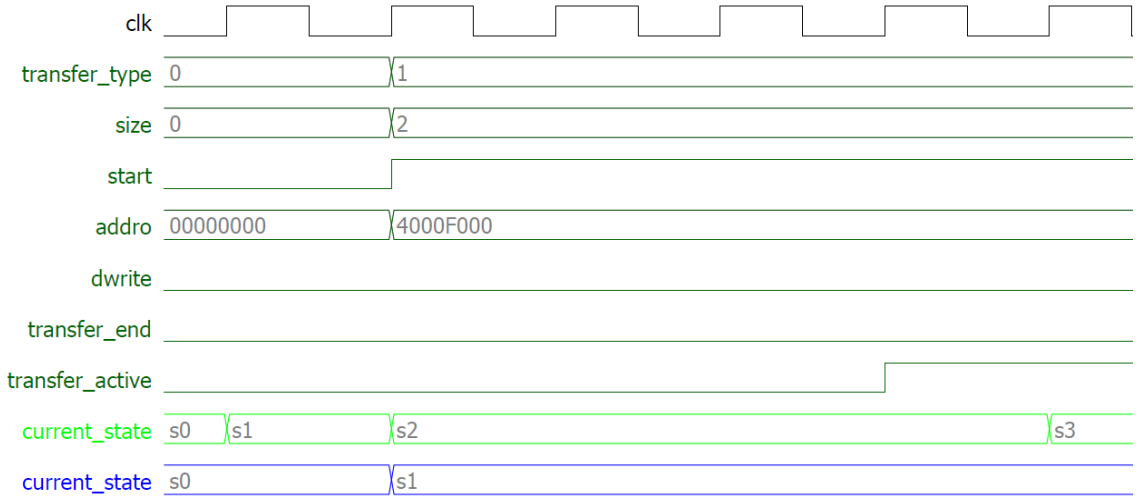


Imagen 5.49 – Latencia de inicio de transferencia (acceso al AHB bus) en S_2

5.7.4 Image_machine: S_3 (transferencia de fila even)

Una vez obtenido el control del AHB bus, tenmos otra latencia de acceso a la memoria externa de aproximadamente 100 ns, tras la que comienza la transferencia.

Ahora bien, si volvemos a lo comentado en el apartado anterior, la elección del tipo de transferencia se debe claramente a la intención de acelerar al máximo el proceso de lectura de los datos de la memoria externa. Con ello se consigue la lectura de una fila cuatro veces más rápido. Pero esto también deriva en pequeñas complicaciones para los siguientes pasos del algoritmo. El más importante de ellos, que tiene que ver con "el tipo de imágenes", lo representamos gráficamente a continuación:

Imagen 5.50 – Tipos de imágenes contempladas en el código

En la imagen anterior se han representado cuatro filas de los cuatro tipos de imágenes que se diferencian en el algoritmo implementado. Estos cuatro tipos se corresponden con las imágenes cuyo número de pixeles por fila es múltiplo de cuatro, múltiplos de cuatro más un

píxel libre, múltiplos de cuatro más dos pixeles libres y múltiplos de cuatro más tres pixeles libres.

Si suponemos (y ha de ser así, ya que si no se debería modificar el código introduciendo un parámetro de offset de dirección) que las imágenes están alineadas en la memoria externa, es decir, direcciones cuyo nibble (4 bits) menos significativo sea múltiplo de cuatro, a la hora de querer leer las filas de la imagen fuente y cargarlas en **src_ram**, debemos tener en cuenta el tamaño de dichas las filas.

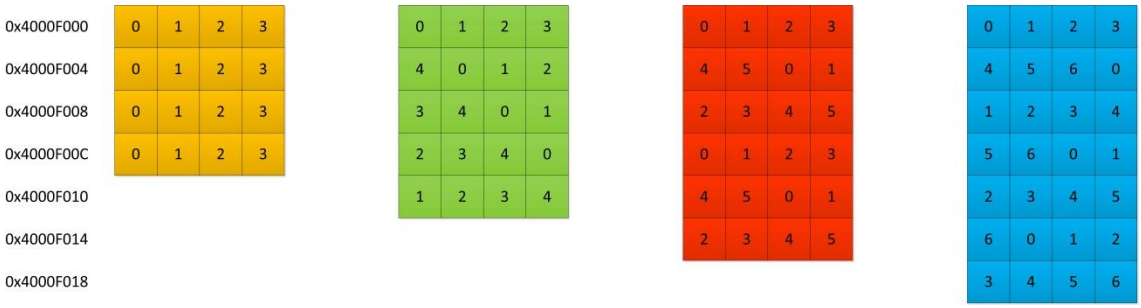


Imagen 5.51 – Alineamiento de filas de los cuatro tipos de imágenes en memoria

Nota: se han colocado las filas en matrices de nx4 para observar el efecto de los pixeles libres.

Esto es, si el tamaño de la fila es múltiplo de cuatro, no tenemos problemas, ya que cada fila comienza en una dirección accesible. Así pues, el número de datos a leer es un número entero e igual a:

$$src\_width\_adjusted = \frac{src\_width\_REG}{4}$$

En cambio, si pretendemos leer una fila no múltiplo de cuatro nos podemos encontrar dos casos diferentes.

El primero concierne a las imágenes no múltiplos de cuatro con uno y dos pixeles libres. Eso se debe a que para obtener una fila de la imagen fuente se necesita leer un número de datos igual a:

$$src\_width\_adjusted = \frac{src\_width\_REG}{4} + 1$$

Si observamos la imagen anterior, en el bloque verde y el bloque rojo, para obtener una fila completa hace falta leer de dos direcciones con respecto al bloque naranja, del cual solo habríamos de leer en una dirección para obtener una fila.

El otro caso es para las imágenes con tamaño de filas no múltiplos de cuatro con tres pixeles libres, para las que se pueden necesitar:

$$src\_width\_adjusted = \frac{src\_width\_REG}{4} + 1$$

o

$$src\_width\_adjusted = \frac{src\_width\_REG}{4} + 2$$

Esto se puede apreciar en el bloque azul, ya que para obtener la primera fila habríamos de leer en dos direcciones pero para obtener la segunda habríamos de leer en tres direcciones. En el código (líneas 876 a 897) se ha implementado la segunda ecuación, para reducir el algoritmo.

Se ha de aclarar que en este aspecto no influye que el número de filas de la imagen fuente sea o no múltiplo de cuatro.

Los datos que reciba el maestro son cargados en **src_ram**, la cual está configurada para albergar todos los pixeles de una fila de la imagen fuente mediante el parámetro **SRC_ADDRESS_WIDTH**. Este valor, como se introdujo en el apartado 5.6.2, es el exponente de la potencia de dos igual o inmediatamente superior al número de columnas de la imagen fuente dividido por 4 (datos de 32 bits, equivalente a 4 bytes o pixeles), que se puede ver aumentado en función del tipo de imagen (debido a lo explicado anteriormente en relación al número de datos que se han de guardar en dicha memoria). A continuación vemos unos ejemplos de su cálculo. Se han cogido tamaños alrededor de una potencia de dos (128) para poder contemplar los casos críticos.

src_width_REG	**src_width_REG**/4	Potencia de dos inmediatamente superior	Tipo de imagen (**src_width_ajusted**)	Nueva potencia de dos inmediatamente superior	**SRC_ADDRESS_WIDTH**
120	30	$2^5 = 32$	0	$30 \rightarrow 2^5 = 32$	5
121	30.25	$2^5 = 32$	+1	$31.25 \rightarrow 2^5 = 32$	5
122	30.5	$2^5 = 32$	+1	$31.5 \rightarrow 2^5 = 32$	5
123	30.75	$2^5 = 32$	+1 o +2	$32.75 \rightarrow 2^6 = 64$	6
124	31	$2^5 = 32$	0	$31 \rightarrow 2^5 = 32$	5
125	31.25	$2^5 = 32$	+1	$32.25 \rightarrow 2^6 = 64$	6
126	31.5	$2^5 = 32$	+1	$32.5 \rightarrow 2^6 = 64$	6
127	31.75	$2^5 = 32$	+1 o +2	$33.75 \rightarrow 2^6 = 64$	6
128	32	$2^5 = 32$	0	$32 \rightarrow 2^5 = 32$	5
129	32.25	$2^6 = 64$	+1	$33.25 \rightarrow 2^6 = 64$	6
130	32.5	$2^6 = 64$	+1	$33.5 \rightarrow 2^6 = 64$	6
131	32.75	$2^6 = 64$	+1 o +2	$34.75 \rightarrow 2^6 = 64$	6

Tabla 5.21 – obtención de valores para SRC_ADDRESS_WIDTH

Como se puede ver, tenemos que controlar el valor de **SRC_ADDRESS_WIDTH** en las imágenes de tamaño 123, 125, 126 y 127, ya que el tipo de imagen hace que se exceda el valor del exponente de la potencia de dos precalculada. Lo mismo ocurre en las demás imágenes con tamaños cercanos a otras potencias de dos.

Estos valores son el mínimo para optimizar al máximo el consumo de recursos, pero se pueden usar valores superiores siempre que no superen los correspondientes a imágenes full HD (1920x1080, realmente el limite es la potencia de dos inmediatamente superior, 2048x2048).

Src_ram está activa (**ram_enable**) en todos los estados de **image_machine**, es decir, siempre se puede realizar una lectura de su contenido. Ahora bien, la escritura (**write_enable_src**) está controlada por **data_valid**, de manera que solo se guardan datos cuando estos son correctos.

La dirección se toma de un retraso de un ciclo de reloj (**beat_counter_d1**) realizado sobre **beat_counter** (líneas 818 a 837).

Al finalizar la transferencia (líneas 744 a 749) **image_machine** pasa a **S_4** y con el siguiente ciclo de reloj **row_machine** a **S_2**. En la siguiente imagen podemos ver la descripción gráfica de todo lo desarrollado en este apartado.

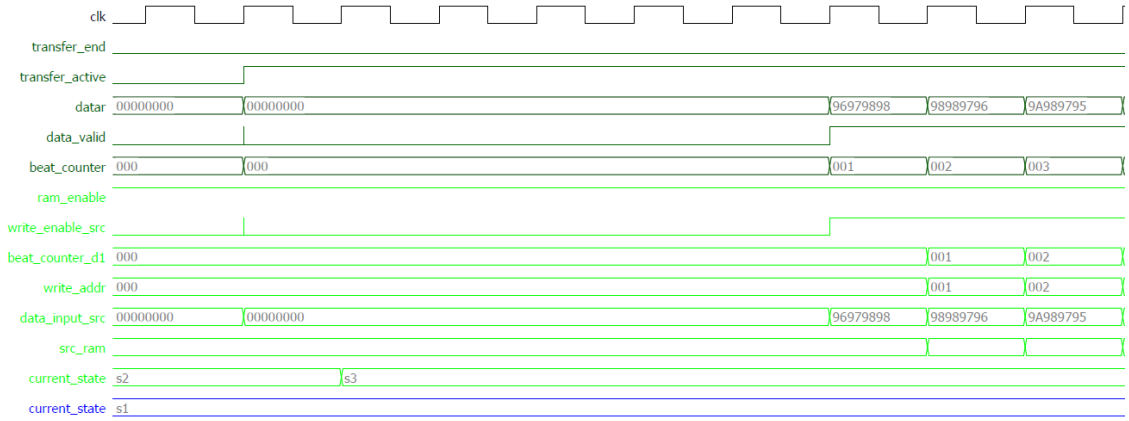


Imagen 5.52 – Inicio de la transferencia en S_3

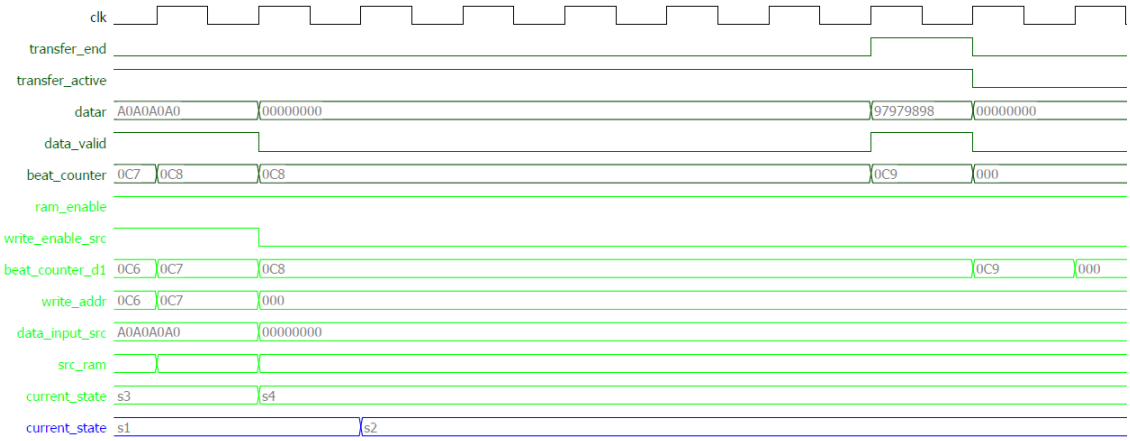


Imagen 5.53 – Finalización de la transferencia y transición de S_3 a S_4

Nota: debido a limitaciones gráficas del simulador, no se han podido representar los datos almacenados en **src_ram**.

5.7.5 Image_machine: S_4 (entrega de pixeles a row_machine)

Una vez finalizada la transferencia de los datos de la fila even, se procede a entregar a **row_machine** los pixeles que se requieran, según el índice de columnas **embedd_xofs**.

Aquí de nuevo volvemos a encontrar una dificultad en el algoritmo al haber almacenado los datos con tamaño de 32 bits, ya que no podemos acceder directamente a un píxel en concreto. Primero debemos obtener la palabra de 32 bits que lo contiene y a continuación sacar de dicha palabra el píxel requerido. Pero para ello contamos con la señal **address_index**, cuyo funcionamiento describimos a continuación (líneas 927 a 934).

Su valor se obtiene a partir del índice de columnas **embedd_xofs** y un offset denominado **index_offset**. Este último se obtiene (líneas 912 a 925) según indique **sel** de las direcciones de lectura de las filas even u odd. Equivale a un offset acumulativo que se debe al tipo de imagen, de ahí que se escojan los dos bits menos significativos de las direcciones de lectura, que son los que dan a conocer el número de pixeles libres. Nos valemos de un ejemplo gráfico para aclarar esto.

0x4000F000	0	1	2	3
0x4000F004	4	0	1	2
0x4000F008	3	4	0	1
0x4000F00C	2	3	4	0
0x4000F010	1	2	3	4

Imagen 5.54 – Ejemplo para la explicación de index_offset

Para la primera fila, la dirección de lectura es 0x4000F000, con lo que **index_offset** vale cero, debido a que el inicio de esta fila se encuentra alineado en memoria. No ocurre lo mismo con las siguientes filas. Por ejemplo, para la tercera, leemos de 0x4000F00A, por lo que **index_offset** vale dos (0xB = 1010 y se escogen los dos bits menos significativos). Este valor es necesario debido a que en **src_ram**, la primera palabra que se guarda puede contener pixeles de la fila anterior, por lo que **address_index** ha de ser consciente de ello para poderlos ignorar.

Calculado **address_index**, asignamos a las direcciones de lectura de **src_ram** (**read_address_outputA** para la salida **data_outputA** y **read_address_outputB** para la salida **data_outputB**) dicho valor dividido por cuatro (de ahí " downto 2"), ya que necesitamos accesos de tamaño word. En este punto debemos ser conscientes de cómo se escogen los pixeles que se entregan a **row_machine**. **Address_index** apunta a la palabra que contiene el **píxel_even** (contenido en la salida **data_outputA**), pero necesitamos también el **píxel_odd** (contenido en la salida **data_outputB**), que es el siguiente a **píxel_even**. Aquí pueden darse ciertos casos. Nos ayudamos de la siguiente imagen para explicarlos.

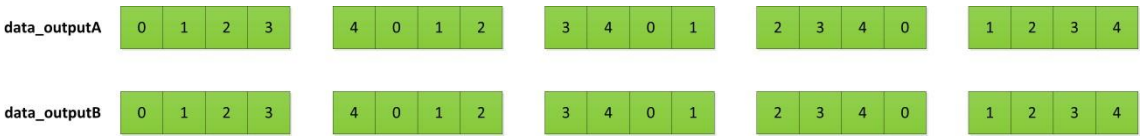


Imagen 5.55 – Ejemplo para la explicación de address_index

En el ejemplo **index_offset** es cero (el inicio de la fila coincide con el inicio de **src_ram**). A continuación obtenemos una tabla de valores de **address_index** para ver varios casos.

address_index	read_address_outputA	read_address_outputB	píxel_even	píxel_odd
0	0	0	0	1
1	0	0	1	2
2	0	0	2	3
3	0	0	3	0
4	1	1	4	0
5	1	1	0	1
6	1	1	1	2
7	1	1	2	4

Tabla 5.22 – Obtención de valores para read_address_outputA y read_address_outputB

Si nos fijamos, hay valores de **píxel_odd** marcados en rojo. Estos pixeles son incorrectos, ya que si **píxel_even** se encuentra en el byte menos significativo de la palabra que lo contiene, necesitamos acceder a la siguiente palabra para obtener el **píxel_odd**. Es por ello que tenemos en el código (líneas 461 a 467) implementada una sentencia *if* que permite saltar a la palabra siguiente cuando sea necesario.

Para la selección de los pixeles se sigue un sencillo algoritmo, como se puede ver en el código (líneas 936 a 959), que consiste en asignar a **píxel_even** y **píxel_odd** el byte correspondiente de los datos obtenidos en **data_outputA** y **data_outputB** según un demultiplexor implementado con el retraso **address_index_d1**, debido a que el dato que se pide a **src_ram** aparece un ciclo de reloj más tarde.

En la siguiente imagen podemos ver gráficamente el proceso explicado anteriormente.

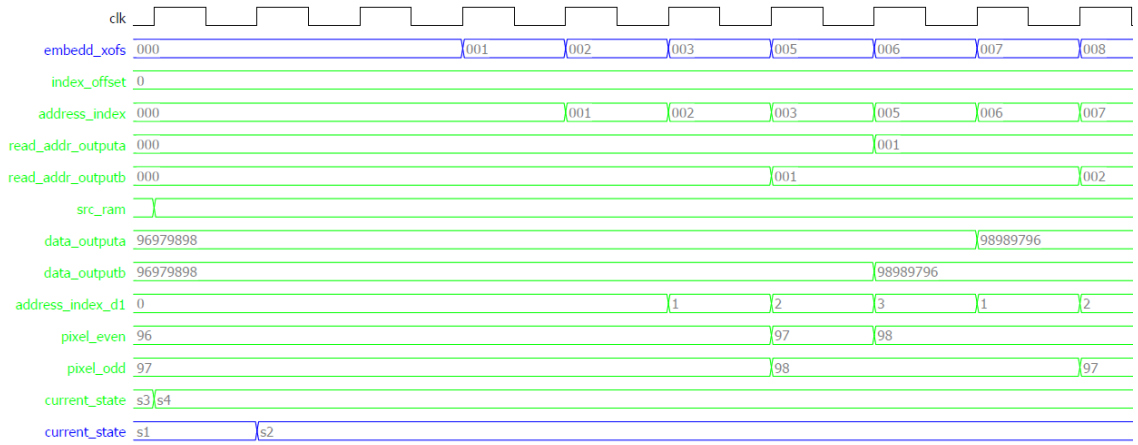


Imagen 5.56 – Funcionamiento del algoritmo de selección de pixeles

Aquí solo se han representado las señales presentes en **image_machine** (vemos en el apartado 5.7.17 la descripción de la obtención de **embedd_xofs** a partir del contador denominado **dx**). El valor de **embedd_xofs** aparece un ciclo de reloj después del cambio en **dx**, de ahí que este valga cero durante dos ciclos de reloj tras haber pasado **row_machine** a $\mathbf{S_2}$. Dado que **index_offset** ya estaba calculado, en el siguiente ciclo de reloj se obtiene el primer valor de **address_index** (valor 0x0 debajo del 0x001 de **embedd_xofs**). En el mismo ciclo de reloj dicho valor dividido por cuatro es asignado a las direcciones de lectura **read_address_outputA** y **read_address_outputB**, y si nos fijamos, cuando **address_index** vale 0x003, **read_address_outputA** vale aún 0x0 pero **read_address_outputB** apunta a la siguiente

palabra, 0x001, almacenada en **src_ram**. Las dos primeras palabras correctas que aparecen en **data_outputA** y **data_outputB** son las que coinciden con el 0x001 de **address_index**, y un ciclo de reloj después se tienen disponibles los dos primeros valores de **píxel_even** (0x96) y **píxel_odd** (0x97) seleccionados por **address_index_d1**. **Image_machine** permanece en **S_4** hasta que se detecte **S_3** en **row_machine**.

5.7.6 Image_machine: S_5 (acceso al AHB bus)

Este estado tiene la misma funcionalidad que en el apartado 5.7.3, salvo por la dirección **addro** de la que leemos dicha fila, que toma el valor de **row_address_read_odd**.

Tras determinar **addro**, configurado **dwrite** en modo lectura (nivel bajo) y con **start** en nivel alto, hay una latencia de acceso al AHB bus, de modo que cuando esta finaliza y tengamos el control del AHB bus, se inicia la transferencia. Esto se determina mediante **transfer_active** (pasa a nivel alto) y **transfer_end** (debe estar en nivel bajo), momento en el que **image_machine** pasa de **S_5** a **S_6**. En la siguiente imagen podemos observar esta trama.

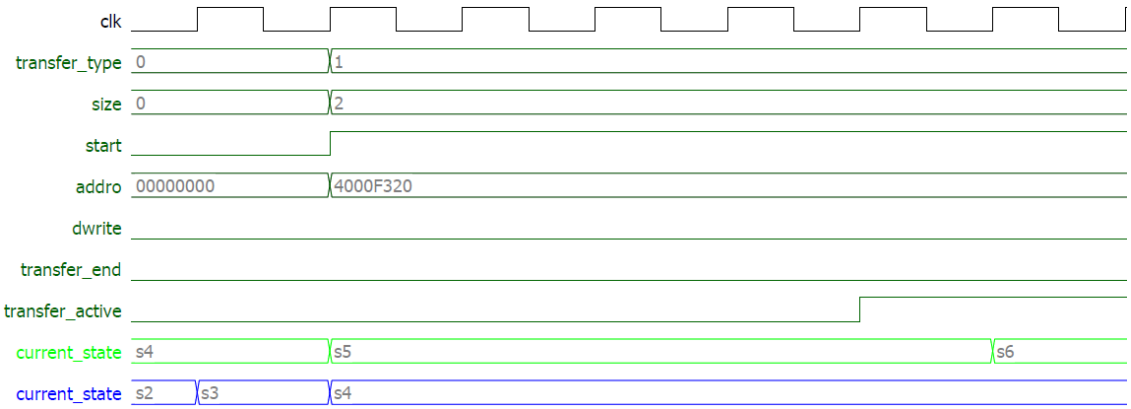


Imagen 5.57 – Latencia de inicio de transferencia (acceso al AHB bus) en S_3

5.7.7 Image_machine: S_6 (transferencia de fila odd)

En este estado, al igual que ocurre en el anterior, se tiene la misma funcionalidad que en el apartado 5.7.4, salvo por la dirección **addro**. Así pues, cuando **image_machine** finaliza la transferencia de la fila odd pasa a **S_7** y un ciclo de reloj después, **row_machine** pasa a **S_5**. En las siguientes imágenes podemos ver el proceso.

