Universidad de Buenos Aires
Facultad de Ciencia Exactas y Naturales
Departamento de Computación

ALGORITMO ADM DE RECONOCIMIENTO DE BORDES EN IMÁGENES IMPLEMENTADO EN FPGA

Tesis presentada para optar al título de Licenciado en Ciencias de la Computación

Maximiliano Antonio Sacco
Directora:
Dra. Patricia Borensztejn

Buenos Aires, 27 de Julio de 2009

Resumen

Los FPGA (Field Programmable Gate Array) son dispositivos formados por miles de celdas básicas de lógica y memoria, interconectadas entre sí. Tanto las celdas lógicas como sus interconexiones pueden programarse (configurarse) para implementar los algoritmos que uno desee.

El interés que tienen estos dispositivos desde el lado de las ciencias de la computación es que posibilitan el desarrollo de sistemas diseñados para ser embebidos en algún producto o sistema mayor. La computación embebida, considerada hoy como la siguiente generación de computadoras por la diseminación de sus productos y el tamaño creciente de su mercado es la principal consumidora de dispositivos cuyos requerimientos son diferentes a aquellos que satisfacen los microprocesadores convencionales. Para realizar procesamiento embebido generalmente es necesario contar con dispositivos pequeños, rápidos, inteligentes, conectados, potentes, y baratos.

Los FPGA son dispositivos de muy bajo consumo, que funcionan a bajas frecuencias de reloj, pero que obtienen rendimientos altos debido a que no son mas que bloques de lógica que pueden ser programados para que funcionen en paralelo.

Hemos puesto especial énfasis en el desarrollo metodológico del diseño de la aplicación debido a que el paradigma con que se los programa (utilizando lenguajes de descripción de hardware) es distinto al paradigma secuencial al que el programador o estudiante de ciencias de la computación está acostumbrado.

La aplicación que se ha diseñado es una aplicación de adquisición de imágenes y reconocimiento de bordes en dicha imagen. El algoritmo de detección de bordes que se aplica a la imágenes es el algoritmo ADM. Las imágenes son imágenes en blanco y negro de 512 por 512 pixels. El sistema corre sobre una FPGA tipo Spartan3 de xilinx, obteniendo un rendimiento de menos de 3 milisegundos por imagen procesada.

Abstract

The FPGAs (Field Programmable Gate Array) are devices formed by thousands of basic logic cells and a memory wired together. The logic cells as well as their connections can be programmed (configured) to deploy any algorithm you want.

These devices are particularly interesting from the computer sciences point of view due to the fact that they enable the development of systems designed to be embedded in a greater product or system. Embedded computing, considered today as the next generation of computers given the spread of its products and its increasing market, is the main consumer of devices whose requirement are different from those satisfied by the conventional microprocessors. To carry out embedded processing, it is typically necessary to have small, fast, intelligent, powerful, connective and economic devices.

The FPGAs are devices with a very low energy consumption, which work at low clock frequencies but nevertheless obtain high performances, since they are nothing but logic blocks that can be programmed to work in parallel.

We have specially focused on the methodological development of the application, since the paradigm under which they are programmed (using hardware description languages) is different from the sequential paradigm to which programmers or computer sciences students are used to.

The application designed is one for image acquisition and border recognition in the image acquired. The border detection algorithm applied to the images is the ADM. The images used are black and white, of 512 x 512 pixels. The system runs in a xilinx Spartan3-type FPGA, obtaining a performance of less than 3 milliseconds by image processed.

Índice general

Capítulo 1

Introducción

La computación embebida es considerada hoy en día como la siguiente generación de computadoras. El mercado es tan amplio, y la diseminación de sus productos es tan significativa que según el consultor Jim Turley (editor de Microprocessor Report) si redondeamos los números, los sistemas embebidos consumen el 100 % de la producción de los microprocesadores [10]. Esta nueva generación (también llamada post-PC) esta formada por dispositivos pequeños, rápidos, inteligentes, conectados, potentes, y baratos. Entre las aplicaciones que corren en estos dispositivos las más características son procesamiento de imágenes, comunicaciones, robótica y aplicaciones para la industria automotriz. Los productos embebidos incluyen un elemento de proceso. Estos elementos de proceso pueden ser desde microprocesadores o procesadores de propósito general, pero también pueden ser procesadores especificamente diseñados para ser embebidos, o microcontroladores, o bien procesadores específicos para señales (DSP), y, debido a que las actuales FPGA son muy potentes, algunas de ellas incorporando núcleos de microprocesadores y conteniendo millones de puertas lógicas, también pueden ser utilizadas para implementar un sistema embebido completo. Una gran ventaja de las FPGA es que se han vuelto muy populares, tanto las placas de desarrollo como sus entornos de programación y es esto mismo lo que las torna particularmente interesantes para el desarrollo de prototipos y para la investigación en ámbitos académicos.