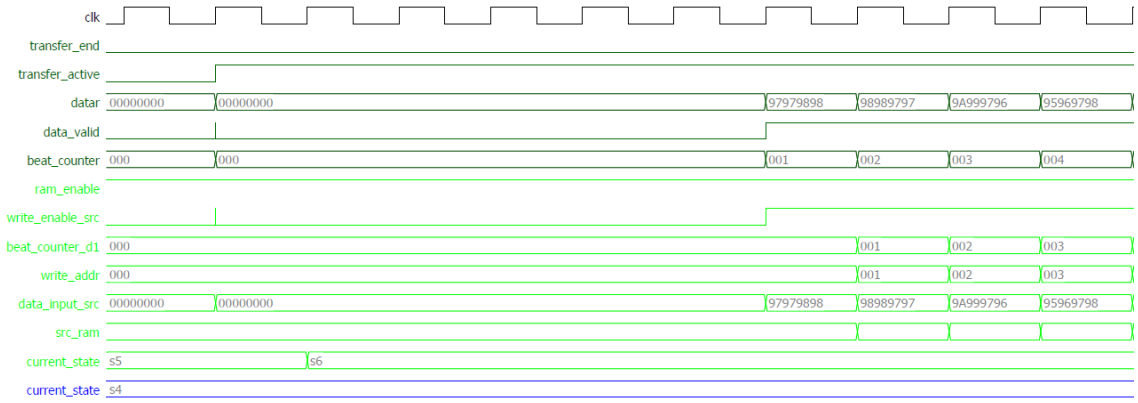


Imagen 5.58 – Inicio de la transferencia en S_6

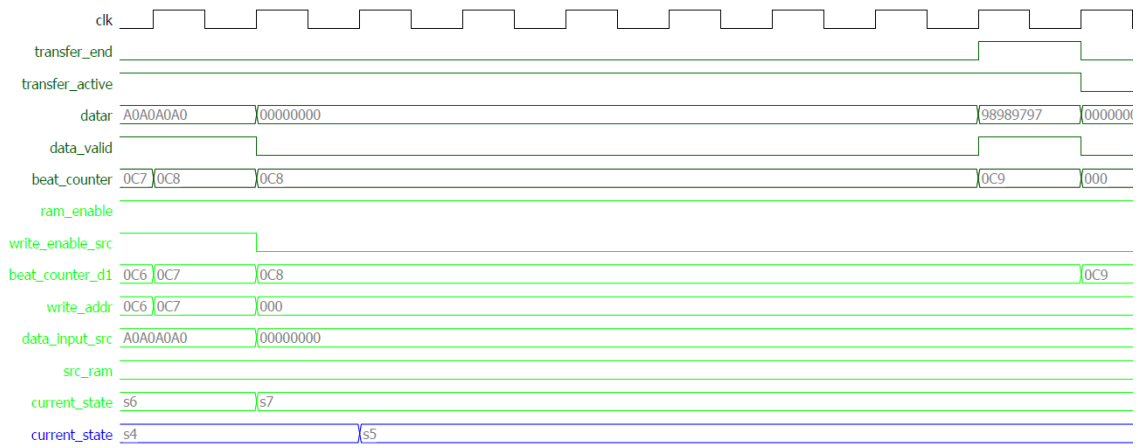


Imagen 5.59 – Finalización de la transferencia y transición de S_6 a S_7

5.7.8 Image_machine: S_7 (entrega de pixeles a row_machine)

Al igual que en el apartado 5.7.5, **image_machine** se encarga de enviar los valores de los pixeles que solicita **row_machine**.

Una vez se comienzan a generar los pixeles de la imagen destino, entra en funcionamiento un algoritmo para componer palabras de 32 bits a partir de cada 4 **píxel_out**.

A continuación vemos paso a paso la descripción de cada uno de los procesos que controlan el algoritmo de generación de palabras.

El primer paso consiste en guardar en señales temporales (**byte0** a **byte3**) los datos que aparecen en **píxel_out**. Así pues, el primero de cada fila se guarda siempre en **byte0** y los posteriores se van guardando en **byte1**, **byte2**, **byte3**, **byte0**,... Este proceso se ayuda de **internal_address_d2**, como podemos observar en el código (líneas 961 a 982).

En la siguiente imagen podemos observar su funcionamiento.

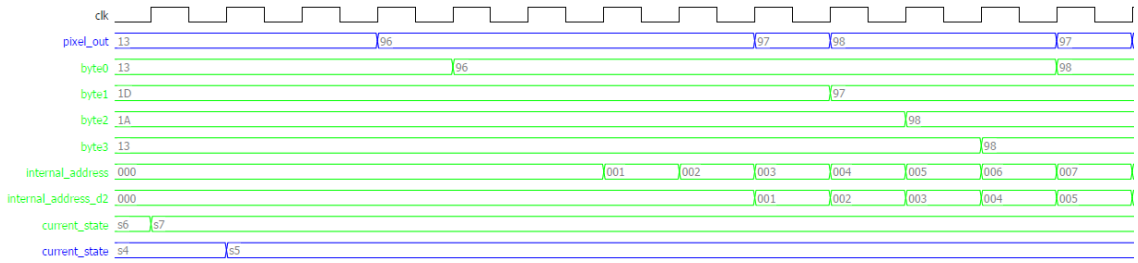


Imagen 5.60 – Asignación de pixel_out a las señales byte0 a byte3

El primer **píxel_out** valido es el que corresponde al valor 0x02 de **internal_address** (su obtención se ve en el apartado 5.7.20), por lo que es **internal_address_d2** quien se encuentra sincronizado con la generación de estos (el valor aparece registrado un ciclo de reloj después).

Una vez se están registrando los datos de **píxel_out** en las señales **byte0**, **byte1**, **byte2** y **byte3**, procedemos a la composición de las palabras de 32 bits que se guardan en **dst_ram** (líneas de código 999 a 1022).

Su funcionamiento se puede apreciar en la siguiente imagen.

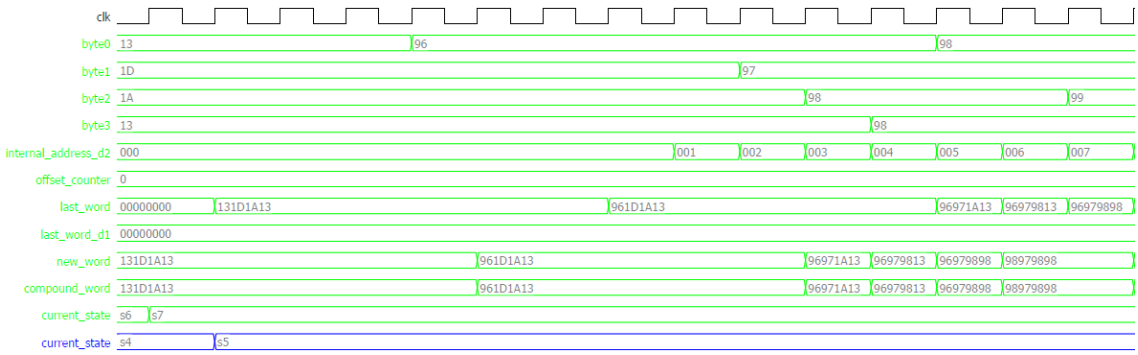


Imagen 5.61 – Generación de new_word y compund_word

Aquí aparecen de nuevo varias señales. La primera de ellas y más interesante es **offset_counter**. Esta se obtiene como se puede ver en el código (líneas 861 a 874).

offset_counter tiene la misma funcionalidad que **index_offset**, pero en este caso para las filas de la imagen destino. Es por ello que toma su valor de los dos bits menos significativos de **dst_width_REG**.

La finalidad de esto reside en que necesitamos que las filas de la imagen destino estén consecutivas en memoria. Para ello nos valemos de **last_word_d1** (retraso especial de **last_word**, registra el valor cuando termina la escritura en **dst_ram**), que guarda la última palabra que se ha almacenado en **dst_ram**.

En el ejemplo de la simulación tenemos que **offset_counter** vale cero ya que la imagen destino tiene un ancho o número de columnas múltiplo de cuatro. Esto determina que las palabras que se guardan en **dst_ram** están solo formadas por las señales **byte0** a **byte3**, y no es necesario tener en cuenta la última guardada, puesto que está formada al completo por pixeles validos de la fila en cuestión.

Cuando tenemos filas de tamaño no múltiplo de cuatro, la última palabra que se guarda en **dst_ram** (y en consecuencia en la memoria externa) no está compuesta entera por pixeles válidos, así que debemos sobrescribir en la dirección de la última palabra que se manda a la memoria externa con otra palabra con los pixeles de la fila anterior y con los pixeles de la siguiente fila hasta completar la palabra de 32 bits. En este aspecto entra en juego **compound_word**. Esta se genera a partir de **last_word_d1** y de las señales **byte0** a **byte3**, y es la primera que se guarda en **dst_ram** (las siguientes son **new_word**), conteniendo así los pixeles de la última fila y los del inicio de la siguiente, de forma que al mandarlas a la memoria externa se sobrescriben los que ya existen, consiguiendo que las filas de la imagen destino estén consecutivas.

Como hemos adelantado, **compound_word** es siempre la primera que se guarda en **dst_ram** y a continuación es **new_word** quien siga proporcionando datos. Para ello nos valemos del código (líneas 1024 a 1045) para multiplexar la entrada de **dst_ram**.

Como se puede apreciar, aquí se tienen más retrasos de **internal_address**. Estos se deben a que, mientras que con más pixeles nuevos se tenga que rellenar **compound_word**, mas ciclos de reloj se ha de esperar, puesto que necesitamos más datos de **píxel_out**.

Una vez **new_word** o **compund_word** (la que corresponda) están listas se procede a guardarlas en **dst_ram**. Como necesitamos accesos de 32 bits, los obtenemos ignorando los dos bits menos significativos (dividir por cuatro) del retraso correspondiente de **internal_address**, como podemos observar en el código (líneas 533 a 543). Dado que **compound_word** solo se usa una vez por fila, la segunda condición para guardar dichos datos en **dst_ram** es que la dirección de esta apunte a la primera palabra, lo cual se obtiene ignorando los dos bit menos significativos del retraso correspondiente de **internal_address** y siendo cero los demás. En la siguiente imagen podemos ver su funcionamiento.

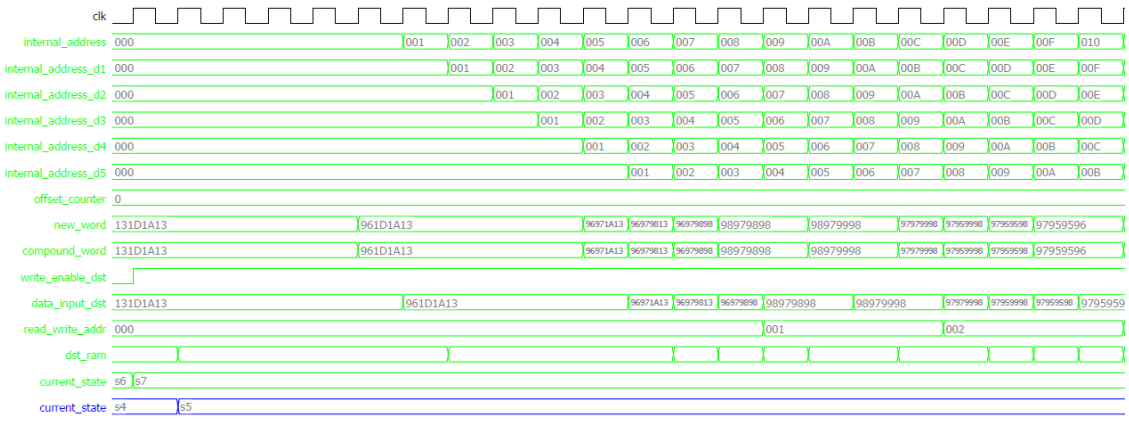


Imagen 5.62 – Almacenamiento de datos en dst_ram

Al igual que hicimos con **src_ram**, procedemos a obtener el parámetro que determina el tamaño de **dst_ram**, **DST_ADDRESS_WIDTH**. Este valor, como se introdujo en el apartado 5.6.2, es el exponente de la potencia de dos igual o inmediatamente superior al número de columnas de la imagen destino dividido por 4, el cual se puede ver aumentado en función del tipo de imagen. A continuación vemos unos ejemplos de su cálculo. Se han cogido tamaños alrededor de una potencia de dos (512) para poder contemplar los casos críticos.

dst_width_REG	**dst_width_REG**/4	Potencia de dos	Tipo de imagen (**dst_width_ajusted**, se verá en el apartado 5.7.10)	Nueva potencia de dos	**DST_ADDRESS_WIDTH**
504	126	$2^7 = 128$	0	$127 \rightarrow 2^7 = 128$	7
505	126.25	$2^7 = 128$	+1	$127.25 \rightarrow 2^7 = 128$	7
506	126.5	$2^7 = 128$	+1	$127.5 \rightarrow 2^7 = 128$	7
507	126.75	$2^7 = 128$	+1 o +2	$128.75 \rightarrow 2^8 = 256$	8
508	127	$2^7 = 128$	0	$128 \rightarrow 2^7 = 128$	7
509	127.25	$2^7 = 128$	+1	$128.25 \rightarrow 2^8 = 256$	8
510	127.5	$2^7 = 128$	+1	$128.5 \rightarrow 2^8 = 256$	8
511	127.75	$2^7 = 128$	+1 o +2	$129.75 \rightarrow 2^8 = 256$	8
512	128	$2^7 = 128$	0	$128 \rightarrow 2^7 = 128$	7
513	128.25	$2^8 = 256$	+1	$129.25 \rightarrow 2^8 = 256$	8
514	128.5	$2^8 = 256$	+1	$129.5 \rightarrow 2^8 = 256$	8
515	128.75	$2^8 = 256$	+1 o +2	$130.75 \rightarrow 2^8 = 256$	8

Tabla 5.23 – Obtención de valores para DST_ADDRESS_WIDTH

Como se puede ver, tenemos que controlar el valor de **DST_ADDRESS_WIDTH** en las imágenes de tamaños 507, 509, 510 y 511, ya el tipo de imagen hace que se exceda el valor del exponente la potencia de dos precalculada.

Dst_ram está activa (**ram_enable**) en todos los estados de **image_machine**, es decir, siempre se puede realizar una lectura de su contenido. Ahora bien, la escritura (**write_enable_dst**) solo está activa en este estado.

A continuación vemos una simulación en la que **offset_counter** tiene un valor distinto de cero para comprender mejor su funcionamiento (escalado de 800x600 a 643x480).

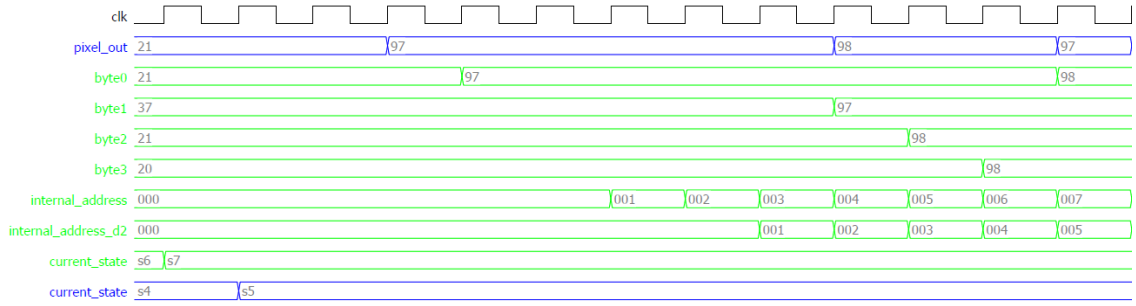


Imagen 5.63 – Asignación de pixel_out a las señales byte0 a byte3 (offset_counter = 3)

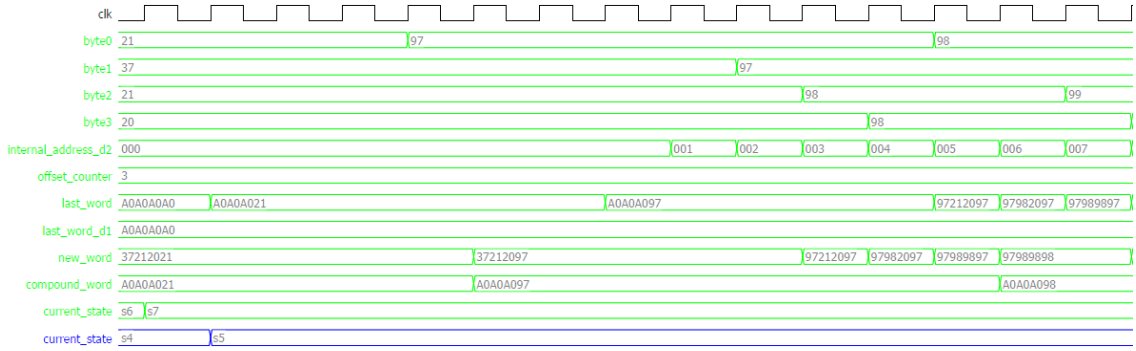


Imagen 5.64 – Generación de new_word y compund_word (offset_counter = 3)

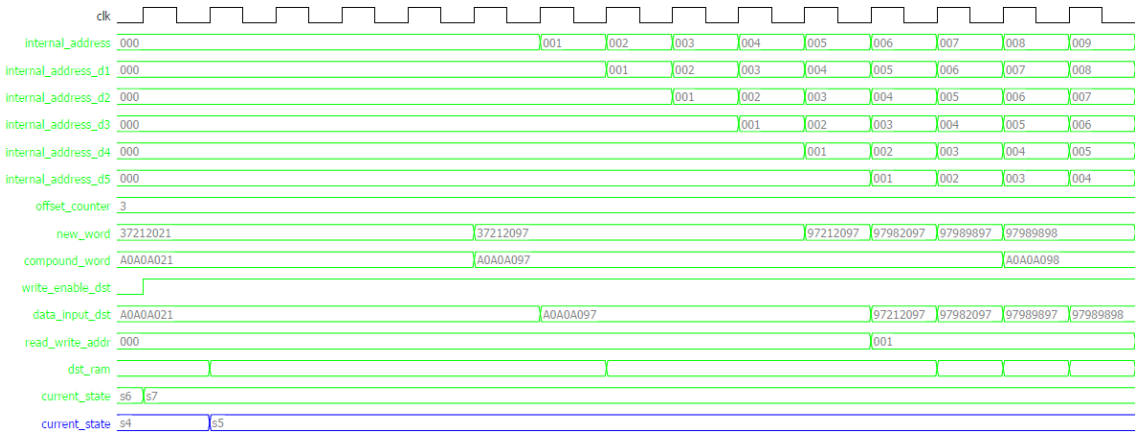


Imagen 5.65 – Almacenamiento de datos en dst_ram (offset_counter = 3)

5.7.9 Image_machine: S_8 (acceso al AHB bus)

Una vez cargada la fila de la imagen destino en **dst_ram**, se procede a la transferencia de esta a la memoria externa. Así pues, iniciamos un ciclo de escritura de datos por el AHB bus.

Al igual que para las lecturas de las filas even y odd, establecemos una transferencia con un número indeterminado de datos a enviar de 32 bits. La dirección de escritura se obtiene de **row_address_write** (líneas 839 a 848).

Dado que hemos optado por una transferencia de datos de 32 bits, nos vemos obligado a acceder a la memoria externa con direcciones múltiplos de 4. Es por ello que en **addro** se carga el valor de los 30 bits más significativos de **row_address_write** y los dos menos significativos son cero.

Tras determinar **addro**, configurado **dwrite** en modo escritura (nivel alto) y con **start** en nivel alto, hay una latencia de acceso al AHB bus, de modo que cuando esta finaliza y tengamos el control del AHB bus, se inicia la transferencia. Esto se determina mediante **transfer_active** (pasa a nivel alto) y **transfer_end** (debe estar en nivel bajo), momento en el que **image_machine** pasa de **S_8** a **S_9**. En la siguiente imagen podemos observar esta trama.

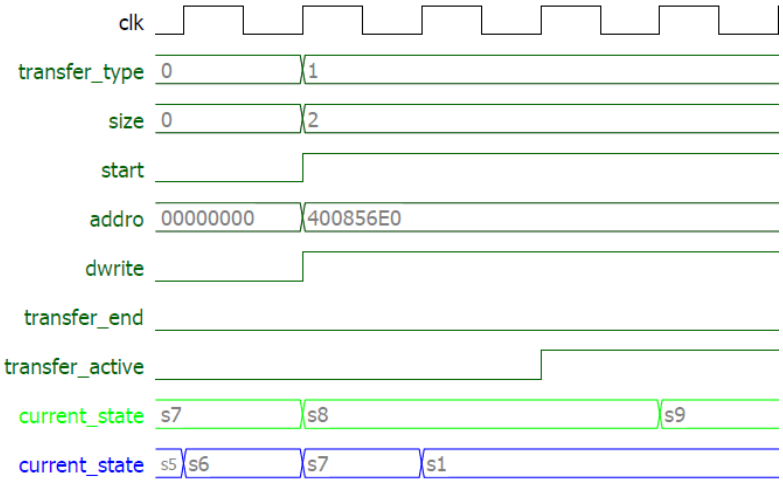


Imagen 5.66 – Latencia de inicio de transferencia (acceso al AHB bus) en S_8

5.7.10 Image_machine: S_9 (transferencia de fila destino)

Una vez obtenido el control del AHB bus, tenemos otra latencia de acceso a la memoria externa de aproximadamente 100 ns, tras la que comienza la transferencia.

Igual que ocurre en las transferencias para la lectura de las filas even y odd, en la escritura de las filas de la imagen destino también debemos tener en cuenta el tamaño de dichas filas. Así pues, si este es múltiplo de cuatro el número de datos a escribir es de:

$$dst\_width\_adjusted = \frac{dst\_width\_REG}{4}$$

En caso contrario, para las imágenes no múltiplos de cuatro con uno y dos pixeles libres se necesitan:

$$dst\_width\_adjusted = \frac{dst\_width\_REG}{4} + 1$$

Para las no múltiplos de cuatro con tres pixeles libres se pueden necesitar:

$$src\_width\_adjusted = \frac{src\_width\_REG}{4} + 1$$

o

$$src\_width\_adjusted = \frac{src\_width\_REG}{4} + 2$$

En el código se ha implementado la segunda ecuación (líneas 876 a 910).

Al "finalizar" la transferencia (líneas 780 a 785), **image_machine** pasa a **S_{10}** como se puede observar en la imagen 5.68.

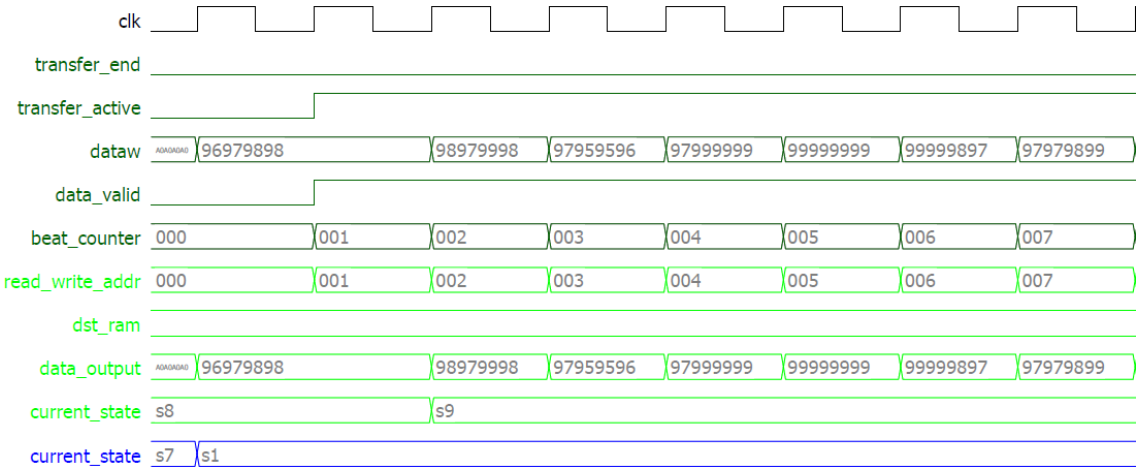


Imagen 5.67 – Inicio de la transferencia en S_9

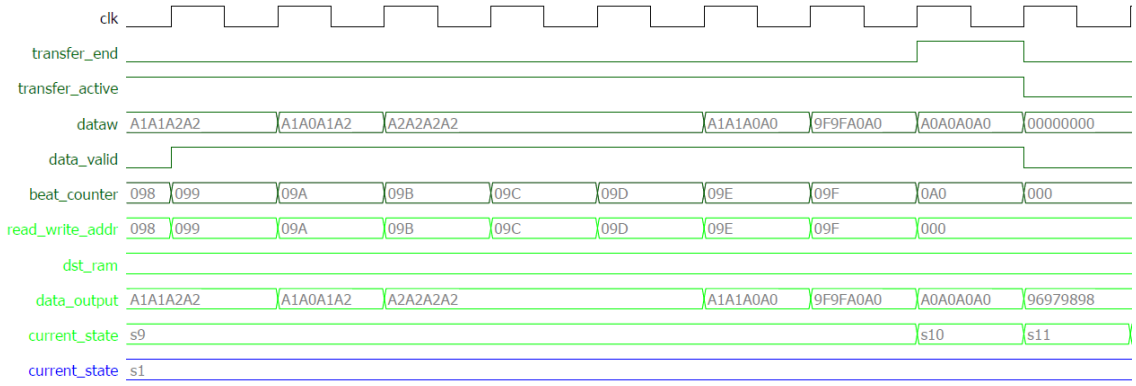


Imagen 5.68 – Finalización de la transferencia y transición de S_9 a S_{10}

5.7.11 Image_machine: S_{10} (finalización de transferencia de fila destino)

Como se puede ver en la condición de transición de **S_9** a **S_{10}**, esta termina antes del valor **dst_width_ajusted** puesto que debemos asegurarnos que el último valor que se manda se envía con la condición de **data_valid** como se puede observar en el código (líneas 786 a 971) y en la imagen 5.68. Tras esto, **image_machine** pasa a **S_{11}**.

5.7.12 Image_machine: S_{11} (actualización de filas procesadas)

Una vez concluida la transferencia, pasamos a incrementar (**inh**) el contador del número de filas procesadas **row_cnt** (en el siguiente ciclo de reloj se obtiene el nuevo valor de **embedd_yofs** para obtener la dirección de lectura de las nuevas filas even y odd), junto con otras dos señales ya explicadas anteriormente, **dst_offset** y **offset_counter**. En la siguiente imagen podemos ver esta trama.

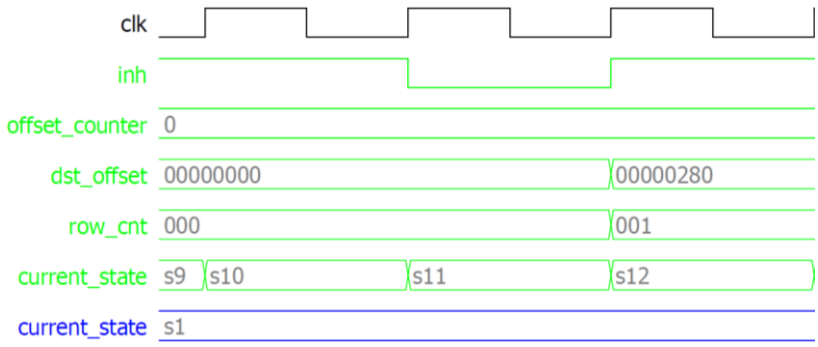


Imagen 5.69 – Incremento de contadores y actualización de acumuladores

5.7.13 Image_machine: S_{12} (comprobación de funcionamiento)

Tras haber realizado el escalado completo de una fila de la imagen destino e incrementado el contador **row_cnt** que controla el algoritmo, procedemos a comprobar (líneas 794 a 799) si la fila que se obtuvo era la final de la imagen destino. Si es así pasamos a $\mathbf{S_0}$ y se detiene el algoritmo, en caso contrario vamos a $\mathbf{S_{13}}$ para obtener las nuevas direcciones de lectura y escritura.

A continuación vemos la simulación de las dos posibles transiciones.

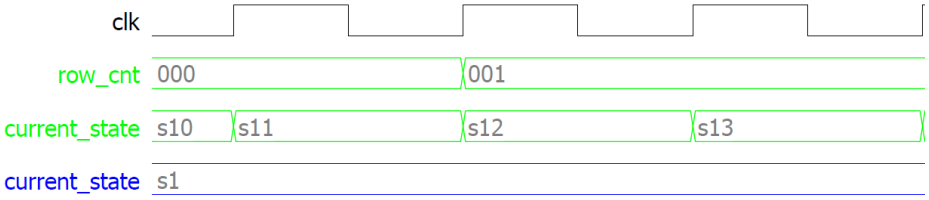


Imagen 5.70 – Transición de S_{12} a S_{13} de image_machine

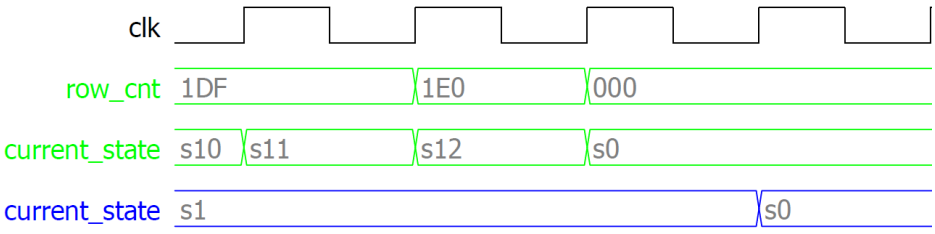


Imagen 5.71 – Transición de S_{12} a S_0 de image_machine

En la primera podemos ver como **image_machine** pasa a $\mathbf{S_{13}}$ y **row_machine** está en $\mathbf{S_1}$ mientras que en la segunda **image_machine** pasa a $\mathbf{S_{13}}$ produciéndose el reinicio de **row_cnt** y un ciclo de reloj después **row_machine** pasa a $\mathbf{S_0}$.

5.7.14 Image_machine: S_{13} a S_{15} (actualización de direcciones)

En **S_{13}** se obtiene el nuevo valor de **embedd_yofs** y la nueva dirección de escritura **row_address_write**, en **S_{14}** se obtiene el nuevo valor de **row_address_read** y en **S_{15}** los nuevos valores de **row_address_read_even** y **row_address_read_odd**, como se puede apreciar en la siguiente imagen.

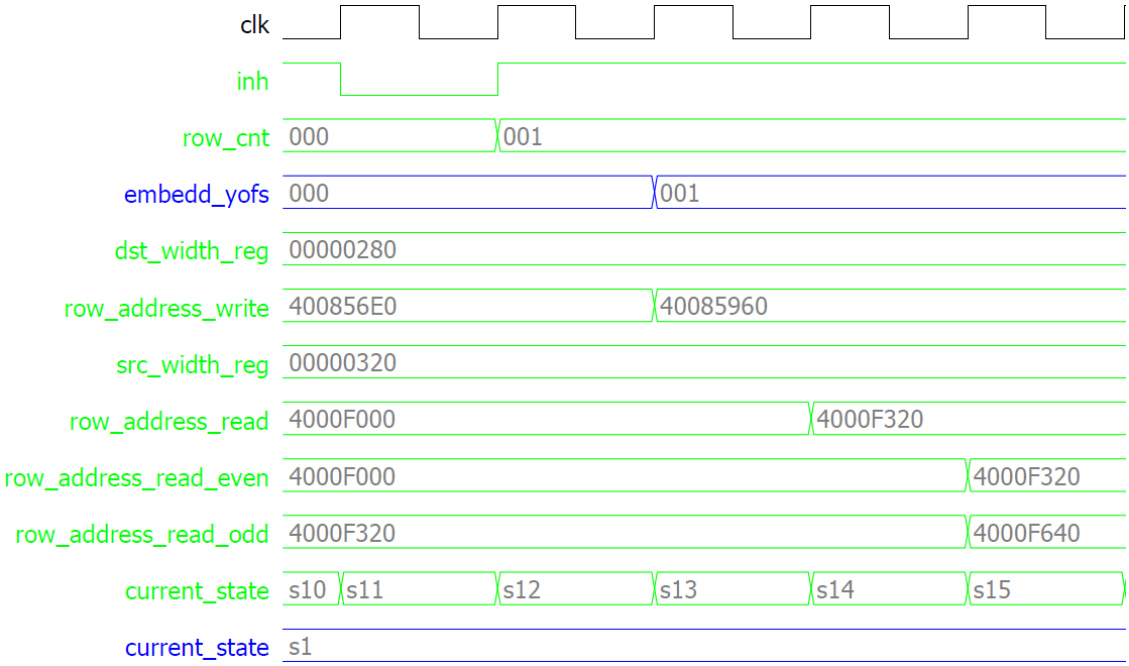


Imagen 5.72 – Actualización de direcciones

Tras esto, **image_machine** vuelve a **S_2** a partir de donde se repite todo el proceso hasta aquí descrito.

5.7.15 Row_machine: S_0 (reset)

Al igual que ocurre en **image_machine**, este es el estado en el que se encuentra **row_machine** cada vez que se inicializa el sistema o se genera un reset y cada vez que finaliza un ciclo de escalado. Es decir, todos los componentes y elementos están en reposo (reseteados), aunque esto no quita que puedan estar excitados (inicializados a algún valor concreto), como es el caso de **row_interpolation_state**, que vale cero, correspondiéndose con el estado en el que se encuentre **row_machine** para establecer la sincronización con **image_machine**. **Row_machine** se mantiene en **S_0** hasta que **image_machine** pase a **S_1** como se ha visto en el apartado 5.7.1.

5.7.16 Row_machine: S_1 (espera para fila even)

Una vez inicializada **row_machine**, se mantiene en este estado a la espera de que **image_machine** llegue a **S_4**, momento en el que ha finalizado la carga en **src_ram** de la fila even correspondiente de la imagen fuente. Al igual que en **S_0**, no se está realizando ninguna función.

5.7.17 Row_machine: S_2 (obtención de pseudo fila even)

En este punto da comienzo el núcleo del algoritmo de interpolación. Primero describimos gráficamente las conexiones de los distintos componentes implementados en

TFG_ROW_NTERPOLATION, para describir después su funcionamiento apoyándonos en imágenes procedentes de la simulación de estos.

A continuación se muestra el esquema de conexión de los componentes descritos en los apartados 5.6.7 a 5.6.10.

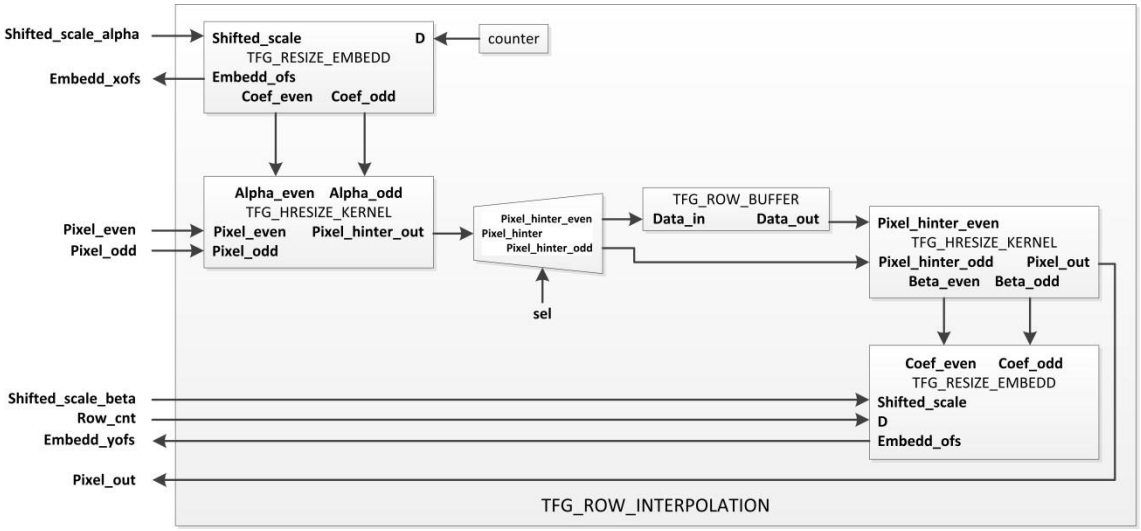


Imagen 5.73 – Pseudoflujo de datos de componentes de TFG_ROW_INTERPOLATION

Como podemos apreciar, se denota una arquitectura en serie de los distintos componentes presentes en TFG_ROW_INTERPOLATION. Esta fue la primera, y más sencilla de las soluciones que se propusieron para implementar el algoritmo de interpolación, ya que es la que menos dificultad supone en cuanto a problemas de sincronización entre los distintos componentes.

Durante $\mathbf{S_2}$ se están recibiendo los datos correspondientes a **píxel_even** y **píxel_odd** que se van solicitando a **image_machine** mediante **embedd_xofs**, el cual se obtiene a partir del contador **dx** que controla el algoritmo. Los datos obtenidos (pseudo fila even) en **píxel_hinter_out** se guardan en TFG_ROW_BUFFER, por lo que el demultiplexor controlado por **sel** apunta a su primera salida. En este estado no se usa TFG_VRESIZE_KERNEL, puesto que necesitamos los datos de la pseudo fila odd.

Como se ve a continuación, para conseguir acelerar el flujo de datos, se ha recurrido a la técnica del *pipeline*, de forma que todos los componentes funcionan al mismo tiempo.

La arquitectura en *pipeline* [49] consiste en ir transformando un flujo de datos en un proceso comprendido por varias fases secuenciales, siendo la entrada de cada una la salida de la anterior.

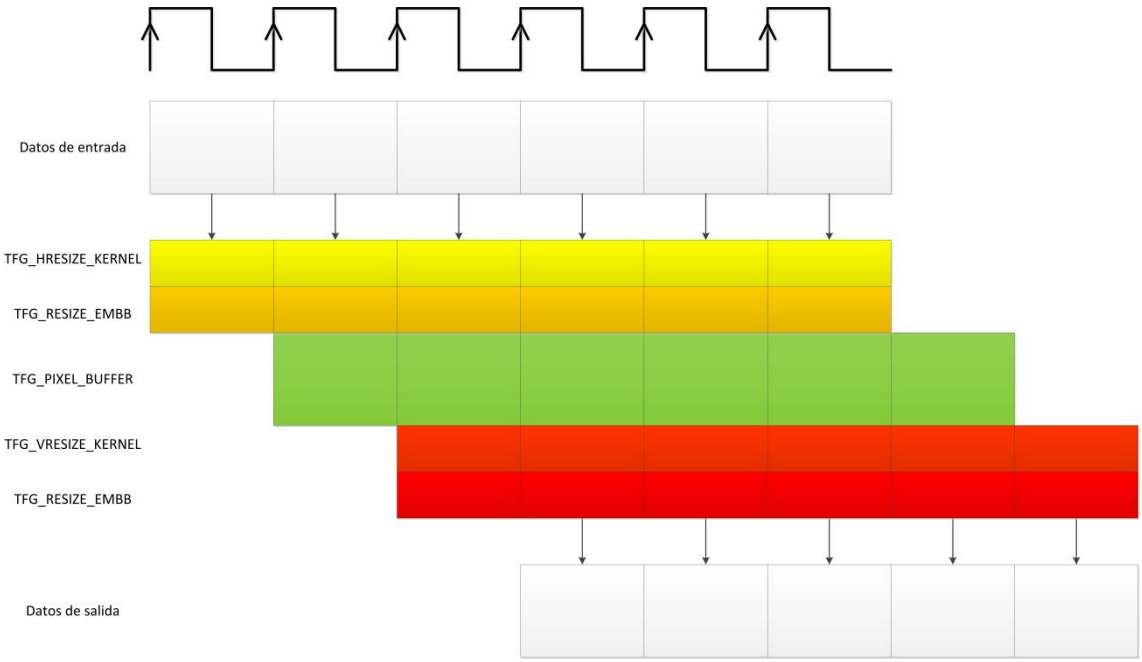


Imagen 5.74 – Pipeline de row_machine

En la siguiente imagen podemos ver la descripción detallada de la técnica del *pipeline* aplicada a los componentes de TFG_ROW_INTERPOLATION que intervienen en **S_2**.

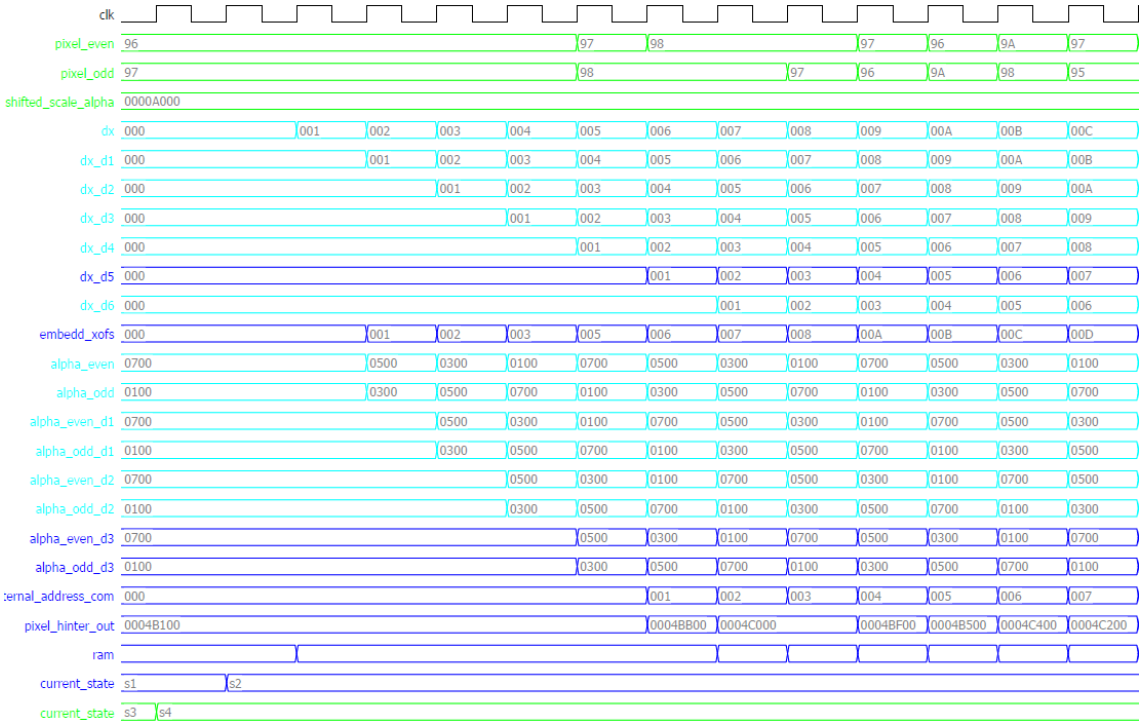


Imagen 5.75 – Pipeline en S_2

En azul tenemos las señales que se usan para obtener **píxel_hinter_out**, mientras que las de color cyan son retrasos (o las señales originales) que se necesitan para realizar la sincronización que requiere el empleo del *pipeline*.

Como vimos en el apartado 5.7.5, los primeros pixeles validos eran el 0x96 para **píxel_even** y 0x97 para **píxel_odd** (correspondientes a 0x003 de **embedd_xofs**). Dado que para obtener el primer **píxel_hinter_out** tenemos que tener los coeficientes correctos (**alpha_even_d3** y **alpha_odd_d3**) a la vez que llegan **píxel_even** y **píxel_odd**, nos vemos obligados a realizar tres retrasos sobre **alpha_even** y **alpha_odd**, ya que estos se obtienen un ciclo de reloj después de actualizar **dx**. Una vez calculado **píxel_hinter_out** procedemos a guardarlo en **ram**, y para indicar la dirección donde se guarda, **internal_address_com**, usamos el retraso **dx_d5**, que coincide con la obtención de **píxel_hinter_out**. Así pues, es **dx_d5** quien marca el final de $\mathbf{S_2}$ al obtenerse el último **píxel_hinter_out** de la pseudo fila even, como podemos ver en la siguiente imagen.