Por lo tanto, nuestro objetivo es utilizar una de estas plataformas para desarrollar e implementar un prototipo de sistema embebido, demostrando además que esta tecnología esta al alcance de la ciencia de la computación. El sistema embebido que elegimos desarrollar es un dispositivo capaz de adquirir imágenes a través de un puerto de comunicación, procesar dicha imagen y devolverla por el mismo canal. El procesamiento elegido para la imagen es un algoritmo de detección de bordes. Se pretende entonces, explorar las gran-

des posibilidades de paralelismo que ofrecen los FPGA y también evaluar la performance alcanzada por los FPGA para considerarlos como alternativa a los ASICs, CPLD, o incluso microprocesadores y microcontroladores en la ejecución de algoritmos específicos que deberán correr en situaciones donde una PC no es conveniente, ya sea por consumo, tamaño, movilidad, costos, performance, etc. Por otro lado, se utilizara una metodología de diseño y codificación que acerque esta tecnología, muy utilizada en el área de la Ingeniería electrónica, a la ciencia de la computación; ocultando dentro de lo posible, los detalles propios del sistema electrónico y utilizando conceptos y herramientas usuales en la computación y construcción de algoritmos.

A continuación se presentara un breve resumen sobre los orígenes tecnológicos y antecesores del FPGA y una descripción conceptual del mismo. En el Capitulo 2 se describirá el algoritmo de detección de bordes en imágenes[3]. El Capitulo 3 explica la tecnología disponible para la implementación del algoritmo y presenta la arquitectura general con que sera construido. Del Capitulo 4 al 6 se diseña la interfase y el comportamiento de los principales módulos necesarios para la construcción del algoritmo. En el capitulo 7 se explica la metodología usada para la programación del FPGA. Finalmente, en el Capitulo 8, se presentan los resultados obtenidos en la ejecución real del algoritmo y las conclusiones pertinentes, como así también se proponen futuras continuación de este trabajo y otras áreas interesantes para seguir investigando.

1.1. Orígenes del FPGA

A continuación se hará una introducción a la tecnología que sera utilizada a lo largo de todo este trabajo. Esta reseña fue extraida de [6]

1.1.1. PLDs

El primer circuito integrado programable, fue llamado *PLD* (programmable logic devices), comenzó a aparecer en escenario en la década de 1970. Al principio eran bastante simples y consistían casi únicamente en PROMS. A lo largo de toda la década fueron ganando complejidad y se llamaron *CPLD* (complex PLD) y se distinguieron de sus ancestros que pasaron a denominarse en contraste *SPLD* (simple PLD).

PROMs

Las PROMs (programmable read only memory), eran básicamente una tecnología basada en fusibles, estos son un enlace que, dependiendo la pro-

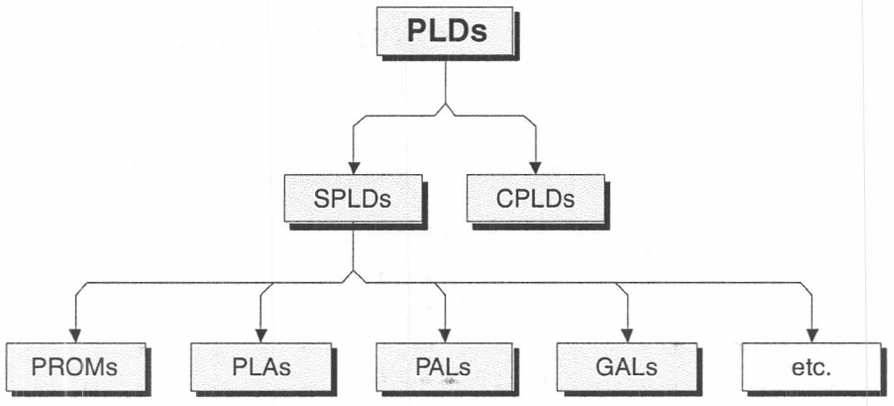


Figura 1.1: Esquema jerárquico de PLDs

gramación, podían quemarse interrumpiendo así la circulación de corriente a través del enlace y de esta manera representaban un 0 lógico.