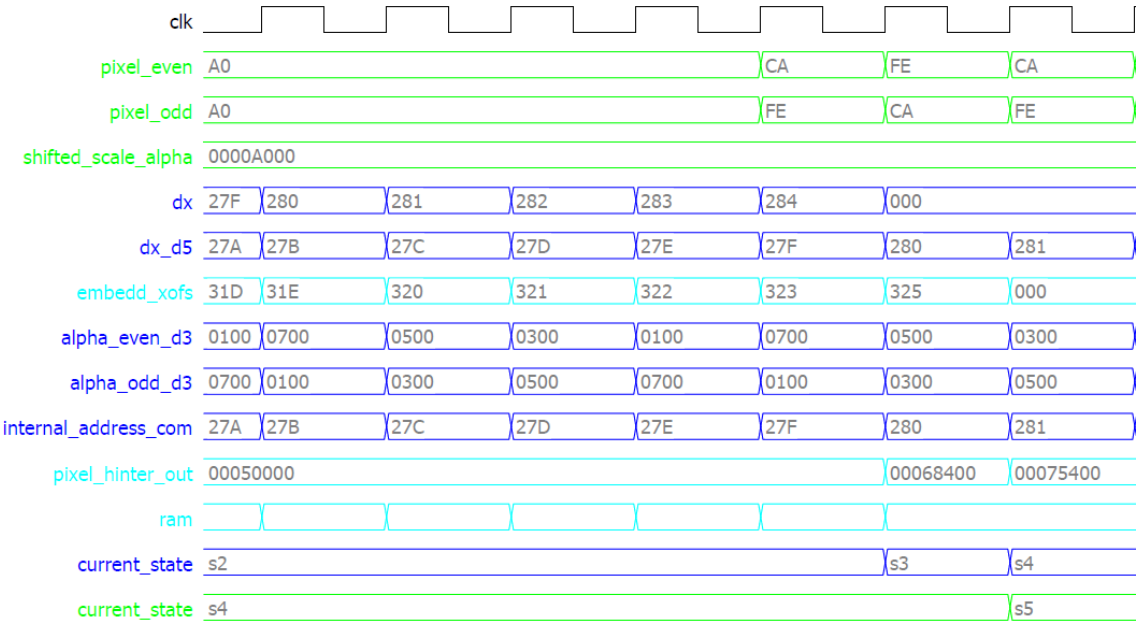


Imagen 5.76 – Finalización de la obtención de la pseudo fila even

Para distinguir el origen de cada una de las señales, se han dibujado en distintos colores. Así pues, las verdes corresponden a **image_machine** y las azules a **row_machine**. En cyan se han representado las correspondientes a los componentes de TFG_ROW_INTERPOLATION.

En este caso, estamos escalando desde una imagen de dimensiones 800x600 a 640x480. Así pues, el final de $\mathbf{S_2}$ se da cuando se detecta 640 – 1 = 639 (0x280 – 0x001 = 0x27F) en **dx_d5**.

5.7.18 Row_machine: S_3 (reset del contador)

Dado que aún nos queda el proceso de interpolación de la fila odd, necesitamos que **dx** comience de nuevo en cero. Para ello, dicho contador tiene implementado un reset asíncrono, como se puede ver en el código (líneas 229 a 238).

Con el siguiente ciclo de reloj **row_machine** pasa a $\mathbf{S_4}$ e **image_machine** a $\mathbf{S_5}$ al detectarse el estado $\mathbf{S_3}$ de **row_machine**, como se puede apreciar en la imagen 5.76.

5.7.19 Row_machine: S_4 (espera para fila odd)

Row_machine se mantiene en este estado hasta que **image_machine** llegua a $\mathbf{S_7}$ (carga de la fila odd en **src_ram**), y al igual que en $\mathbf{S_1}$, no se está realizando ninguna función.

5.7.20 Row_machine: S_5 (obtención de pseudo fila odd y fila destino)

Es aquí donde por fin se obtienen los pixeles de la fila de la imagen destino. Al igual que en el apartado 5.7.17, seguimos empleando la técnica del *pipeline*, y en este caso con todos los componentes representados en el esquema de la imagen 5.73. Así pues, se reciben los **píxel_even** y **píxel_odd** de la fila odd, y junto a los coeficientes de interpolación se obtienen los **píxel_hinter_out**, que en este caso se envían directamente a TFG_VRESIZE_KERNEL, por lo que la señal de control **sel** del demultiplexor se encontra en nivel alto para colocar su entrada en la segunda salida, mandando este dato a la entrada **píxel_hinter_odd**. La entrada **píxel_hinter_even** se obtiene de los datos guardados en TFG_ROW_BUFFER. Los datos obtenidos en **píxel_out** son enviados a **image_machine** para guardarlos en **dst_ram**. A continuación vemos en detalle esto para aclarar su funcionamiento.

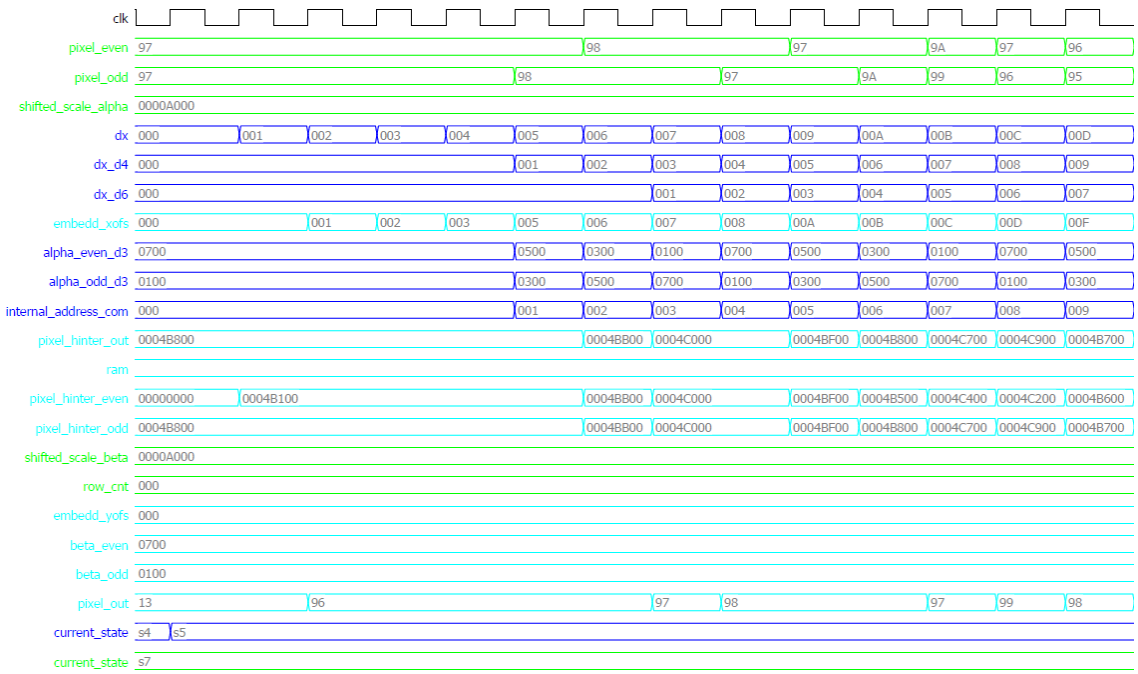


Imagen 5.77 – Pipeline S_5

Los primeros valores correctos de **píxel_even** y **píxel_odd** son 0x97 y 0x97 que corresponden con el 0x004 de **dx**. Dado que para obtener **píxel_out** necesitamos tener disponibles **píxel_hinter_even y píxel_hinter_odd**, el primero se obtiene de **ram** usando como **internal_address_com** el retraso **dx_d4**, el segundo se obtiene mediante el *pipeline*. Los coeficientes **beta_even** y **beta_odd** ya se encuentran calculados al momento de usarlos. Una vez obtenido el primer valor correcto de **píxel_out**, 0x96 correspondiente al 0x006 de **dx**, se pasa a **image_machine** junto con el valor que marque **internal_address_com**, para, como se vio en el apartado 5.7.8, poder juntar los pixeles en palabras de 32 bits para que estos sean guardados en la memoria externa.

Como en esta ocasión el *pipeline* dura un ciclo de reloj más que en el caso anterior, necesitamos otro retraso mas, **dx_d6,** que es el encargado de finalizar el proceso, como se puede ver en la siguiente imagen.

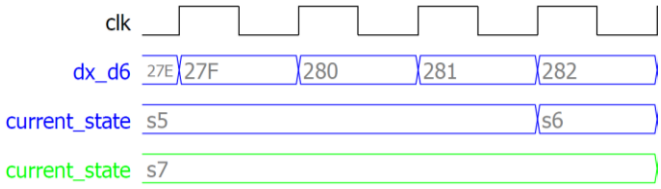


Imagen 5.78 – Transición de S_5 a S_6 de row_machine

Como se puede apreciar, no termina con el valor 0x27F (como terminaba en el estado **S_2**), sino con 0x281. Esto se debe a que se daban problemas de funcionamiento aleatorios (en la FPGA) debidos a las sentencias combinacionales de las señales implicadas en la generación de las palabras que se guardan en **dst_ram**, por lo que se necesitó secuencializar con el reloj dichas señales, lo que producía dos ciclos de retraso que fueron solventados de esa manera.

A continuación demostramos la eficiencia de la técnica del *pipeline* en la obtención de una fila de la imagen destino. Para ello nos valemos de simulaciones solo de **row_machine** realizadas con el simulador Isim en las primeras fases de desarrollo.

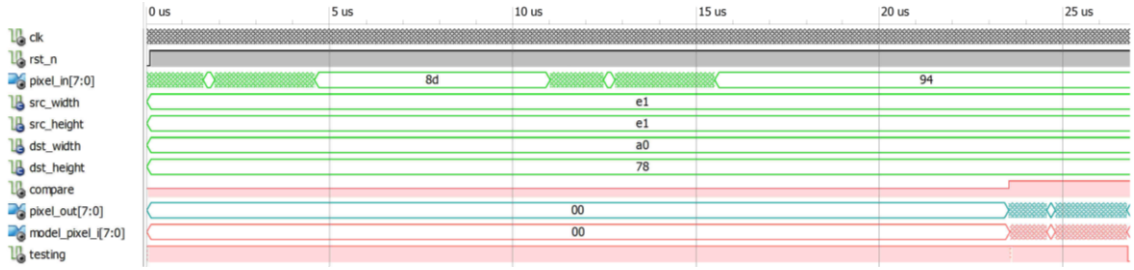


Imagen 5.79 – Obtención de fila sin pipeline en versión primitiva

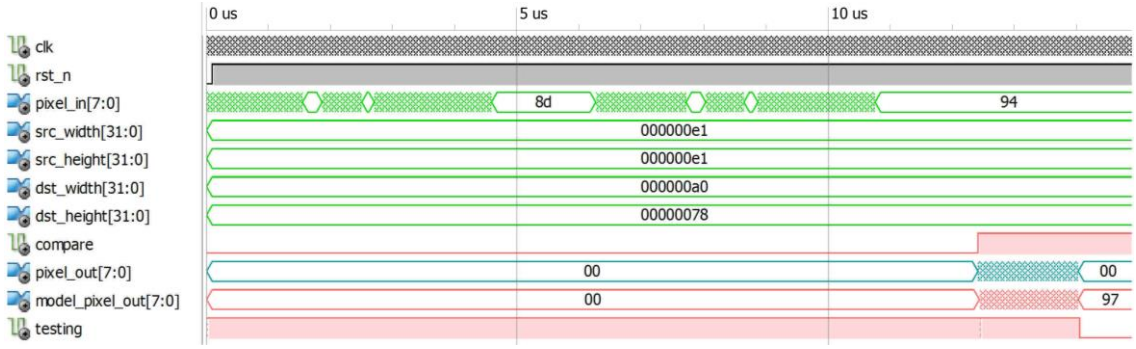


Imagen 5.80 – Obtención de fila con pipeline en versión primitiva

En estas imágenes se obtiene el escalado de una fila de una imagen de 225x225 a 160x140. Los datos píxel_in provienen de un fichero *.txt. Una vez se ha completado el escalado de dicha fila se activa la señal compare para pasar a comprobar si los valores obtenidos corresponden con los almacenados en otro fichero *.txt que contiene los valores que se obtienen con el software. Si ambos valores coinciden, testing estará en nivel alto.

Si nos referimos a términos temporales, sin el *pipeline* implementado se tarda aproximadamente 23.6 µs en obtener la fila, en cambio, con el *pipeline* implementado se tarda unos 12.4 µs. Así pues, obtenemos un rendimiento aproximadamente del 190%, es decir, se obtiene la fila utilizando *pipeline* casi en la mitad de tiempo que si no se empleara.

5.7.21 Row_machine: S_6 (sincronización)

Dependiendo de **offset_counter** (señal de **image_machine**, apartado 5.7.8) y del tamaño de la imagen (tipo de imagen), si realizamos el reset de **dx** nada más terminar $\mathbf{S_5}$, podemos tener un comportamiento no deseado, puesto que el *pipeline* implementado en $\mathbf{S_5}$ tiene retrasos impuestos por el algoritmo de generación de palabras, de forma que un reset prematuro en **dx** se propagaría hasta **internal_address_com**, pudiendo enviar un valor incorrecto a **image_machine**.

Con el siguiente ciclo de reloj, **row_machine** pasa a $\mathbf{S_7}$ e **image_machine** a $\mathbf{S_8}$.

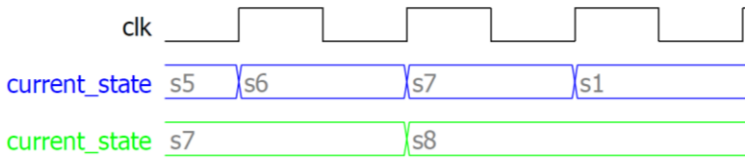


Imagen 5.81 – Transición de S_6 a S_7 de row_machine

5.7.22 Row_machine: S_7 (reset del contador)

Una vez finalizado el proceso de obtención de una fila de la imagen destino, debemos dejar **row_machine** lista (reset en **dx**) para comenzar de nuevo todo el proceso descrito. Así pues, un ciclo de reloj después de $\mathbf{S_7}$, **row_machine** pasa a $\mathbf{S_1}$ y se repete todo el proceso, como se puede apreciar en la imagen 5.81.

Por último, mostramos la simulación completa del escalado de una imagen desde 800x600 a 640x480 (se han escogido señales significativas para observar el inicio y fin del escalado).

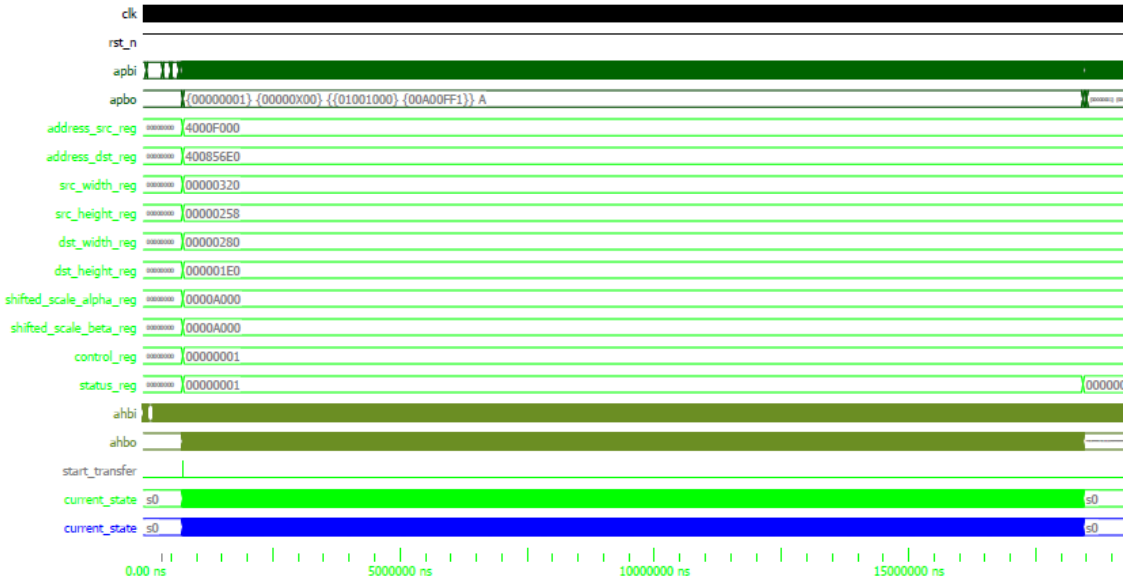


Imagen 5.82 – Escalado completo

Se obtiene un tiempo aproximado de escalado de 18000000 – 770000 = 17230000 ns = 17.23 ms. En tiempo real se obtienen unos 20 ms, debido a que la forma de medir en software no resulta tan exacta como en la simulación además de que al trabajar en tiempo real el AMBA bus se encuentra compartido por otros dispositivos, por lo que se pueden ralentizar las transferencias de datos.

6. Resultados

En este capítulo se describen, en primer lugar, los resultados obtenidos tras la síntesis (consumo de recursos y potencia) del proyecto de ISE, siguiendo el orden inverso en el que se han descrito los componentes del módulo IP en el Capítulo 5. El otro aspecto de este capítulo hace referencia a los resultados del funcionamiento en tiempo real, de forma que se han realizado medidas comparativas entre la ejecución en software y en hardware.

6.1 Resultados de síntesis

Para llevar a cabo esta tarea hemos hecho uso de la herramienta de Xilinx ISE Design Suite y pretendemos con ello determinar el consumo de recursos con intención de seleccionar los dispositivos FPGA que permitan la implementación del módulo IP, optando en nuestro caso por la *Virtex 5 FPGA Evaluation Platform ML505*, ya que sabemos de antemano que el procesador LEON3 que controla el módulo IP puede ser implementado en ella y dispone de recursos suficientes para incluir otros desarrollos.

La creación del proyecto para ISE, una vez configurado el procesador mediante la herramienta gráfica *LEON3MP Design Configuration* (comando *make xconfig*), se guía por los siguientes comandos que introducimos en el terminal de MinGW: *make* distclean (en caso de que se reconfigure el proyecto para borrar archivos) y *make scripts* (genera scripts para arrancar y compilar el proyecto). Dado que en nuestro caso tenemos implementado el controlador MIG DDR2 de Xilinx, se necesitan dos comandos más antes de *make scripts*: *make mig* (genera el controlador) y *make install-unisim* (para las librerías del controlador). Una vez creado el proyecto, podemos arrancarlo mediante el comando *make ise-launch*, como podemos ver en la siguiente imagen.

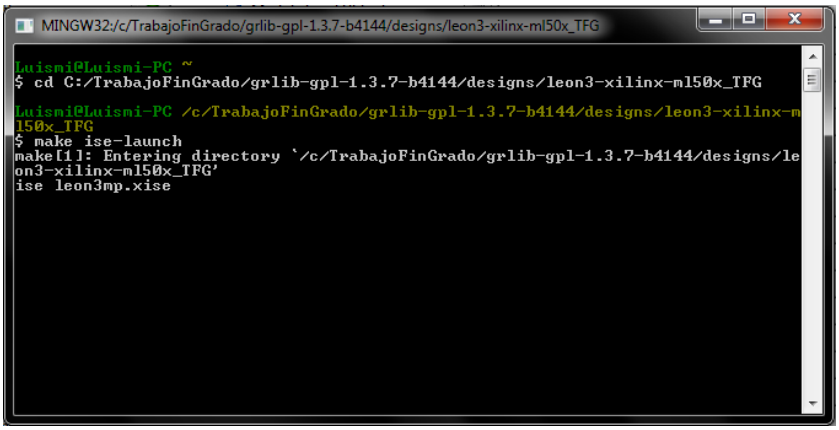


Imagen 6.1 – Terminal MinGW

Y en la siguiente imagen podemos ver la configuración del proyecto de ISE para nuestra FPGA.

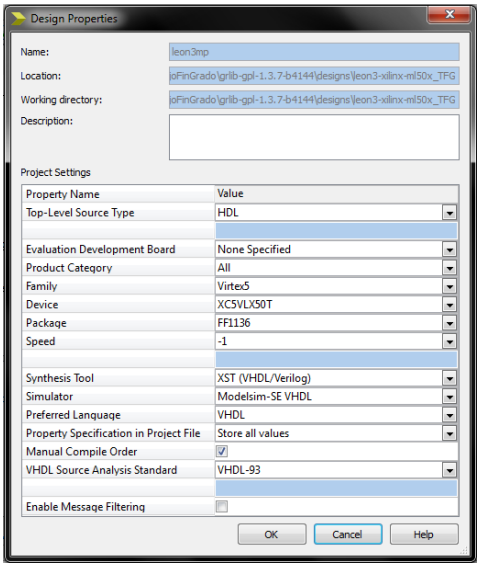


Imagen 6.2 – Configuración del proyecto en ISE Design Suite

El desarrollo de los siguientes apartados sigue la cronología inversa de los apartados 5.6.1 a 5.6.10, con la intención de ascender en cuanto a consumo de recursos. En cada uno presentamos los resultados de *Synthesis Report* y *Module Level Utilization*. Del primero obtenemos información acerca de las unidades funcionales (registros, sumadores, memorias, etc.) que se han detectado durante la síntesis. El segundo nos determina el consumo de recursos de la FPGA. Para el último apartado, correspondiente al módulo superior, nos valemos también de *Device Utilization Summary*, por el que nos regimos para la selección de FPGAs. Vemos aquí también el consumo de potencia del módulo IP y del procesador en general.

Dado que el módulo IP es parametrizable, debemos tener en cuenta los valores con los que se ha procedido a la síntesis, que se pueden obtener de la instanciación de este.

```
TFG_IMAGE_SCALING_1 : TFG_IMAGE_SCALING

        generic map (FABTECH            => fabtech,

                     MEMTECH            => memtech,

                     PINDEX             => 10,

                     PADDR              => 10,

                     PMASK              => 16#0ff#,

                     HINDEX             => (NCPU+CFG_AHB_UART+CFG_GRETH+CFG_AHB_JTAG+
                                            CFG_SVGA_ENABLE),

                     --AHBACCSZ         =>

                     --BURSTLEN         =>

                     AHB_SLV_HINDEX     => 7,

                     AHB_SLV_HADDR      => 16#a00#,

                     --AHB_SLV_HMASK    =>
```

```
                    HIRQ              => 10,
                    INT_ADDRESS_WIDTH => 11,
                    SRC_ADDRESS_WIDTH => 9,
                    DST_ADDRESS_WIDTH => 9,
                    ADDRESS_HEIGHT    => 11)
        port map (Rst_n => rstn,
                  Clk   => clkm,
                  Apbi => apbi,
                  Apbo => apbo(10),
                  Ahbi => ahbmi,
                  Ahbo =>  ahbmo(NCPU+CFG_AHB_UART+CFG_GRETH+CFG_AHB_JTAG+
                           CFG_SVGA_ENABLE));
```

Los cuatro últimos genéricos están configurados para trabajar con imágenes de entrada de tamaño máximo 2048x2048 (2^{11} = 2048) e imágenes de salida de cualquier tamaño menor que el de entrada.

6.1.1 TFG_ROW_BUFFER

La síntesis de este componente nos ha reportado las siguientes unidades funcionales.

Functional unit	Size	Quantity
RAMs	2048x32 bit single port	1
Registers	32 bit	1

Tabla 6.1 – Unidades funcionales de TFG_ROW_BUFFER

A continuación mostramos los resultados de *Module Level Utilization* para este componente.

Slice Logic Utilization	Used
Registers	32
LUTs	1097
LUTRAM	1024

Tabla 6.2 – Consumo de recursos de TFG_ROW_BUFFER

6.1.2 TFG_VRESIZE_KERNEL

La síntesis de este componente nos ha reportado las siguientes unidades funcionales.

Functional unit	Size	Quantity
Multipliers	32x16 bit	2
Adders/Subtractors	48 bit adder	2
Registers	8 bit	1
Comparators	10 bit comparator greater	1

Tabla 6.3 – Unidades funcionales de TFG_VRESIZE_KERNEL

A continuación mostramos los resultados de *Module Level Utilization* para este componente.

Slice Logic Utilization	Used
Registers	8
LUTs	43
DSP48E	4

Tabla 6.4 – Consumo de recursos de TFG_VRESIZE_KERNEL

6.1.3 TFG_HRESIZE_KERNEL

La síntesis de este componente nos ha reportado las siguientes unidades funcionales.

Functional unit	Size	Quantity
Multipliers	16x8 bit	2
Adders/Subtractors	24 bit adder	1
Registers	32 bit	1

Tabla 6.5 – Unidades funcionales de TFG_HRESIZE_KERNEL

A continuación mostramos los resultados de *Module Level Utilization* para este componente.

Slice Logic Utilization	Used
Registers	24
DSP48E	2

Tabla 6.6 – Consumo de recursos de TFG_HRESIZE_KERNEL

6.1.4 TFG_RESIZE_EMBB

La síntesis de este componente nos ha reportado las siguientes unidades funcionales.

Functional unit	Size	Quantity
Multipliers	32x12 bit	1
Adders/Subtractors	12 bit adder	1
	32 bit adder	1
	44 bit subtractor	1
Registers	11 bit	1
	16 bit	2
Comparators	32 bit comparator greater	2

Tabla 6.7 – Unidades funcionales de TFG_RESIZE_EMBB

A continuación mostramos los resultados de *Module Level Utilization* para este componente.

Slice Logic Utilization	Used
Registers	34
LUTs	79
DSP48E	2

Tabla 6.8 – Consumo de recursos de TFG_RESIZE_EMBB

De este componente se tienen dos instanciaciones, por lo que a la hora de contabilizar los recursos totales, tenemos que multiplicar estos por dos.

6.1.5 TFG_ROW_INTERPOLATION

La síntesis de este componente nos ha reportado las siguientes unidades funcionales.

Functional unit	Size	Quantity
FSMs	8 states one hot encoding	1
Adders/Subtractors	32 bit adder	1
	32 bit subtractor	1
Counters	11 bit up counter	1
Registers	8 bit	1
	11 bit	6
	16 bit	6
Comparators	32 bit comparator greatequal	2

Tabla 6.9 – Unidades funcionales de TFG_ROW_INTERPOLATION (sin contabilizar componentes)

A continuación mostramos los resultados de *Module Level Utilization* para este componente.

Slice Logic Utilization	Used	Total
Registers	135	267
LUTs	139	1437
LUTRAM	11	1035
DSP48E	0	10

Tabla 6.10 – Consumo de recursos de TFG_ROW_INTERPOLATION

En la siguiente gráfica mostramos el consumo de recursos hasta aquí expuesto.

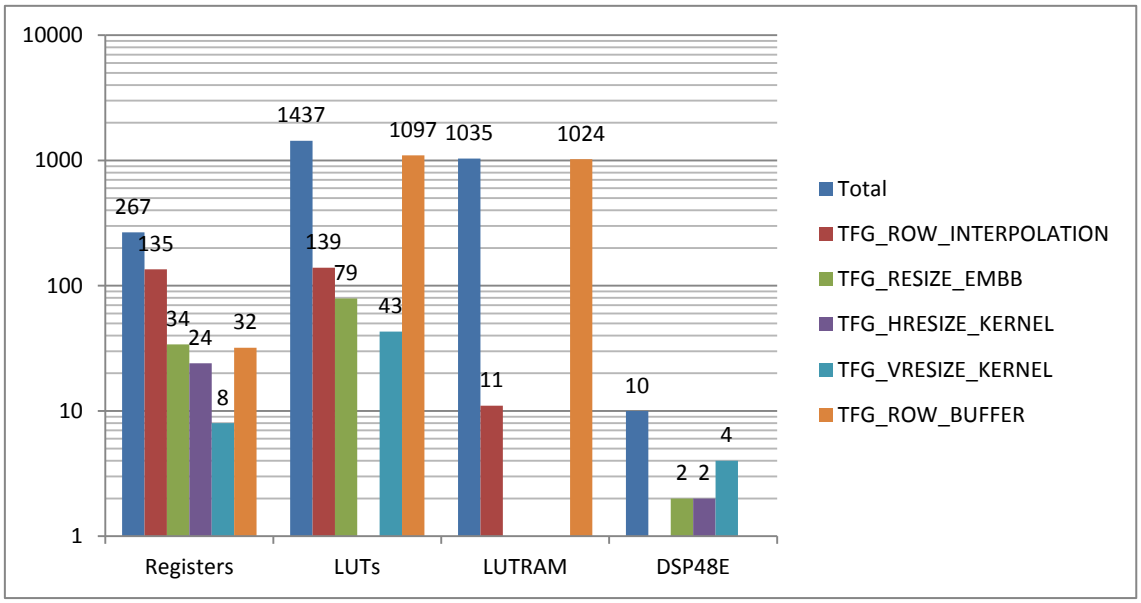


Imagen 6.3 – Gráfica de consumo de recursos de TFG_ROW_INTERPOLATION

Nota: para contabilizar el total se ha de multiplicar por dos los recursos de TFG_RESIZE_EMBB.

6.1.6 TFG_APB_SLAVE

La síntesis de este componente nos ha reportado las siguientes unidades funcionales.

Functional unit	Size	Quantity
Registers	1 bit	1

Tabla 6.11 – Unidades funcionales de TFG_APB_SLAVE

Dado el escaso consumo de recursos, no se obtienen resultados en *Module Level Utilization*.

6.1.7 TFG_AHB_MASTER

La síntesis de este componente nos ha reportado las siguientes unidades funcionales.

Functional unit	Size	Quantity
ROMs	8x10 bit	1
Adders/Subtractors	10 bit adder	1
	32 bit adder	1
Registers	1 bit	6
	10 bit	2
	32 bit	1
Comparators	10 bit comparator equal	1

Tabla 6.12 – Unidades funcionales de TFG_AHB_MASTER

A continuación mostramos los resultados de *Module Level Utilization* para este componente.

Slice Logic Utilization	Used
Registers	48
LUTs	154

Tabla 6.13 – Consumo de recursos de TFG_AHB_MASTER

6.1.8 TFG_IMAGE_SCALING

La síntesis de este componente nos ha reportado las siguientes unidades funcionales.

Functional unit	Size	Quantity
FSMs	16 states one hot encoding	1
RAMs	512x32 bit single port	1
	512x32 bit dual port	2
Multipliers	16x16 bit	1
Adders/Subtractors	9 bit adder	1
	10 bit adder	2
	11 bit adder	1
	32 bit adder	3
	10 bit subtractor	1
Counters	11 bit up counter	1

Acumulators	2 bit up 141ccumulator 32 bit up acumulator	1 1
Registers	1 bit 2 bit 8 bit 10 bit 11 bit 16 bit 32 bit	1 3 6 3 6 1 21
Comparators	10 bit comparator equal 11 bit comparator greatequal	2 1

Tabla 6.14 – Unidades funcionales de TFG_IMAGE_SCALING (sin contabilizar componentes)

A continuación mostramos los resultados de *Module Level Utilization* para este componente.

Slice Logic Utilization	Used	Total
Registers	757	1072
LUTs	515	2106
LUTRAM	0	1035
BRAM/FIFO	2	2
DSP48E	1	11

Tabla 6.15 – Consumo de recursos de TFG_IMAGE_SCALING

En la siguiente gráfica podemos ver el consumo de recursos de los componentes de TFG_IMAGE_SCALING respecto de su total.

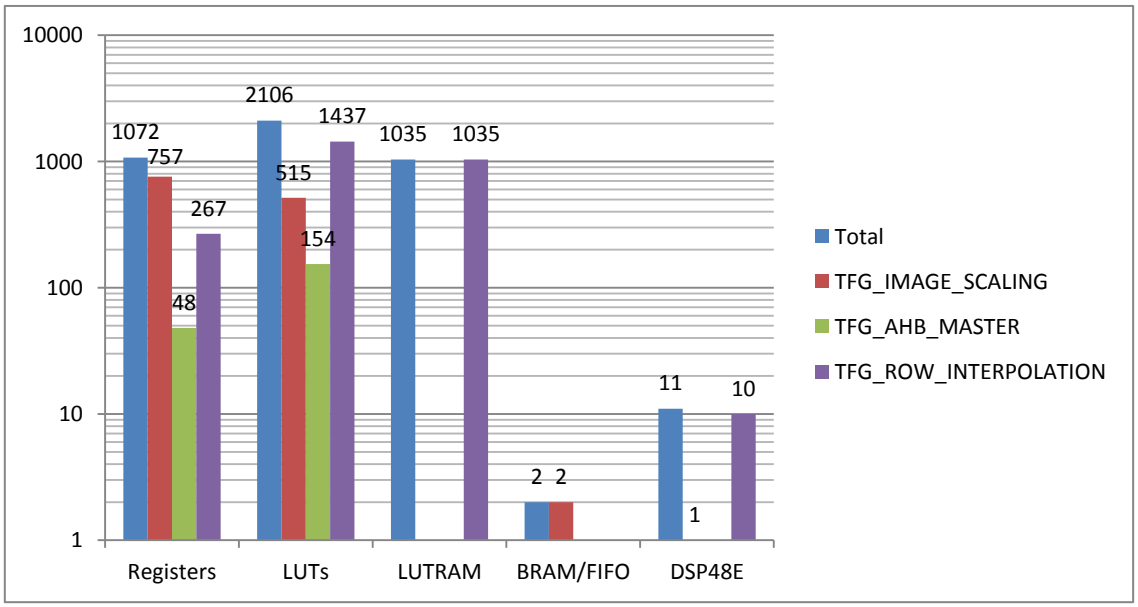


Imagen 6.4 – Gráfica de consumo de recursos de TFG_IMAGE_SCALING

6.1.9 leon3mp

Dado que este es el modulo superior del proyecto, y contiene la instanciación de todos los módulos IP, no resulta de interés (además de desmesurado) los datos de *Synthesis Report*, por lo que mostraremos solo el consumo de recursos. Por el contrario, si resulta de mucho interés la frecuencia máxima de trabajo, resultando esta de 90.065 MHz.

A continuación mostramos los resultados de *Module Level Utilization* para este componente.

Slice Logic Utilization	Total
Registers	10886
LUTs	18965
LUTRAM	1143
BRAM/FIFO	29
DSP48E	15
BUFG	15
DCM_ADV	4
PLL_ADV	1

Tabla 6.16 – Consumo de recursos de leon3mp (contabilizando componentes)

Y por último los resultados finales de la síntesis, procedentes de *Device Utilization Summary*.

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	10886	28800	37%
Number of Slice LUTs	18972	28800	65%
Number used as logic	17794	28800	61%
Number used as Memory	1143	7680	14%
Number used as Dual Port RAM	40		
Number used as Single Port RAM	1024		
Number used as Shift Register	79		
Number used as exclusive route-thru	35		
Number of route-thrus	440		
Number of occupied Slices	6506	7200	90%
Number of LUT Flip Flop pairs used	21431		
Number with an unused Flip Flop	10545	21431	49%
Number with an unused LUT	2459	21431	11%
Number of fully used LUT-FF pairs	8427	21431	39%
Number of unique control sets	1036		
Number of slice register sites lost to control set instructions	2360	28800	8%
Number of bonded IOBs	308	480	64%
Number of LOCed IOBs	308	308	100%
IOB Flip Flops	534		
Number of BlockRAM/FIFO	29	60	48%
Number of BlockRAM only	26		
Number of FIFO only	3		
Number of 36k BlockRAM used	7		
Number of 18k BlockRAM used	37		
Number of 36k FIFO used	3		
Total memory used (KB)	1026	2160	47%
Number of BUFG/NUFGCTRLs	15	32	46%
Number used as BUFGs	15		
Number of IDELAYCTRLs	3	16	18%

Number of BSCANs	2	4	50%
Number of BUFIOs	8	56	14%
Number of DCM_ADVs	4	12	33%
Number of DSP48E	15	48	31%
Number of PLL_ADVs	1	6	16%
Number of SYSMONs	1	1	100%
Average Fanout of Non-Clock Nets	4.38		

Tabla 6.17 – Consumo de recursos del procesador completo para la selección de FPGA

En las siguientes gráficas podemos ver el consumo de recursos de TFG_IMAGE_SCALING y leon3mp respecto del total disponible de la FPGA.

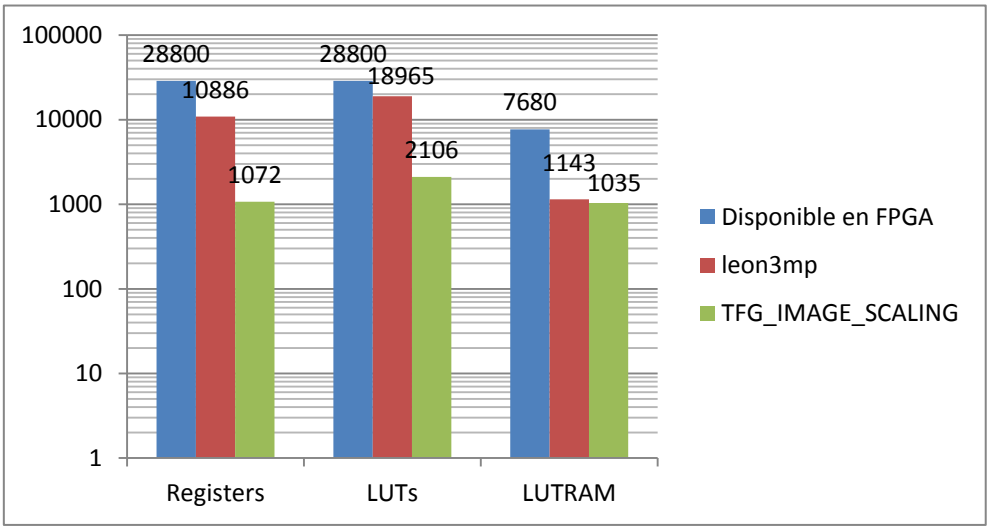


Imagen 6.5 – Gráfica de consumo de recursos de leon3mp 1

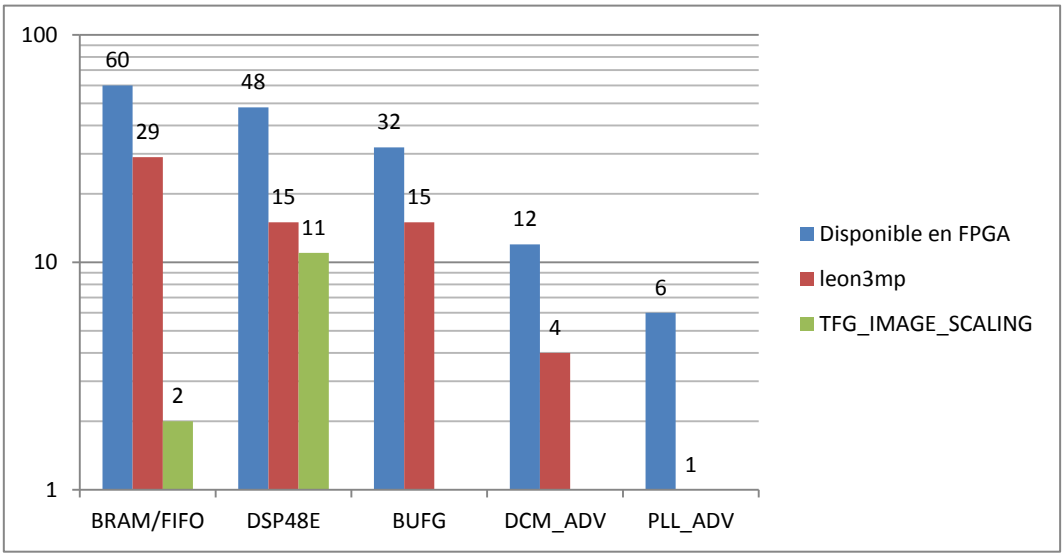


Imagen 6.6 – Gráfica de consumo de recursos de leon3mp 2

Por ultimo mostramos el consumo de potencia del procesador y del módulo IP.

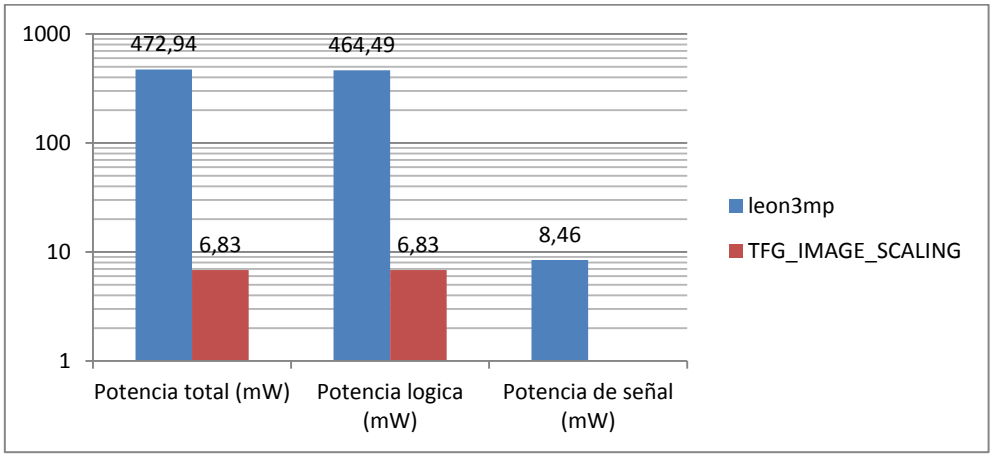


Imagen 6.7 – Gráfica de consumo de potencia

Dado el reducido consumo de potencia de TFG_IMAGE_SCALING, *XPower* Analyzer no proporciona información acerca del consumo de potencia de los componentes internos.

6.2 Resultados en tiempo real

Para llevar a cabo este proceso procedemos a la programación de la FPGA con el fichero leon3mp.bit generado por ISE Design Suite. La programación se realiza con el programa ISE iMPACT, siguiendo unos sencillos pasos. Una vez arrancado, hacemos click en *Boundary Scan* y a continuación en el icono *Initialize Chain* (conectar con la interfaz JTAG). Haciendo click derecho sobre el icono que corresponde al núcleo de la FPGA, seleccionamos *Assign New Configuration File* y buscamos leon3mp.bit. Por último, hacemos click derecho nuevamente y clicamos en *Program*.

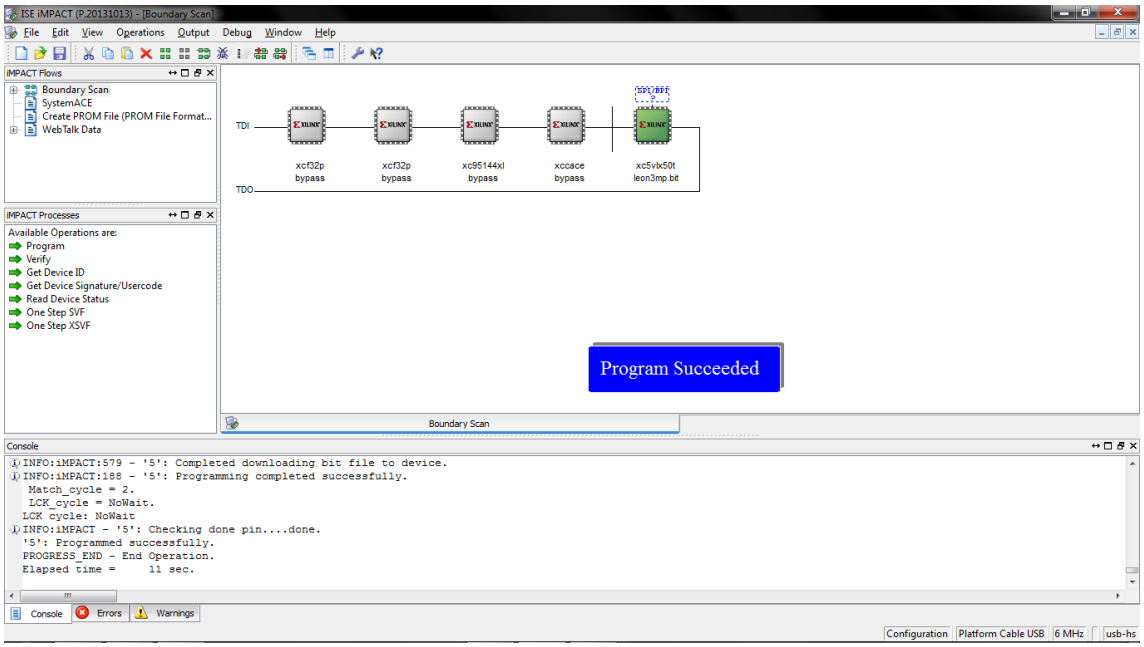


Imagen 6.8 – ISE iMPACT

Una vez programada la FPGA, abrimos un terminal de MinGW para arrancar el depurador GRMON del procesador LEON3 con el comando *grmon.exe -xilusb -u*. Tras esto nos aparece

información del procesador como la frecuencia (60 MHz, esta ha de ser menor que la frecuencia máxima obtenida en el *Synthesis Report*) y los componentes implementados. El módulo IP desarrollado en este trabajo aparece con la designación *Unknow device* y *Unknow vendor* para el AHB master y *Unknow device* y *Aeroflex Gaisler* para el APB slave. Dado que para observar el funcionamiento del módulo IP necesitamos una interfaz gráfica, debemos inicializar la interfaz DVI (*SVGA frame buffer*) mediante el comando *i2c dvi init_ml50x_vga*. A continuación cargamos el fichero que se genera tras la compilación del programa en C mediante el comando *lo resultados* y arrancamos el procesador con *run*.