El primero de los PLDs simples fue construido sobre un PROM. Consistía en un arreglo de compuertas AND con sus enlaces fijos conectados a las entradas y su salida conectada a un arreglo de compuertas OR con enlaces programables. La figura 1.2 muestra un esquema representativo del funcionamiento de un SPLD PROM. Así, uno podía construir funciones lógicas, representadas como suma de productos de las variables de entrada, programando los enlaces del arreglo de compuertas ORs. Como el arreglo de compuertas ANDs estaba predefinido por el fabricante, estos SPLDs eran útiles para representar ecuaciones que requerían muchos términos productos. La desventaja es que soportaban pocas entradas y que todas ellas debían estar combinadas y usadas.

PLAs

Los PLAs (programmable logic arrays) aparecieron en el año 1975 y surgieron como solución al problema de los PROM. Estos también tenían un arreglo de ANDs y otro de ORs pero en este caso ambos eran programables. Así el arreglo de ANDs era independiente de la cantidad de entradas del dispositivo y el arreglo de ORs también era independiente tanto de las variables de entrada como de la cantidad de compuertas ANDs. Los PLAs no tuvieron un gran éxito de mercado y su mayor desventaja era el alto tiempo de propagación de las señales a lo largo de los enlaces programables en comparación con su contraparte de enlaces fijos. Como resultado, eran considerablemente mas lentos que los PROMs

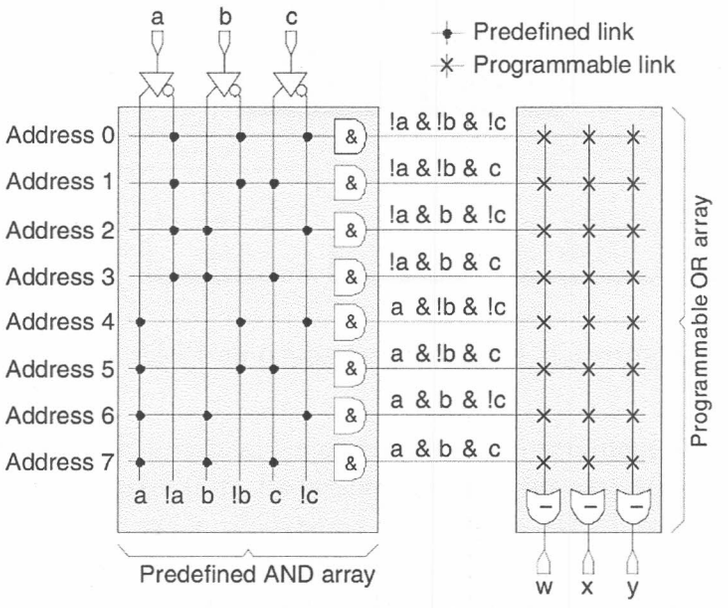


Figura 1.2: PLD PROM no programada

PALs y GALs

Estos SPLDs surgieron a finales de la década del '70 para abordar el problema de los PLAs. Conceptualmente son el opuesto a los PROMs ya que estos consisten en un arreglo de compuertas ANDs con enlaces programables y un arreglo de compuertas ORs con enlaces fijos. Comparados con los PLAs estos son mas rápidos debido a que poseen menor cantidad de enlaces programables por tener el arreglo de ORs con enlaces fijos, pero al mismo tiempo, esto los limita en la cantidad de términos que pueden sumar (OR) en las ecuaciones lógicas.

CPLDs

A comienzos de la década del '80 aparecieron los CPLDs. Con el avance de la tecnología CMOS, se lograron dispositivos de muy alta densidad y complejidad y bajo consumo. Así, la firma Altera produjo el chip llamado MegaPAL, que consistía en muchas PALs dentro del mismo chip con una matriz de interconexión que enlazaba el 100 % de las entradas y salidas de todas las PALs. La figura 1.3 muestra un diagrama conceptual de los CPLDs, puesto que cada fabricante los diseñaba según su propia arquitectura.