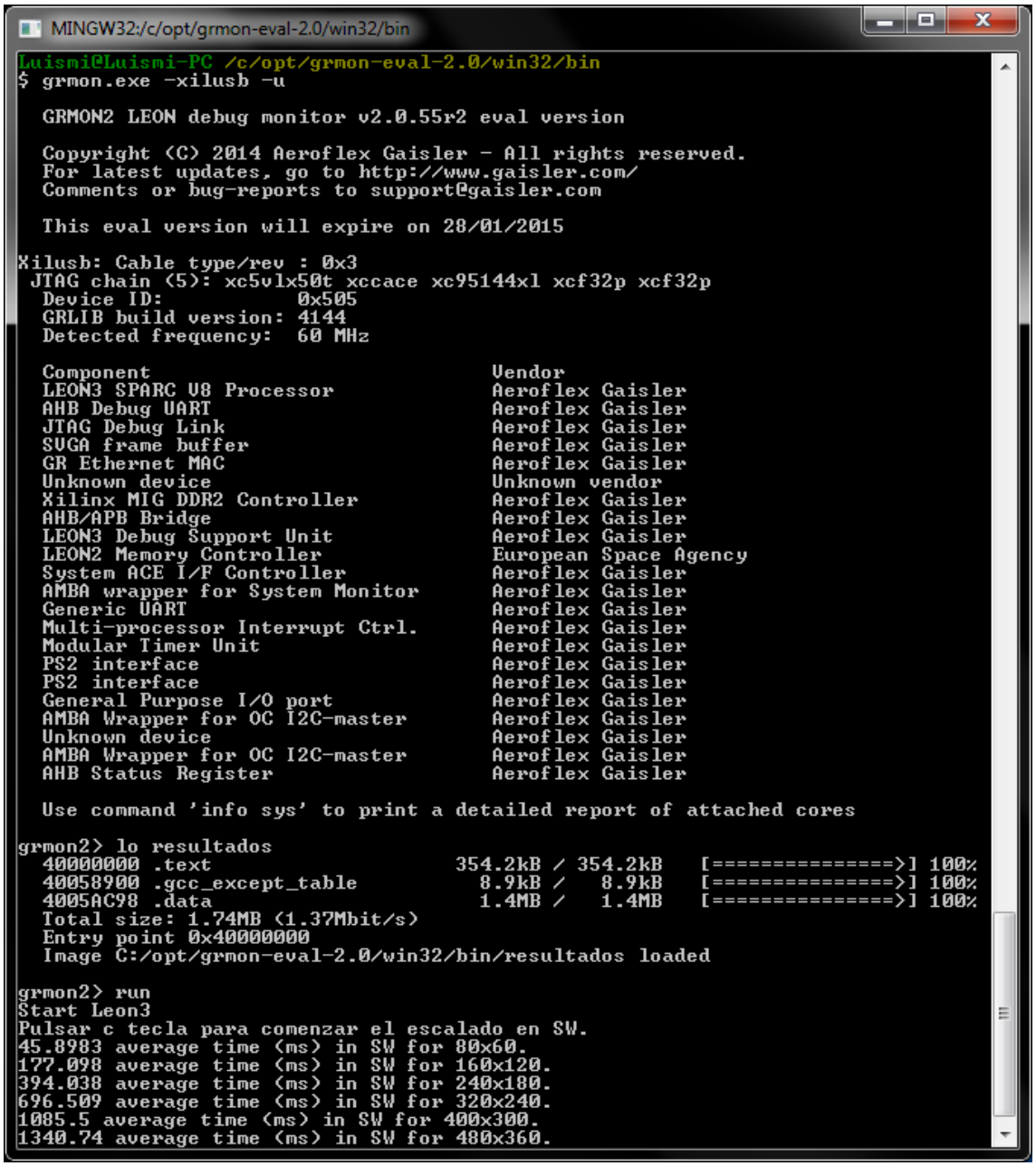


Imagen 6.9 – Depuración LEON3

Para la medición del tiempo de las funciones de escalado en software y hardware se ha empleado la función *clock ()*, que retorna el tiempo usado por el procesador entre los intervalos en los que se realiza la llamada a dicha función. La precisión de esta función es

reducida y dependiente del sistema, por lo que se realizan las mediciones un número determinado de veces y se calcula el promedio de los valores obtenidos. Así pues, para las pruebas que se presentan en los siguientes apartados, se repite cada escalado 100 veces.

El estudio se ha dividido en tres partes principales. En la primera de ellas se han realizado dos sencillas pruebas para comparar los tiempos de procesado entre software y hardware. Para la primera prueba se parte de una imagen fuente fija y se van realizando sucesivas variaciones de escala. La segunda prueba es al contrario, se fijan las dimensiones de la imagen destino y es la imagen fuente la que se va variando.

En la segunda parte del estudio se ha entrado más en profundidad en los tiempos de escalado mediante la realización de otras dos pruebas, justificando principalmente el comportamiento del algoritmo en software. Así pues, dichas pruebas han consistido en fijar una de las dimensiones de la imagen destino y se irá variando la otra.

La tercera parte ha estado centrada en el algoritmo hardware, con el objetivo de determinar una expresión que calcule el tiempo de escalado en función de las dimensiones de la imagen fuente y destino.

En el Anexo A.3 podemos ver el código empleado en los siguientes apartados (se realizan pequeñas modificaciones entre prueba y prueba).

Volvemos a recordar aquí que el algoritmo no realiza la ampliación de escala, ya que no se requería esta funcionalidad para la aplicación a que está destinado e introducía mayor complejidad en el desarrollo del código, por lo que no se realizan pruebas para determinar los tiempos en la ampliación de escala.

6.2.1 Comparativa entre el software y el hardware

En este apartado nos centramos principalmente en comparar los tiempos entre el software y el hardware. En la primera prueba partimos de una imagen fuente de 800x600 y vamos realizando escalados con incrementos proporcionales en las dimensiones de la imagen destino desde 80x60 pixeles hasta llegar al máximo (dimensiones de la imagen fuente). Los resultados obtenidos se pueden ver en la tabla 6.18 y en la imagen 6.10.

Imagen destino	Tiempo (ms) en software	Tiempo (ms) en hardware	Rendimiento
80x60	45.898	1.058	43.381
160x120	177.098	2.493	71.038
240x180	394.038	4.297	91.7
320x240	696.509	6.483	107.436
400x300	1085.5	9.058	119.838
480x360	1340.74	11.964	112.064
560x420	1611.85	15.281	105.48
640x480	1900.24	18.994	100.044
720x540	2192.57	23.014	95.271
800x600	2501.2	27.442	91.144

Tabla 6.18 – Tiempos en escalado con dimensiones de la imagen fuente fijas

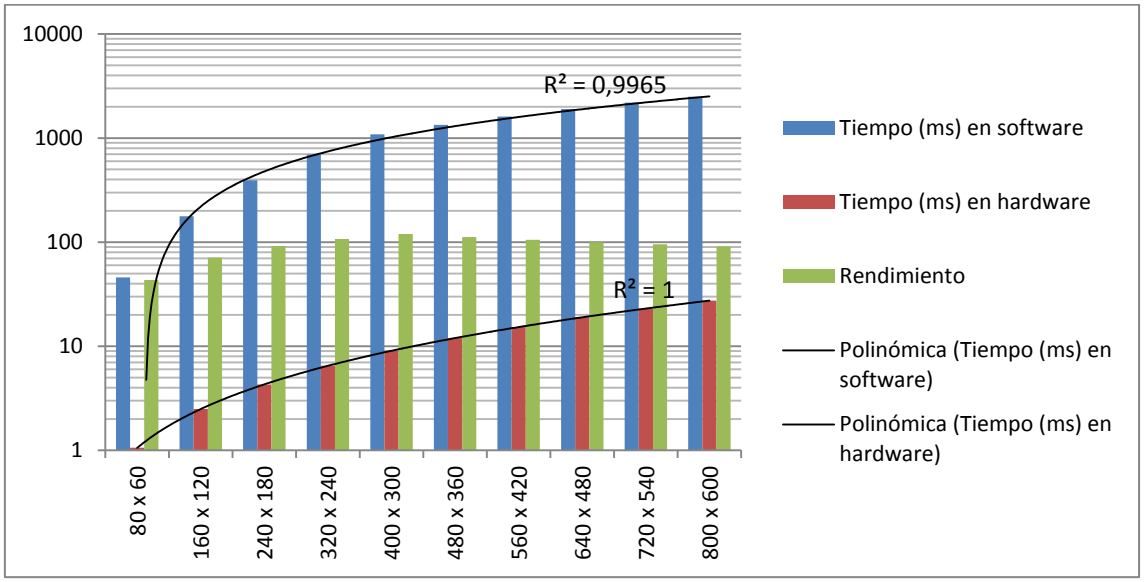


Imagen 6.10 – Gráfica de tiempos en escalado con dimensiones de la imagen fuente fijas

Como podemos apreciar, la principal conclusión a la que llegamos al observar las rectas de regresión de las gráficas de tiempos, es que el comportamiento es aproximadamente cuadrático, dado que se modifican las dos dimensiones de la imagen destino, y por tanto, fuertemente dependiente de los valores de esta. También podemos observar que la respuesta del algoritmo en hardware es más aproximada a la cuadrática (R = 1) que en software. Dichas rectas de regresión son solo orientativas, como se verá en el siguiente apartado.

Si ahora nos fijamos en el rendimiento, este aumenta hasta llegar a la mitad del número de filas de la imagen destino (que coincide también con la mitad del número de columnas de esta). Este comportamiento se verá justificado en el siguiente apartado, al realizar un estudio más en profundidad de la dependencia del tiempo de escalado del algoritmo en software (su comportamiento no es lineal con respecto a la variación del número de filas).

En la segunda prueba definimos un tamaño fijo de la imagen destino (640x480) y se va variando el tamaño de la imagen fuente. A continuación mostramos los resultados obtenidos.

Imagen fuente	Tiempo (ms) en software	Tiempo (ms) en hardware	Rendimiento
1200x900	2638.67	22.201	118.853
1280x960	2786.81	22.806	122.196
1360x1020	2787.27	23.524	118.486
1440x1080	2790.08	24.262	114.997
1520x1140	2793.39	24.847	112.436
1600x1200	2795.4	25.582	109.272
1680x1260	2799.42	26.175	106.950
1760x1320	2801.05	26.903	104.116
1840x1380	2802.73	27.494	101.939
1920x1440	2806.08	28.225	99.418

Tabla 6.19 – Tiempos en escalado con dimensiones de la imagen destino fijas

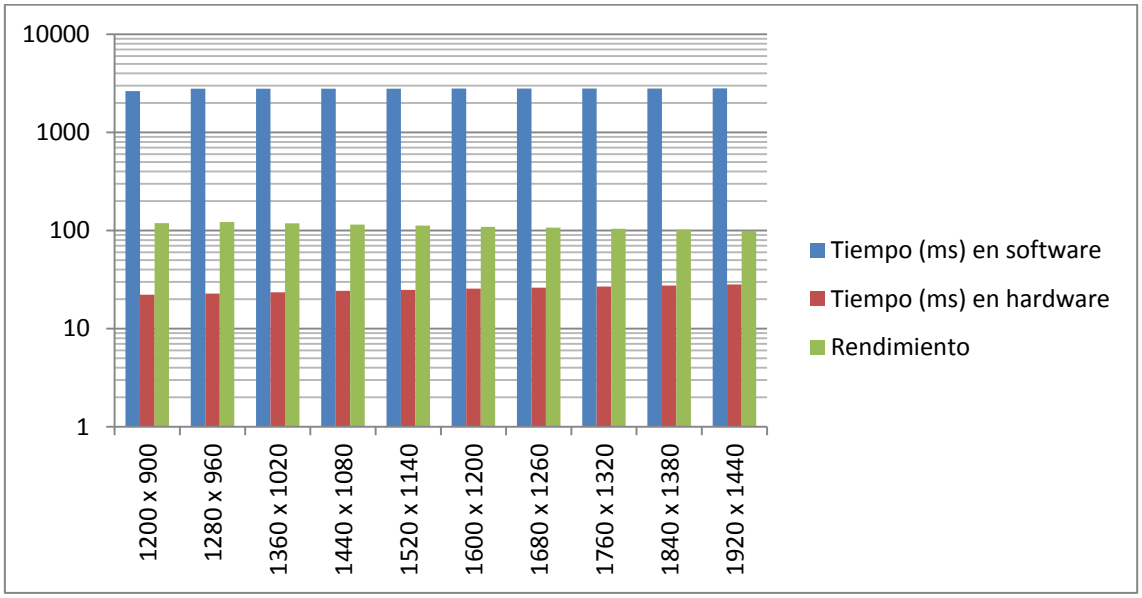


Imagen 6.11 – Gráfica de tiempos en escalado con dimensiones de la imagen destino fijas

En este caso podemos ver como la dependencia del tiempo de escalado de los algoritmos en software y hardware es menor respecto de las dimensiones de la imagen fuente. Aun así, esto no nos permite obtener conclusiones acerca del tiempo de escalado respecto de las dimensiones de las imágenes fuente y destino. También podemos observar, mirando los valores de rendimiento de la tabla 6.19, que este va aumentando hasta que el número de filas de la imagen fuente es el doble de la imagen destino, comportamiento apreciado también en la prueba anterior.

6.2.2 Comparativa entre el software y el hardware. Comportamiento en software

Para esta prueba se parte de la imagen fuente de 800x600 y se mantiene constante el número de columnas de la imagen destino para observar la dependencia del tiempo de escalado con respecto al número de filas de la imagen destino.

Imagen destino	Tiempo (ms) en software	Tiempo (ms) en hardware	Rendimiento
800x60	435.581	2.747	158.566
800x120	868.002	5.489	158.134
800x180	1302.42	8.238	158.099
800x240	1735.02	10.977	158.059
800x300	2168.58	13.732	157.921
800x360	2235.28	16.478	135.652
800x420	2299.64	19.209	119.716
800x480	2368.29	21.937	107.958
800x540	2435.4	24.717	98.531
800x600	2500.95	27.441	91.139

Tabla 6.20 – Tiempos en escalado con número de columnas fijo

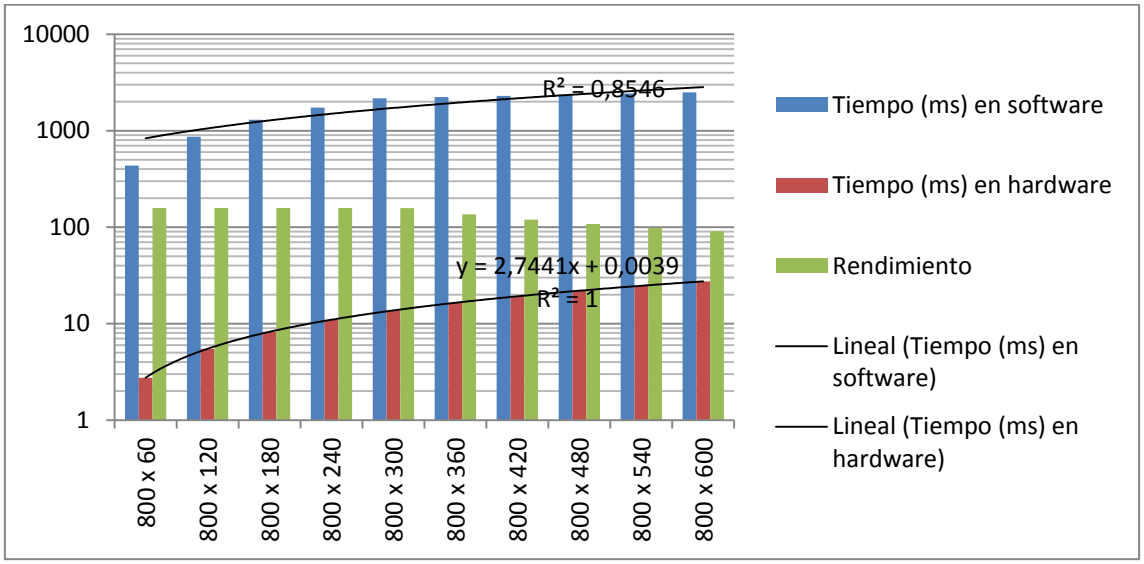


Imagen 6.12 – Gráfica de tiempos en escalado con número de columnas fijo

Como comentamos anteriormente, las rectas de regresión representadas sobre estas gráficas se corresponden en cuanto al valor del coeficiente de correlación R con la real, pero su ecuación no (ver siguiente imagen), puesto que el valor del eje de abscisas lo determina Excel. Para justificar este aspecto nos valemos de la siguiente imagen, en la cual se han representado los mismos valores de tiempo, pero dando en este caso el valor correspondiente al eje de abscisas, el número de filas de la imagen destino, de forma que la recta representada en el grafico si corresponde con la evolución real de dicha dimensión.

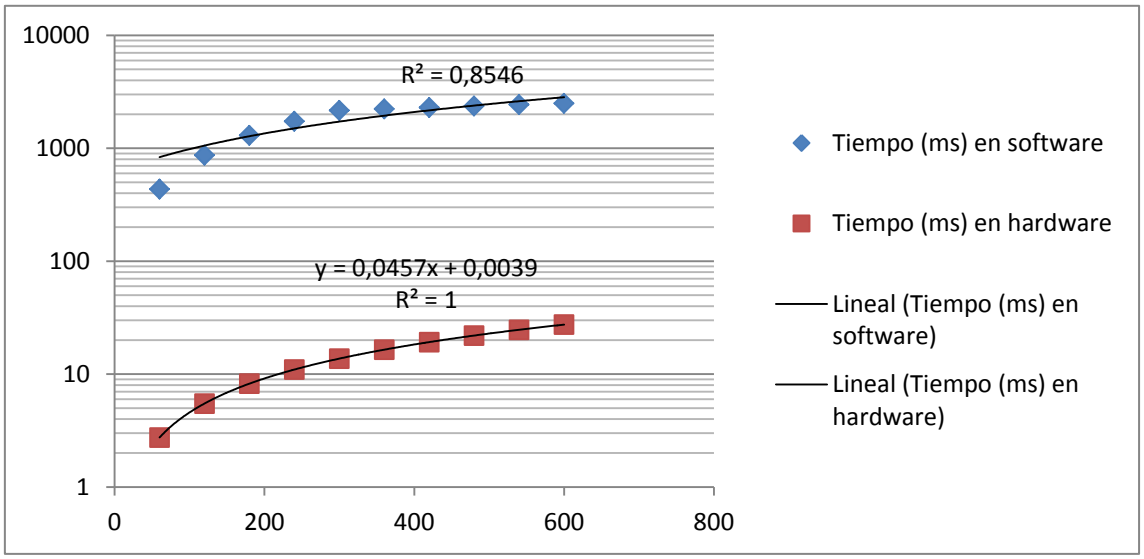


Imagen 6.13 – Rectas de regresión reales de la gráfica de la imagen 6.12

Como podemos observar, el algoritmo en hardware tiene un comportamiento lineal con respecto al número de filas de la imagen destino. No ocurre lo mismo con el algoritmo en software, dado que este no tiene un comportamiento lineal para la variación actual de dimensiones de la imagen destino como veremos a continuación. En consecuencia el rendimiento disminuye a medida que aumentan las dimensiones de la imagen destino.

A continuación mostramos un estudio para las distintas funciones del algoritmo en software con la misma variación de las dimensiones de la imagen destino. Para justificar el comportamiento anterior, representamos el tiempo de escalado de las funciones del algoritmo software que más tiempo consumen, siendo estas **imse_VResizeLinear**, **imse_HResizeLinear** e **imse_resizeGeneric**. La suma de los tiempos de las dos primeras, más un tiempo interno de la tercera, da el total de **imse_resizeGeneric**, que supone más del 99% del tiempo de **imse_Resize** (no se presentan los valores, solo la gráfica de estos).

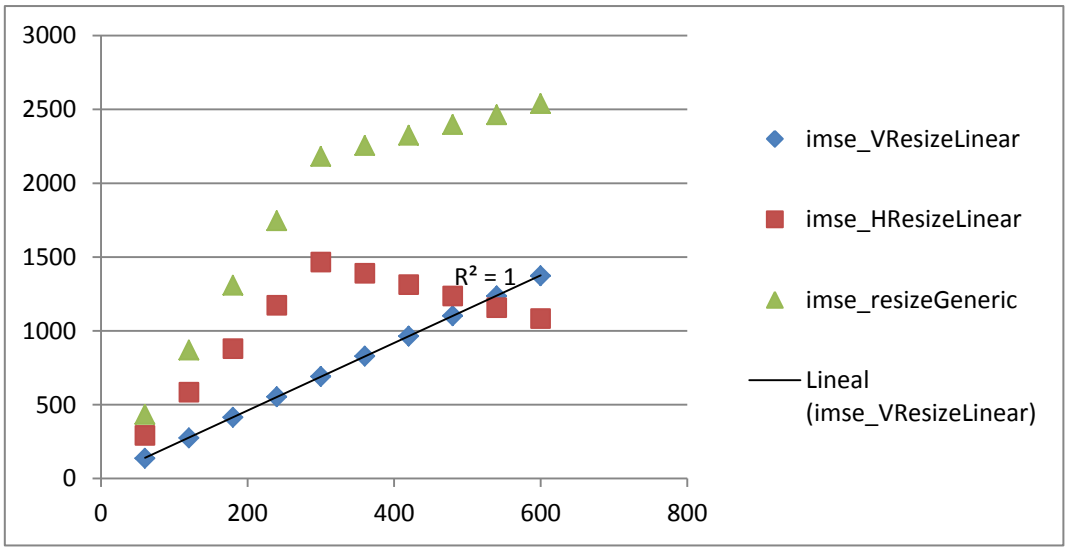


Imagen 6.14 – Gráfica de tiempos de las funciones principales del algoritmo en software 1

Como se puede apreciar, el comportamiento de **imse_VResizeLinear** es lineal respecto al número de filas de la imagen destino, pero no ocurre lo mismo con **imse_HResizeLinear**, lo que provoca la alinealidad vista en la gráfica de la imagen 6.13.

En la siguiente prueba tenemos también una imagen fuente de 800x600 y en este caso dejamos constante el número de filas de la imagen destino, para observar la dependencia del tiempo de escalado con respecto al número de columnas de la imagen destino. Los resultados obtenidos se pueden apreciar en la tabla 6.21 y en la imagen 6.15.

Imagen destino	Tiempo (ms) en software	Tiempo (ms) en hardware	Rendimiento
80x600	266.531	10.565	25.227
160x600	513.781	12.459	41.237
240x600	761.969	14.319	53.213
320x600	1010.3	16.203	62.352
400x600	1257.82	18.111	69.45
480x600	1506.28	19.936	75.555
560x600	1758.03	21.824	80.554
640x600	2009.34	23.739	84.642
720x600	2251.23	25.567	88.052
800x600	2500.94	27.439	91.145

Tabla 6.21 – Tiempos en escalado con numero de filas fijo

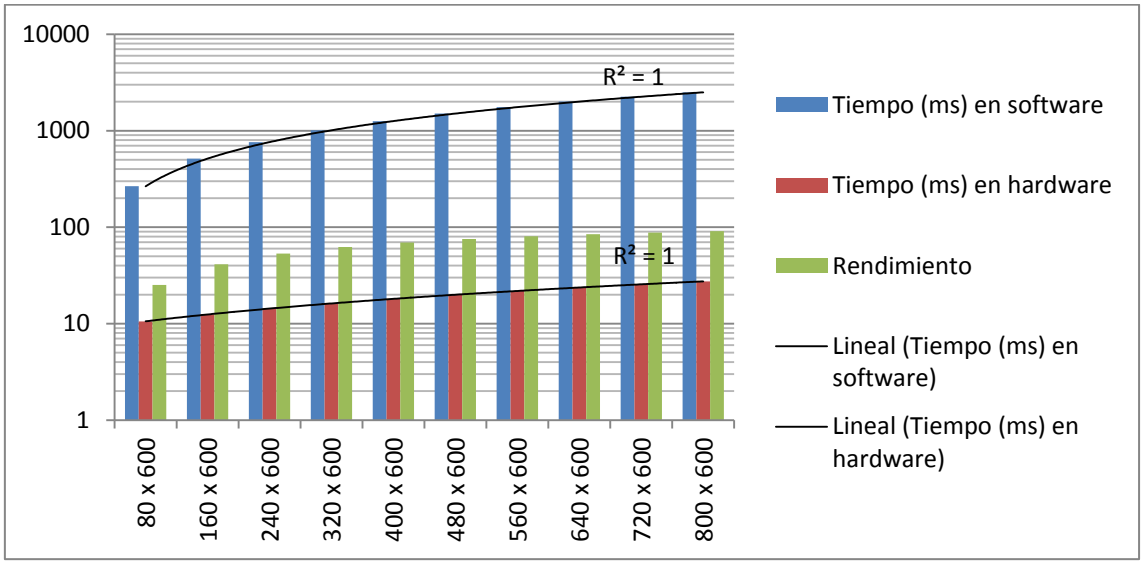


Imagen 6.15 – Gráfica de tiempos en escalado con número de filas fijo

Al igual que para la prueba anterior, representamos sobre una nueva gráfica las rectas de regresión correspondientes a los tiempos de escalado en software y hardware.

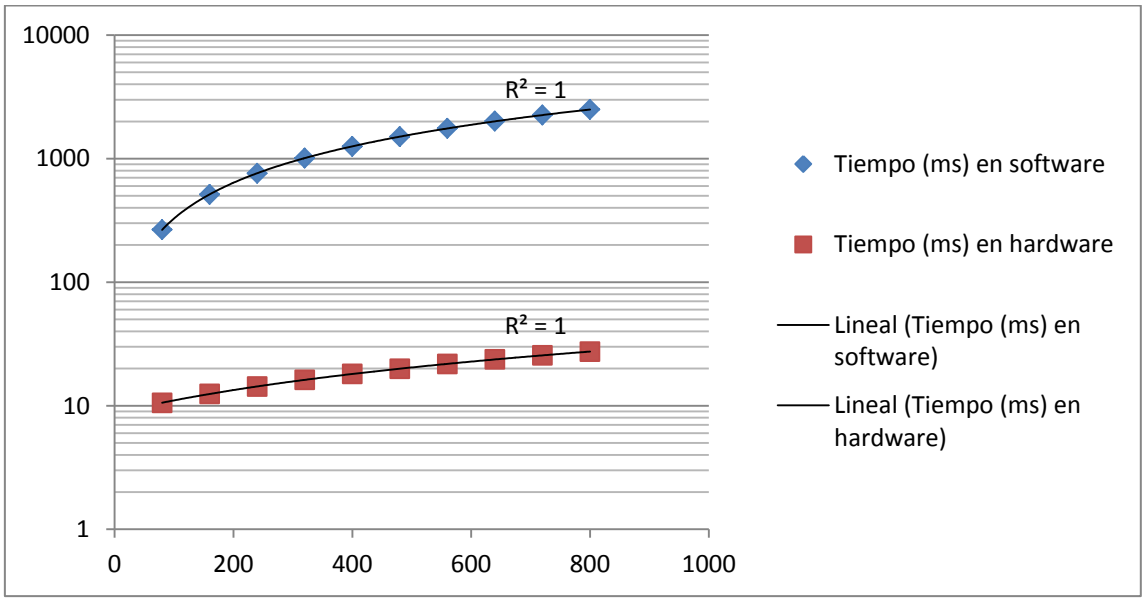


Imagen 6.16 – Rectas de regresión de la gráfica de la imagen 6.15

Como podemos ver, en este caso ambos algoritmos responden linealmente ante la variación del número de columnas manteniendo constante el número de filas.

Aquí volvemos a presentar el mismo estudio del algoritmo en software para justificar el comportamiento mostrado en la gráfica de la imagen anterior.

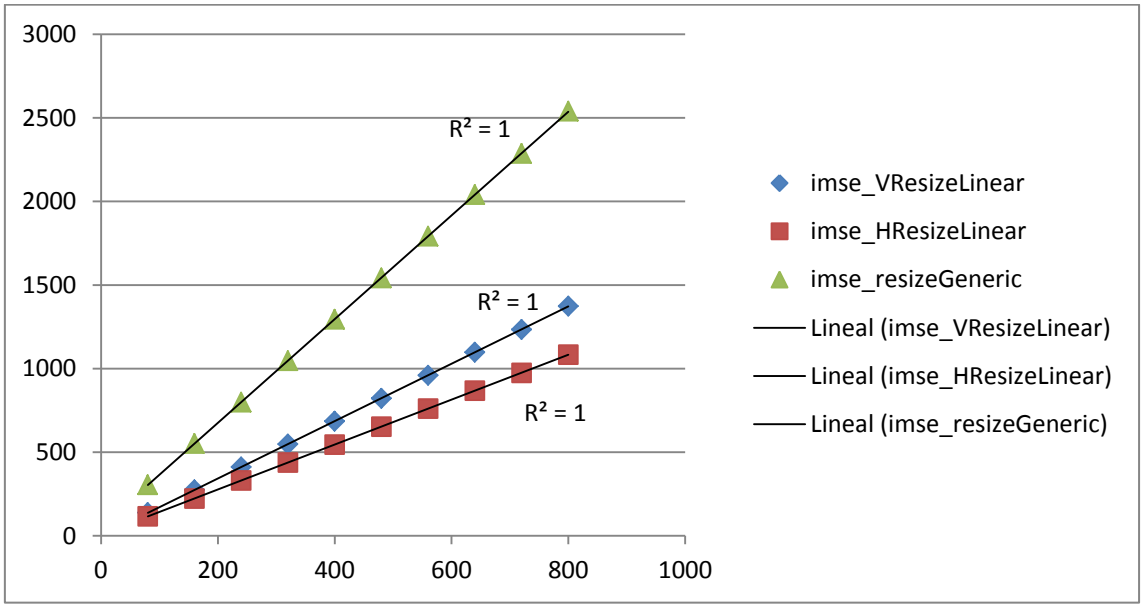


Imagen 6.17 – Gráfica de tiempos de las funciones principales del algoritmo software 2

Se tiene que la respuesta de las funciones principales (**imse_VResizeLinear** e **imse_HResizeLinear**) del algoritmo en software es lineal con respecto al número de columnas, lo cual deriva en un comportamiento lineal del algoritmo completo.

6.2.3 Comportamiento en hardware

En este apartado nos disponemos a determinar, en la medida de lo posible, una expresión que calcule el tiempo de escalado del algoritmo en hardware con respecto a las dimensiones de la imagen fuente y destino.

Este aspecto se posibilita gracias a las simulaciones y medidas en tiempo real de dicho algoritmo, lo cual resultaba de mayor dificultad con el algoritmo software y también de menos interés.

Gracias a los resultados obtenidos en las pruebas anteriores, llegamos a la principal conclusión de que el comportamiento del algoritmo en hardware es completamente lineal respecto a las dimensiones de la imagen destino.

Para finalizar el estudio, mostramos una simulación de un ciclo de escalado, a partir de la cual determinamos los porcentajes de tiempo que el algoritmo emplea en cada una de las funciones principales, lectura y escritura de datos y cálculo de los correspondientes a las filas de la imagen destino. Determinamos el tiempo a partir de la duración de los estados de la máquina de estados de TFG_IMAGE_SCALING que se correspondan con las funciones comentadas.

Así pues, tenemos que durante los estados $\mathbf{S_2}$ y $\mathbf{S_3}$ se están leyendo los datos de la fila even, en el estado $\mathbf{S_4}$ se están obteniendo los valores de la pseudo fila even, durante los estados $\mathbf{S_5}$ y $\mathbf{S_6}$ se están leyendo los datos de la fila odd, en el estado $\mathbf{S_7}$ se están obteniendo los valores de la fila de la imagen destino, durante los estados $\mathbf{S_8}$ a $\mathbf{S_{10}}$ se están escribiendo los valores de la fila de la imagen destino en la memoria y durante los estados $\mathbf{S_{11}}$ a $\mathbf{S_{15}}$ se determinan los valores

de las nuevas direcciones de lectura y escritura. Obviemos el estado $\mathbf{S_1}$ por su duración (un ciclo de reloj) y porque solo se da una vez por ciclo de escalado.

A continuación mostramos la simulación de los estados descritos anteriormente y la tabla de tiempos para determinar el porcentaje de utilización de cada función. Se está realizando un escalado desde 800x600 a 640x480.

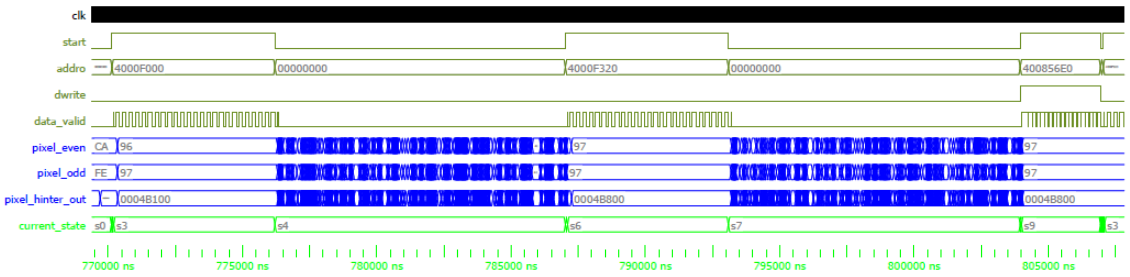


Imagen 6.18 – Simulación de la obtención de una fila de la imagen destino

Función / Tiempo	Inicio (ns)	Fin (ns)	Parcial (ns)	Porcentaje
Lectura de datos ($\mathbf{S_2}$ y $\mathbf{S_3}$)	770167.332	776218.33	6050.998	16.442
Obtención pseudo fila ($\mathbf{S_4}$)	776218.33	787001.664	10783.334	29.302
Lectura de datos ($\mathbf{S_5}$ y $\mathbf{S_6}$)	787001.664	793051.666	6050.002	16.44
Obtención fila destino ($\mathbf{S_7}$)	793051.666	803884.998	10833.332	29.438
Escritura de datos ($\mathbf{S_8}$ a $\mathbf{S_{10}}$)	803884.998	806884.998	3000	8.152
Cálculo de direcciones ($\mathbf{S_{11}}$ a $\mathbf{S_{15}}$)	806884.998	806968.332	83.334	0.226
Total fila destino	36801 ns	Total imagen destino	36801 · 480 = 17664480 ns	

Tabla 6.22 – Tiempos teóricos de las distintas funciones del algoritmo en hardware

Los tiempos aquí obtenidos son teóricos, es decir, no están afectados por posibles interrupciones del procesador y accesos de otros módulos al AHB bus. Así pues, esto afectaría principalmente a las funciones de transferencia de datos, como se puede apreciar al comparar con el tiempo de escalado obtenido en la tabla 6.18, siendo de 18.994 ms.

La forma correcta de determinar los tiempos de escalado sería midiendo en tiempo real la duración de cada estado, pero esto resulta difícil debido a la escasa precisión que proporciona la función *clock ()*. Para intentar determinar una expresión que relacione las dimensiones de las imágenes fuente y destino con el tiempo de escalado, asumimos que los tiempos que no influyen en transferencias de datos son constantes y repartimos la diferencia obtenida entre el tiempo teórico y el real entre dichas funciones. El tiempo a repartir es de:

$$18994000 - 17664480 = 1329520\ ns \rightarrow t = \frac{1329520}{480 \cdot 3} = 923.277\ ns \quad [6.1]$$

A continuación mostramos la tabla en la que se presentan los porcentajes (supuestos) de cada función en tiempo real.

Función / Tiempo	Parcial (ns)	Porcentaje
Lectura de datos (**S_2** y **S_3**)	6974.275	17.625
Obtención pseudo fila (**S_4**)	10783.334	27.251
Lectura de datos (**S_5** y **S_6**)	6973.299	17.622
Obtención fila destino (**S_7**)	10833.332	27.377
Escritura de datos (**S_8** a **S_{10}**)	3923.277	9.915
Cálculo de direcciones (**S_{11}** a **S_{15}**)	83.334	0.21
Total fila destino	39570.833 ns	

Tabla 6.23 – Tiempos "reales" de las distintas funciones del algoritmo en hardware

Teniendo en cuenta todo esto pasamos determinar una expresión que ayude a calcular el tiempo aproximado que durará el escalado en hardware. Así pues, el tiempo del cálculo de direcciones siempre es constante y los tiempos de obtención de datos los supondremos directamente proporcional al número de columnas de la imagen destino.

$$t_{parcial} = 83.334 + \frac{10833.332}{640} \cdot dst_{width} + \frac{10783.334}{640} \cdot dst_{width} \quad (ns) \quad [6.2]$$

$$t_{parcial} = 83.334 + 33.776 \cdot dst_{width} \quad (ns) \quad [6.3]$$

Lo siguiente es introducir el tiempo de las transferencias. Para las lecturas es

$$t_{lectura\_even} = \frac{6974.275}{800} \cdot src_{width} = 8.718 \cdot src_{width} \quad (ns) \quad [6.4]$$

$$t_{lectura\_odd} = \frac{6973.299}{800} \cdot src_{width} = 8.717 \cdot src_{width} \quad (ns) \quad [6.5]$$

y para la escritura

$$t_{escritura} = \frac{3923.277}{640} \cdot dst_{width} = 6.13 \cdot dst_{width} \quad (ns) \quad [6.6]$$

Con todo esto, el tiempo de escalado de una fila de la imagen destino es la suma de los tiempos anteriores:

$$t_{fila} = 17.435 \cdot src_{width} + 39.906 \cdot dst_{width} + 83.334 \quad (ns) \quad [6.7]$$

Finalmente, para la imagen completa se multiplica t_{fila} por el número de filas de la imagen destino, quedando:

$$t_{escalado} = [17.435 \cdot src_{width} + 39.906 \cdot dst_{width} + 83.334] \cdot dst_{height} \quad (ns) \quad [6.8]$$

Por ultimo mostramos una tabla de valores para determinar el error de esta ecuación para los tiempos de escalado de la tabla 6.18.

Imagen destino	Tiempo (ms) real	Tiempo (ms) expresión	Error relativo
80x60	1.058	1.033	0.0236 (2.36%)
160x120	2.493	2.449	0.0176 (1.76%)
240x180	4.297	4.249	0.0111 (1.11%)
320x240	6.483	6.432	0.0078 (0.78%)
400x300	9.058	8.998	0.0066 (0.66%)
480x360	11.964	11.947	0.0014 (0.14%)
560x420	15.281	15.279	0.0001 (0.01%)
640x480	18.994	18.994	0
720x540	23.014	23.092	- 0.0033 (- 0.33%)
800x600	27.442	27.573	- 0.0047 (- 0.47%)

Tabla 6.24 Error de la expresión del tiempo de escalado

Como podemos ver, la expresión obtenida aproxima con cierta precisión el tiempo de escalado, pero debemos considerar también que esta se ha obtenido a partir de una sola medición. Si queremos una expresión más fiel al tiempo real, debemos repetir varias veces los cálculos de las ecuaciones [6.1] a [6.8], lo cual conlleva bastante tiempo al tener que realizar un elevado número de simulaciones.

6.2.4 Comparativa entre el software en un PC y el hardware

A continuación mostramos una comparativa entre el tiempo de escalado de la función software en un PC con procesador Pentium Dual Core 2 GHz, 4 GB RAM y Windows 7 Professional de 64 bits y el hardware, con objeto de justificar el uso de este último.

Imagen destino	Tiempo (ms) en software (PC)	Tiempo (ms) en hardware (FPGA)
80x60	1	1.058
160x120	1	2.493
240x180	2	4.297
320x240	4	6.483
400x300	6	9.058
480x360	7	11.964
560x420	9	15.281
640x480	10	18.994
720x540	11	23.014
800x600	13	27.442

Tabla 6.25 – Tiempos de escalado con PC

Como se puede apreciar, los tiempos en software son menores que en hardware. Debido a la diferencia de frecuencias (2 GHz en software frente a los 60 MHz en hardware), se podría intuir que la ejecución en software fuera mejor, pero también debemos considerar que es un sistema de 64 bits frente a los 32 bits del procesador LEON3. Si se realizaran las optimizaciones descritas en el Capítulo 8, se superaría o al menos alcanzaría sin dificultad el tiempo de ejecución que se obtiene del software en el PC.

7. Presupuesto

En este capítulo se detalla el presupuesto necesario para el desarrollo del presente trabajo. Para ello se ha evaluado el coste en horas de trabajo de personal y los costes de material empleados. Por último se muestran distintos dispositivos en los que se puede implementar el módulo IP.

7.1 Costes de personal

Determinamos el coste de personal a partir del diagrama de Gantt presentado en el Capítulo 1. En este se detalla el número de días que se ha necesitado para llevar a cabo cada tarea del trabajo. A continuación mostramos dicho diagrama y un resumen con las horas trabajadas.

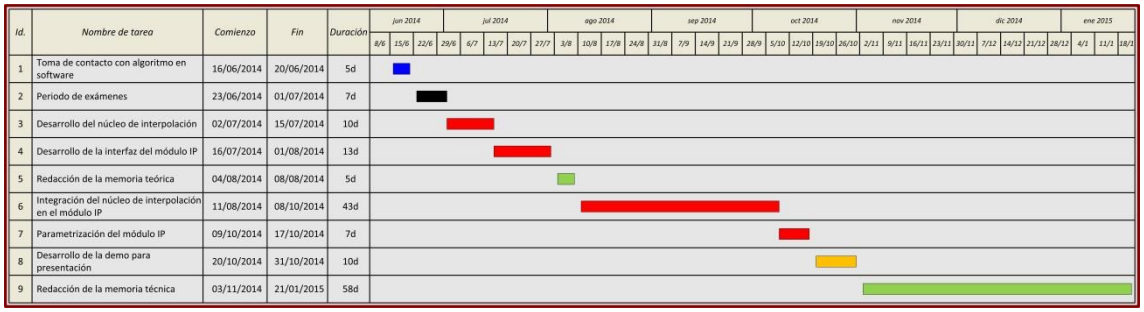


Imagen 7.1 – Diagrama de Gantt

Para determinar el total de horas empleadas en el desarrollo de este trabajo asumimos que los fines de semana no se han realizado trabajo alguno (si se realiza trabajo pero se incluye en las horas de los días laborables) y que las tareas de desarrollo de código han supuesto 8 horas/día y las tareas de redacción 6 horas/día. Suponemos el coste de un Ingeniero Industrial en 22 €/hora [50].

Tarea	Duración (días)	Horas/día	Total horas	Coste (€)
Toma de contacto con el algoritmo en software	5	6	30	660
Desarrollo del núcleo de interpolación	10	8	80	1760
Desarrollo de la interfaz del módulo IP	13	8	104	2288
Redacción de la memoria teórica	5	6	30	660
Integración del núcleo de interpolación en el módulo IP	43	8	344	7568
Parametrización del módulo IP	7	8	56	1232
Desarrollo de la demo para presentación	10	8	80	1760
Redacción de la memoria técnica	58	6	348	7656
Coste total (€)				23584

Tabla 7.1 – Costes de personal

7.2 Costes de material

Además del coste de personal, tenemos que incluir los costes del material, que dividimos en costes de hardware y costes de software, que se han necesitado en las distintas fases de desarrollo del trabajo. La expresión para obtener el coste de amortización es la siguiente:

$$amortizacion = \frac{uso\ del\ equipo}{periodo\ de\ depreciacion} \cdot coste\ del\ equipo \cdot \%\ uso\ dedicado\ al\ trabajo$$

El porcentaje de uso dedicado al trabajo es del 100%.

7.2.1 Costes de hardware

Concepto	Precio unitario (€)	Periodo de depreciación (meses)	Uso (meses)	Coste (€)
Portátil Acer Extensa 5635ZG	520	120	8	346.67
Virtex 5 FPGA Evaluation Platform ML505 + JTAG	1200 + 200	60	7	163.33
Monitor PC	90	-	-	90
Coste total (€)				600

Tabla 7.2 – Costes de hardware

7.2.2 Costes de software

Concepto	Precio unitario (€)	Periodo de depreciación (meses)	Uso (meses)	Coste (€)
Eclipse CDT	Libre	-	-	0
ISE Design Suite 13.4	Licencia WebPack	-	-	0
ModelSim 6.6	Licencia PE Student Edition	-	-	0
Microsoft Office Professional 2010	699.90	72	3	29.16
Microsoft Visio Premium 2010	1295	72	3	53.96
Coste total (€)				83.12

Tabla 7.3 – Costes de software

7.3 Presupuesto de ejecución material

En este apartado se obtiene el total que suponen los costes de personal y los costes de material. Los costes indirectos son del 20% del coste total.

Concepto	Coste (€)
Costes de personal	23584
Costes de hardware	600
Costes de software	83.12
Costes indirectos (20%)	4853.42
Total	28437.42

Tabla 7.4 – Presupuesto de ejecución material

7.4 Presupuesto de ejecución por contrata

Al presupuesto de ejecución material hay que añadir los gastos generales de empresa (13%) y el beneficio industrial (6%).

Concepto	Coste (€)
Presupuesto de ejecución material	28437.42
Gastos generales (13%)	3696.86
Beneficio industrial (6%)	1706.24
Total	33840.52

Tabla 7.5 – Presupuesto de contrata

7.5 Presupuesto total

Finalmente, se aplica el Impuesto del Valor Añadido (IVA) del 21%.

Concepto	Coste (€)
Presupuesto de contrata	33840.52
IVA (21%)	7106.5
Total (€)	40947.02

Tabla 7.6 – Presupuesto total

El presupuesto total asciende a la cantidad de CUARENTA MIL NOVECIENTOS CUARENTA Y SIETE CON CERO DOS EUROS.

7.6 Selección de FPGAs para la implementación del módulo IP

Para finalizar el capítulo de presupuesto, damos a conocer otras opciones de hardware para la implementación del módulo IP. Para ello realizamos la síntesis del proyecto de ISE con aquellos dispositivos que estén disponibles en la versión WebPack de este. En la siguiente imagen se presentan los dispositivos para los cuales existe un diseño del procesador LEON3.

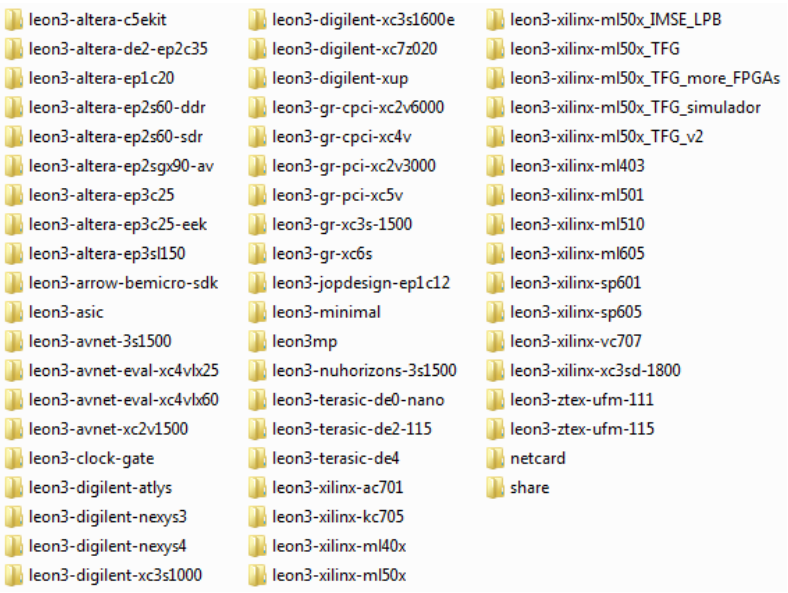


Imagen 7.2 – Diseño del procesador LEON3 para distintas FPGAs

El diseño empleado durante el desarrollo del módulo IP se corresponde con el denominado *leon3-xillinx-ml50x_TFG*.