El costo de interconectar todos contra todos era alto y poco escalable, por lo tanto, la siguiente generación de CPLDs tenían una matriz de interco-

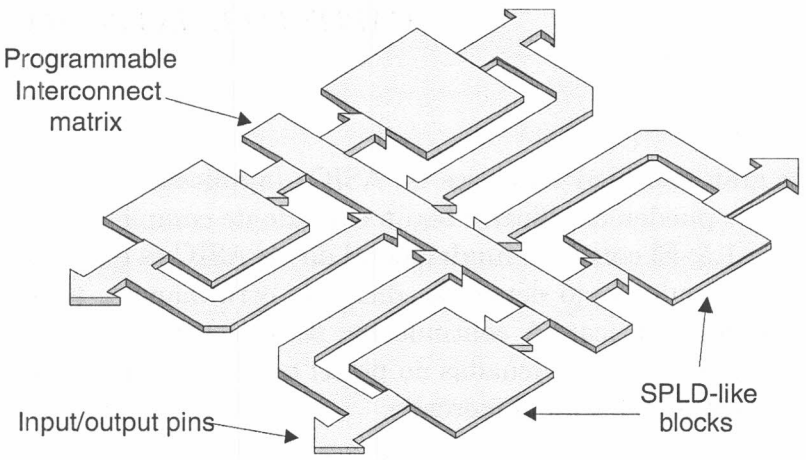


Figura 1.3: Estructura genérica de un CPLD

nexión programable que no llegaba a interconectar directamente al 100 % de los PALs. Digamos por ejemplo que una matriz de interconexión podría tener alrededor de 100 cables, mientras que los bloques de SPLDs manejan alrededor de 30 cables. Así, se intercalaba en el acceso a la matriz de interconexión, un multiplexor programable como lo muestra la figura 1.4. Esto incremento enormemente la complejidad del software de programación de los dispositivos pero mantuvo a estos últimos rápidos, escalables y de bajo consumo.

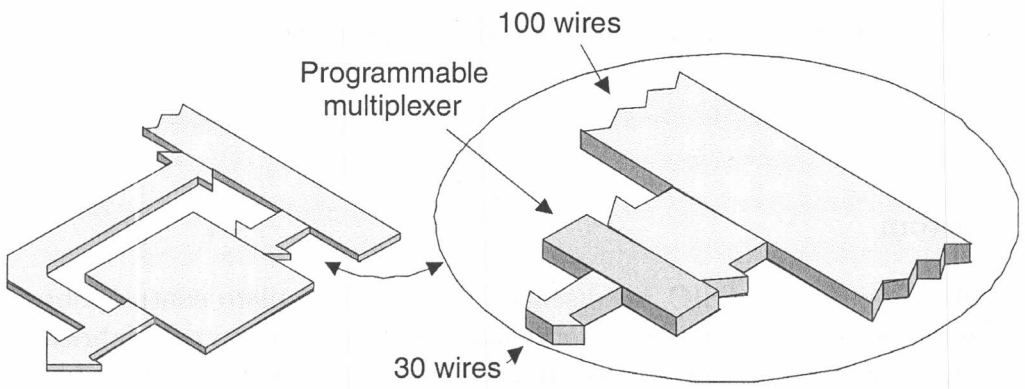


Figura 1.4: CPLD usando multiplexor programable

1.1.2. ASICs

Existe una gran cantidad de clases de ASICs (application specific integrated circuit) que pueden clasificarse según el grado de complejidad como lo muestra la figura 1.5. El concepto fundamental de los ASIC es que son circuitos integrados fabricados bajo demanda, dicho de otra manera, uno realiza el diseño y lo manda fabricar. A continuación se hará una breve reseña de las distintas clases de ASICs ordenadas no por el orden de complejidad, sino por su aparición cronológica en el mercado.

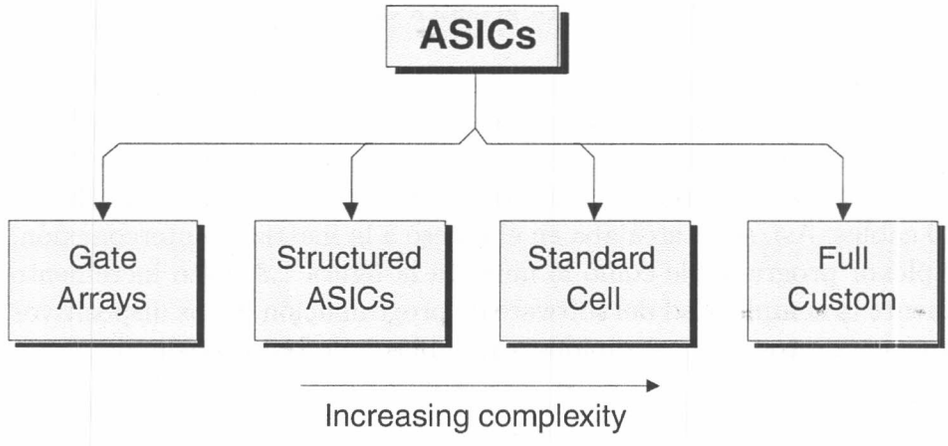


Figura 1.5: Clases de ASICs según su complejidad

Full Custom

En el diseño de este ASIC, los diseñadores tienen completo control sobre todas las capas de máscaras usadas sobre las superficie de silicio para fabricar el chip. El fabricante no provee ningún componente ni librerías predefinidas de compuertas lógicas. Así, el diseñador puede disponer las dimensiones de cada transistor, produciendo así funciones de mas alto nivel basándose en estos últimos. De esta manera, si se desean compuertas mas rápidas, se puede alterar las dimensiones de los transitores utilizados para construirla. El diseño de estos dispositivos es muy complejo, y posee tiempos muy grandes de diseño, pero el chip resultante contiene la mayor cantidad de compuertas lógicas con un mínimo desperdicio de silicio.

Gate Arrays

Los gate arrays están basados en la idea de *celdas básicas* que consisten en una colección de transistores y resistencias desconectados entre si. Se disponían estas celdas en arreglos separados por canales de interconexión dedicados. Los fabricantes definían un mapa de interconexión de las celdas básicas para construir funciones lógicas, multiplexores y registros. A estos bloques se los denomino celdas (distintas de las celdas básicas) y eran utilizados para construir diseños mas complejos. Así, cada fabricante soportaba distintos conjuntos de funciones y a estos se los llamaba *librería de celdas.* Los diseñadores entonces construían sus sistemas a partir de estas librerías y definían como estas celdas debían interconectarse entre si (netlist). Con software especializado se construía una máscara que era aplicada a la superficie de silicio para establecer las conexiones diseñadas. Estos ASICs ofrecían un bajo costo puesto que los transistores y otros componentes básicos estaban prefabricados y solo la máscara era ajustable por el diseñador. Las desventajas consistían en que la mayoría de los diseños dejaban muchos componentes internos sin ser utilizados, había restricciones en la ubicación física de las compuertas diseñadas y el ruteo interno no era óptimo. Estos factores impactaban negativamente en la performance y consumo de los diseños.

Standard Cell

A comienzos de la década del '80 y para solucionar algunas de las desventajas de los gate arrays, aparecieron los llamados Standard Cell. En este caso, los fabricantes también proporcionaban librerías de celdas que ofrecían a los diseñadores elementos tales como procesadores, funciones de comunicación y memorias RAM, pero estas no estaban basadas en las celdas básicas como eran con los gate arrays y no había ningún componente prefabricado en el chip. Así, mediante el uso de herramientas de software, el diseñador construía una red de interconexión de celdas que, también mediante software, eran mapeadas y ruteadas de manera óptima sobre la estructura de silicio. El concepto de celdas standard permitió que cada función lógica fuera creada usando el mínimo número de transistores y fueran posicionadas de manera tal de facilitar su interconexión.

Structure ASICs

Estos chips surgieron a principios de la década del '90 y retomaron la idea de los gate arrays pero cambiando las celdas básicas por estructuras mucho mas complejas que llamaron módulos. Estos módulos podía contener, según el fabricante, una mezcla de funciones lógicas genéricas (implementadas con

compuertas, multiplexores o tablas de busqueda *lookup tables*), uno o mas registros y posiblemente algo de memoria RAM local. Así, un arreglo de estos módulos prefabricados eran dispuestos sobre la superficie de silicio y el diseño consistía en una máscara que los interconectaba. Pero, debido a la gran sofisticación de estos módulos, las máscaras en general también están prefabricadas reduciendo así los costos del diseño y el tiempo de desarrollo.

1.2. FPGAs

El panorama tecnológico a principios de la década del '80 mostraba por un lado dispositivos programables como los CPLDs y SPLDs que eran altamente configurables y tenían tiempos de desarrollo y modificación muy rápidos, pero no podían soportar funciones muy grandes o complejas.