A continuación escogeremos, de entre los dispositivos de Xillinx disponibles en la lista de la imagen anterior, los más ajustados y los de mayores prestaciones que permiten la implementación del módulo IP.

En la siguiente tabla mostramos todos los dispositivos para los que se ha realizado la síntesis del módulo IP y en caso de ser soportado, su precio.

Dispositivo FPGA	¿Soportado?	Precio (€)
ac701/XC7A200T-FBG484	No	
ac701/XC7A200T-FBG676	Si	1092.49
ac701/XC7A200T-FFG1156	Si	
kc705	No disponible en ISE WebPack	1487.96
ml403/XC4VLX25-FF668	No	
ml403/XC4VLX25-FF676	No	-
ml403/XC4VLX25-SF363	No	
ml501/XC5VLX50-FF324	No	
ml501/XC5VLX50-FF676	Si	1548.55
ml501/XC5VLX50-FF1153	Si	
ml505/XC5VLX50T-FF665	Si	1200
ml505/XC5VLX50T-FF1136	Si	
ml510	No disponible en ISE WebPack	3720.4
ml605	No disponible en ISE WebPack	1741.03
sp601/XC6SLX16-CPG196	No	
sp601/XC6SLX16-CSG225	No	-
sp601/XC6SLX16-CSG324	No	
sp601/XC6SLX16- FTG256	No	
sp605/XC6SLX45T-CSG324	No	
sp605/XC6SLX45T-CSG484	No	-
sp605/XC6SLX45T-FGG484	No	
vc707	No disponible en ISE WebPack	2948.48

Tabla 7.7 – Dispositivos FPGA

Las principal causa por la que muchos de los dispositivos presentados no soportan el módulo desarrollado, es debido a la gran cantidad de lógica que se emplea (hay que recordar que el módulo IP se desarrolla junto al procesador LEON3). Satisfecho este aspecto, el siguiente inconveniente es el elevado número de entradas/salidas que se necesitan para los distintos módulos implementados junto al procesador. A continuación mostramos los resultados de la síntesis para algunos de los dispositivos mostrados en la tabla anterior.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	10875	259600	4%
Number of Slice LUTs	18363	129800	14%
Number of fully used LUT-FF pairs	8571	20667	41%
Number of bonded IOBs	308	285	108%
Number of Block RAM/FIFO	35	365	9%
Number of BUFG/BUFGCTRLs	16	32	50%
Number of DSP48E1s	15	740	2%

Imagen 7.3 – Síntesis con ac701/XC7A200T-FBG484

Con la ac701 se excede el número de pines de entrada/salida disponibles.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	16773	10752	155%
Number of Slice Flip Flops	11175	21504	51%
Number of 4 input LUTs	29686	21504	138%
Number of bonded IOBs	307	448	68%
Number of FIFO16/RAMB16s	44	72	61%
Number of GCLKs	16	32	50%
Number of BUFIOs	8	32	25%
Number of DCM_ADVs	4	8	50%
Number of DSP48s	15	48	31%

Imagen 7.4 – Síntesis con ml403/XC4VLX25-FF668

Con la ml403 se excede los recursos lógicos disponibles.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	10787	18224	59%
Number of Slice LUTs	19491	9112	213%
Number of fully used LUT-FF pairs	8678	21600	40%
Number of bonded IOBs	308	106	290%
Number of Block RAM/FIFO	43	32	134%
Number of BUFG/BUFGCTRLs	16	16	100%
Number of DSP48A1s	15	32	46%
Number of PLL_ADVs	1	2	50%

Imagen 7.5 – Síntesis con sp601/XC6SLX16-CPG196

Con la sp601 se exceden los recursos lógicos, el número de pines de entrada/salida y los bloques de memoria disponibles.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	10658	54576	19%
Number of Slice LUTs	18335	27288	67%
Number of fully used LUT-FF pairs	8540	20453	41%
Number of bonded IOBs	308	190	162%
Number of Block RAM/FIFO	47	116	40%
Number of BUFG/BUFGCTRLs	16	16	100%
Number of DSP48A1s	15	58	25%
Number of PLL_ADVs	1	4	25%

Imagen 7.6 – Síntesis con sp6015/XC6SLX45Y-CSG324

Con la sp605 se excede el número de pines de entrada/salida disponibles.

8. Conclusiones y líneas de trabajos futuros

En este capítulo se presentan las principales conclusiones que se han obtenido tras el desarrollo del presente trabajo, dando a conocer los aspectos que supusieron mayor complejidad y comentando los resultados tras observar su funcionamiento en una aplicación de escalado de imagen.

Posteriormente se comentan las posibles aplicaciones del trabajo desarrollado y las principales mejoras que permiten ampliar su funcionalidad y eficiencia.

8.1 Conclusiones del trabajo

El objetivo principal de este trabajo era el desarrollo e implementación de un módulo IP para el escalado de imágenes. Su uso está destinado a la aceleración de una aplicación de detección de rostros que actualmente utiliza una función software para el escalado de imagen. El módulo IP ha sido diseñado para ser utilizado en un sistema empotrado (*SoC*). Se ha desarrollado bajo la arquitectura AMBA bus, de forma que puede ser integrado junto a un diseño básico del procesador LEON3, lo que permite realizar la configuración de los distintos parámetros del módulo IP.

Dado que se carecía de conocimientos de los algoritmos de interpolación para el escalado de imágenes, se partió de la función que se empleaba en una aplicación de detección de rostros. A partir de esta se comenzó a generar el algoritmo en hardware. En la primera implementación, y que pasó a ser la definitiva, se pretendió seguir el mismo flujo de datos que en el algoritmo software, reduciendo así la complejidad que supone el desarrollo en hardware, al poder realizar pruebas en cada una de las etapas del algoritmo que confirman la correcta funcionalidad de este.

El primer paso fue el desarrollo del núcleo de interpolación, destinado a la obtención de una fila de la imagen destino. Tras obtener una correcta funcionalidad de este, se determinó acelerar su funcionamiento mediante el empleo de la técnica del *pipeline*. El desarrollo de este núcleo no presentó excesiva dificultad, ya que se hacía pleno uso de los recientes conocimientos adquiridos en la asignatura de Diseño Digital Avanzado. Una vez concluido este aspecto, se pasó a desarrollar la interfaz del módulo IP. Para ello se emplearon dos módulos de comunicación compatibles con la arquitectura AMBA bus el APB slave (para la configuración de los registros del módulo IP) y el AHB master (para las transferencias de datos). En este punto comienza a complicarse el desarrollo del trabajo, debido a que se detectan errores en la funcionalidad de estos dos módulos (principalmente en el AHB master), que hay que corregir para poder continuar con la siguiente etapa. Tras esto se pasó a desarrollar el algoritmo que sincroniza las transferencias de datos con el núcleo de interpolación. De nuevo se dificulta su desarrollo, principalmente por el uso de lógica combinacional en los algoritmos de selección de datos para el núcleo de interpolación y de generación de palabras de 32 bits para la escritura de datos en memoria. Dicho problema se trataba de un correcto funcionamiento en el software de simulación, pero un comportamiento aleatorio en la ejecución en hardware, siendo la única y correcta solución el empleo de lógica secuencial, a cambio de un retraso de unos cuantos ciclos de reloj. Una vez se obtiene una correcta funcionalidad del módulo IP, se pasa a la última fase (y el aspecto más interesante del desarrollo de hardware), la parametrización del diseño para ajustarlo a las necesidades de la aplicación a la que se destine

y a las limitaciones de recursos lógicos del dispositivo donde se implemente. Este aspecto introduce limitaciones en las dimensiones mínimas de las imágenes con las que puede interaccionar el módulo IP, no suponiendo esto un problema considerable, pues hablamos de imágenes de dimensiones menores de 2^4 pixeles.

En todo momento estaba presente la complejidad de la depuración del hardware. Esta, unida al tiempo de síntesis del código, dificultó enormemente las últimas etapas de desarrollo del módulo IP, alargando el tiempo previsto de finalización. A efectos de comprensión de este aspecto, el tiempo aproximado de síntesis en el portátil usado en el desarrollo del trabajo (Pentium Dual Core a 2 GHz, 4 GB RAM y Windows 7 Professional de 64 bits) era aproximadamente de una hora, mientras que en un pc de sobremesa de mayores prestaciones (Intel Core i7-4770, 16 GB RAM y Windows 7 Professional de 64 bits) el tiempo rondaba los 15 minutos aproximadamente.

Como objetivo adicional al desarrollo del módulo IP de escalado de imagen, se propuso ejecutar una aplicación que permitiera ver el funcionamiento en tiempo real. Esto conllevó interactuar con la interfaz SVGA para desarrollar una aplicación que mostrara por pantalla la imagen fuente (primero en color y a continuación en gris, que es sobre la que se aplica el escalado) y a continuación la ejecución del módulo IP con distintas dimensiones de la imagen destino presentando los tiempos de escalado en software y en hardware. Aquí se ha de destacar el defecto *screen tearing* (“lagrimeo de pantalla”) que se produce en la representación de la imagen destino cuando se está efectuando un escalado en hardware, debido a la desincronización que se produce en la actualización del buffer de la interfaz VGA provocada por las continuas transferencias de datos por el AHB bus.

En definitiva, y como conclusiones principales de lo anteriormente expuesto, se deriva la complejidad del desarrollo en hardware, justificado con los excelentes resultados en su uso frente al software y el completo desarrollo de los objetivos propuestos. En lo personal, el desarrollo del presente trabajo ha supuesto una grata experiencia a la vez que una tarea que ha requerido mucho tiempo y esfuerzo compensada con la satisfacción de los resultados obtenidos y el excelente entorno de trabajo que brinda el Instituto de Microelectrónica de Sevilla además de los conocimientos adquiridos en los aspectos de desarrollo de diseños digitales para dispositivos FPGA. Aun siendo la Electrónica Analógica el origen de mi interés por la carrera de Grado en Ingeniería Electrónica Industrial, la conclusión de este trabajo ha supuesto un incentivo para decantar la continuación de mis estudios en esta rama de la Ingeniería Industrial.

8.2 Líneas de trabajos futuros

Como es evidente, y se ha justificado a lo largo de este texto, el módulo IP desarrollado se encuentra, por decirlo de alguna manera, en su versión más primitiva. Es decir, realiza correctamente la función para la que se va a destinar inicialmente, pero bien es cierto, que dada la relación que tiene con muchos aspectos cotidianos de la tecnología, sería muy interesante enumerar posibles trabajos futuros que amplíen su funcionalidad y mejoren su funcionamiento.

De entre los aspectos más relevantes en cuanto a aumentar su funcionalidad podemos enumerar los siguientes:

- El más importante, y en el cual se ha comenzado a trabajar, está el dotar al diseño de la funcionalidad completa del algoritmo en software, es decir, que este permita también la ampliación de escala, objetivo que conlleva observar casos especiales, lo cual implica cierta complejidad en hardware.
- Otro aspecto muy interesante pasa por implementar otros algoritmos de interpolación, como *nearest neighbor* o la interpolación bicúbica, y poder seleccionar el que más convenga cuando se tengan limitaciones temporales o se busque un mejor rendimiento visual.

Igualmente podemos introducir mejoras que permitan un funcionamiento más eficiente del algoritmo en hardware, entre las que podemos destacar:

- Redefinir el flujo de datos del núcleo de interpolación, de forma que introduciendo otro componente de interpolación horizontal (TFG_HRESIZE_KERNEL), se reduzca a la mitad el tiempo de obtención de una fila de la imagen destino, a expensas de algunos recursos limitados (como multiplicadores) pero que también conllevaría la eliminación de la memoria TFG_ROW_BUFFER. También se vería reducida la complejidad de la máquina de estados de TFG_ROW_INTERPOLATION al seguir empleando la técnica del *pipeline*.
- Si se adoptara la anterior propuesta, también reduciríamos la complejidad de la máquina de estados de TFG_IMAGE_SCALING, al necesitar solo una lectura y una escritura de datos en la memoria externa. Por el contrario, deberíamos aumentar la capacidad de **src_ram** para que fuera capaz de albergar dos filas de la imagen fuente y también deberíamos duplicar el algoritmo de selección de datos de esta.
- Otra mejora relevante sería hacer uso de la arquitectura AMBA bus en su versión de 64 bits, frente a los 32 actuales, de forma que reduciríamos a la mitad el tiempo de las transferencias de datos.

Una vez desarrollada una versión más completa y eficiente del módulo IP se propone la opción de ponerla a disposición de los usuarios que precisen de su utilización en la librería S.H.O.R.E.S [48].

Bibliografía y referencias

[1] Embedded Vision Alliance.
www.embedded-vision.com

[2] Wikipedia, “Quality Intellectual Property Metric”.
http://en.wikipedia.org/wiki/Quality_Intellectual_Property_Metric#mediaviewer/File:IP_Quality_%26_Reuse.jpg

[3] OpenCV.
http://opencv.org/

[4] Wikipedia, “OpenCV”.
http://es.wikipedia.org/wiki/OpenCV#mediaviewer/File:OpenCV_Logo_with_text.png

[5] ARM – The Architecture for the Digital World, “AMBA Specifications”.
http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[6] Aeroflex Microeletronic Solutions, “LEON3 Processor”.
http://www.gaisler.com/index.php/products/processors/leon3

[7] Visión artificial e interacción sin mandos.
http://sabia.tic.udc.es/gc/Contenidos%20adicionales/trabajos/3D/VisionArtificial/index.html

[8] A. G. Marcos, F.J.M. de Pisón Ascacíbar, A. V. P. Espinoza, F. A. Elías, M. C. Limas, J. O. Meré, E. V. González, “Técnicas y algoritmos básicos de visión artificial”. *Introducción*. Universidad de la Rioja. Servicio de Publicaciones. 2006.
https://publicaciones.unirioja.es/catalogo/online/VisionArtificial.pdf

[9] Tecnología e Inteligencia Artificial, “Contenido sobre la I.A y Tecnologías”.
http://tecnointart.files.wordpress.com/2013/08/pizap-com13812924947751.jpg

[10] Wikipedia, “Computer Vision”.
http://en.wikipedia.org/wiki/Computer_vision#mediaviewer/File:CVoverview2.svg

[11] Dolthink: Soluciones Tecnológicas para Empresas, “Visión artificial: Una imagen vale más que mil palabras”.
http://www.dolthink.com/vision-artificial.html

[12] Fernando Ortiz Rellina,”Creación de mapas visuales panorámicos mediante técnicas de visión artificial”. *Estado del arte*. Proyecto de Fin de Carrera. Universidad Carlos III de Madrid. 2012.
http://e-archivo.uc3m.es/bitstream/handle/10016/15231/pfc_fernando_ortiz_renilla_2012.pdf?sequence=8

[13] Wikipedia, “Semiconductor intelectual property core”.
http://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core

[14] I. Tuomi, “The Future of Semiconductor Intellectual Property Architectural Blocks in Europe”. *Executive Summary*. *Introduction*. JRC Scientific and Technical Reports. 2009.
http://ftp.jrc.es/EURdoc/JRC52422.pdf

[15] OpenCores.
www.opencores.org

[16] A. González, F.J. Martínez, A. V. Pernía, F. Alba, M. Castejón, J. Ordieres, E. Vergara, (integrantes del Grupo de Investigación EDMANS) “Técnicas y algoritmos básicos de visión artificial”. *Introducción. Preprocesado*. Universidad de la Rioja. Servicio de Publicaciones. 2006.
https://publicaciones.unirioja.es/catalogo/online/VisionArtificial.pdf

[17] Carmen Virginia Gámez Jiménez, “Diseño y Desarrollo de un Sistema de Reconocimiento de Caras”. *Estado del arte*. Proyecto de Fin de Carrera. Universidad Carlos III de Madrid. 2009.
http://e-archivo.uc3m.es/bitstream/handle/10016/5831/PFC_CarmenVirginia_Gamez_Jimenez.pdf?sequence=1

[18] Android Ayuda, “Pantallas ¿Qué son el tamaño las resolución y la densidad de pixeles?”.
http://androidayuda.com/app/uploads/2012/09/pixel-3.jpg

[19] Instituto Portugês do Mar e da Atmosfera, “Área educativa – Satélites meteorológicos”.
https://www.ipma.pt/pt/educativa/observar.tempo/index.jsp?page=satelite03.xml&print=true

[20] Intel (R) Integrated Performance Primitives Refence Manual, “Color Models”.
https://software.intel.com/sites/products/documentation/doclib/iss/2013/ipp/ipp_manual/IPPI/ippi_ch6/ch6_color_models.htm

[21] SAV – Universidad de Sevilla, “Multiplicación y División de Imágenes”.
http://www.sav.us.es/formaciononline/asignaturas/asigpid/apartados/textos/recursos/multiplicacionDivision/Index.htm

[22] Universidad de Murcia, Teledetección, “Tema 6 – Técnicas de filtrado”.
http://www.um.es/geograf/sigmur/teledet/tema06.pdf

[23] J. Vélez, B. Moreno, A. Sánchez, J. L. Esteban, “Visión por computador”. *Filtrado y realzado de imagen*. 2003.
http://www.escet.urjc.es/~visionc/VisionPorComputador.pdf

[24] Ethan E. Danahy, Sos S. Agaian, Karen A. Panetta, “Algorithms for the resizing of binary and grayscale images using a logical transform”. *Introduction*. SPIE Proceedings Vol. 6497. 2007.
https://sisu.ut.ee/sites/default/files/imageprocessing/files/interpolation2.pdf

[25] Tech-Algorithm, “Nearest Nighbor Image Scaling”.
http://tech-algorithm.com/articles/nearest-neighbor-image-scaling/

[26] Cambridge in Colour – Photography Tutorials & Learning Community, “Digital Image Interpolation”.
http://www.cambridgeincolour.com/tutorials/image-interpolation.htm

[27] Vision System Design – Machine Vision Systems and Image Processing Applications, “understanding image-interpolation techniques”.
http://www.vision-systems.com/articles/print/volume-12/issue-10/departments/wilsons-websites/understanding-image-interpolation-techniques.html

[28] Wikipedia, “Interpolación bilineal”.
http://es.wikipedia.org/wiki/Interpolaci%C3%B3n_bilineal

[29] Wikipedia, “Bilinear Interpolation”.
http://en.wikipedia.org/wiki/Bilinear_interpolation

[30] Tech-Algorithm, “Bilinear Image Scaling”.
http://tech-algorithm.com/articles/bilinear-image-scaling/

[31] The Supercomputing Blog, “Coding Bilinear Interpolation”.
http://supercomputingblog.com/graphics/coding-bilinear-interpolation/

[32] ALGLIB, “Bilinaer and bicubic spline interpolation”.
http://www.alglib.net/interpolation/spline2d.php

[33] Wikipedia, “Interpolación bilineal”.
http://es.wikipedia.org/wiki/Interpolaci%C3%B3n_bilineal#mediaviewer/File:Bilinear_interpolation.png

[34] Wikipedia, “Bicubic interpolation”.
http://en.wikipedia.org/wiki/Bicubic_interpolation

[35] Wikipedia, “Stairstep interpolation”.
http://en.wikipedia.org/wiki/Stairstep_interpolation

[36] NickGallery, “Stair interpolation”.
http://homepage.ntlworld.com/nick.thomas93/interpolation/interpolation.htm

[37] Wikipedia, “Gráfico vectorial”.
http://es.wikipedia.org/wiki/Gr%C3%A1fico_vectorial

[38] “Imagen digital: conceptos básicos”.
http://platea.pntic.mec.es/~lgonzale/tic/imagen/conceptos.html

[39] Design Automation Standards Committee of the IEEE Computer Society, “IEEE Estándar VHDL Languaje Reference Manual”. IEEE Std 1076-2008.
http://web02.gonzaga.edu/faculty/talarico/polibari/documents/VHDLrefManual.pdf

[40] Fernando Pardo, Jose A. Boluda, “VHDL: Lenguage para sintesis y modelado de circuitos”. Departamento de Informática y Electrónica. Universitat Valencìa. 1999.

[41] Xillinx, “ML505/ML506/ML507 Evaluation Platform. User guide”. 2006.
http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf

[42] Xillinx, “ML505 front”.
http://www.xilinx.com/images/boards/ml505/ml505_front.jpg

[43] Xillinx, “ML505 back”.
http://www.xilinx.com/images/boards/ml505/ml505_back.jpg

[44] Aeroflex Microeletronic Solutions, “LEON3 Processor”.
http://www.gaisler.com/index.php/products/processors/leon3

[45] Aeroflex Microeletronic Solutions, “LEON C/C++ IDE for Eclipse”.
http://www.gaisler.com/index.php/products/compilers/51-section-software-category-ide/148-leon-cc-ide-for-eclipse

[46] Aeroflex Gaisler, “GRLIB IP Core User´s Manual”. *LEON3/FT – High-performance SPARC V8 32-bit Processor*. 2014.
http://www.gaisler.com/products/grlib/grip.pdf

[47] ARM, “AMBA Specification”. *Introduction to the AMBA Buses*. *AMBA Signals*. *AMBA AHB*. *AMBAAPB*. 1999.
http://www-micro.deis.unibo.it/~magagni/amba99.pdf

[48] Software & Hardware Open Repository for Embedded Systems, “Hardware”.
http://www2.imse-cnm.csic.es/shores/index.php/Hardware

[49] Wikipedia, “Arquitectura en pipeline (informática)”.
http://es.wikipedia.org/wiki/Arquitectura_en_pipeline_%28inform%C3%A1tica%29

[50] INGEAGRO, “Costes de Personal”.
http://www.ingeagro.es/Tarifas.html

Anexo A

A.1 Función imse_Resize

A.2 Código para simulaciones

A.3 Código empleado en el capítulo de resultados

A.4 Código empleado en el video de la demo de presentación

Anexo B

B.1 leon3mp.vhd

B.2 TFG_IMAGE_SCALING.vhd

B.3 TFG_AHB_MASTER.vhd

B.4 TFG_APB_SLAVE.vhd

B.5 TFG_ROW_INTERPOLATION.vhd

B.6 TFG_RESIZE_EMBB.vhd

B.7 TFG_HRESIZE_KERNEL.vhd

B.8 TFG_VRESIZE_KERNEL.vhd

B.9 TFG_ROW_BUFFER.vhd