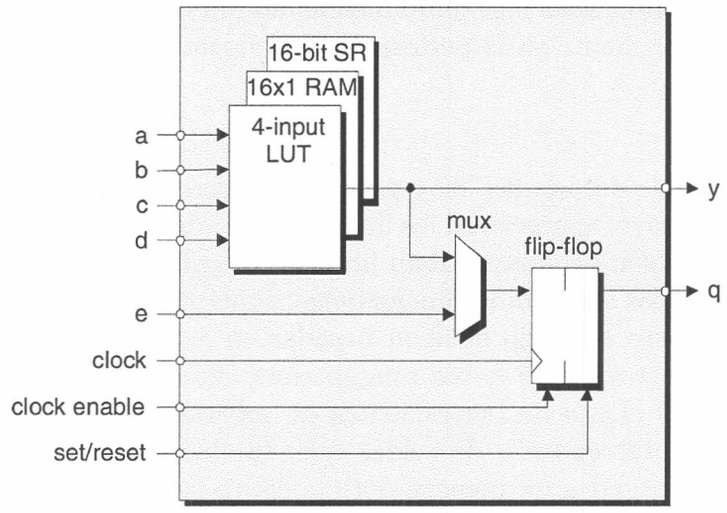


Figura 1.6: Estructura genérica de una celda lógica de Xilinx

Por otro lado estaban los ASICs que soportaban diseños muy grandes y complejos pero con excesivos costos de fabricación y diseño que una vez implementados, no podían modificarse (a menos que se fabricaran nuevamente). Era claro entonces que alguna tecnología intermedia debía aparecer para acortar la brecha entre los PLDs y los ASICs. A mediados de los '80 la empresa Xilinx desarrolló un circuito integrado que llamó FPGA (field-programmable gate array) con la intención de rellenar este hueco (un dispositivo que permita el desarrollo de grandes sistemas y con bajos costos de diseño y fabricación).

Actualmente, la estructura genérica de un FPGA de Xilinx consiste en una unidad básica llamada celda lógica LC (logic cell). Esta contiene princi-

palmente una LUT de 4 entradas (que puede ser utilizada como una RAM de 16×1 bits, o como un desplazador de registro), un multiplexor y un registro (Fig. 1.6)

El siguiente paso en la jerarquía se llama *slice* y básicamente consiste en dos celdas lógicas que comparten la señal de reloj (clock), habilitación del reloj (clock enable) y reset. Dando un paso mas, están los bloque lógicos configurables *CLB* que pueden contener 2 o 4 slices (Fig. 1.7)

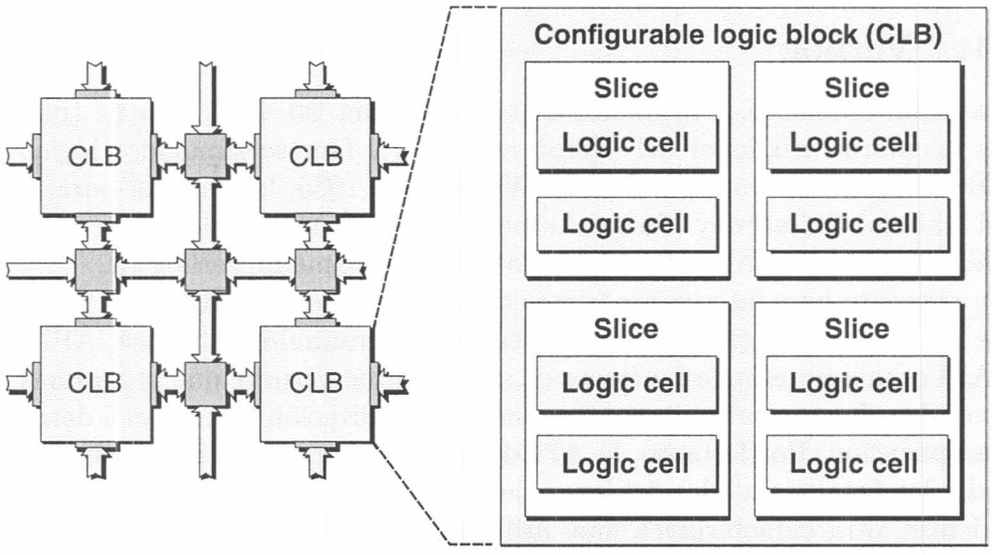


Figura 1.7: Bloque de lógica configurable de 4 slices

La razón por la cual se tiene esta jerarquía de bloques, $LC \rightarrow$ Slice (con dos LC) $\rightarrow$ CLB (con cuatro slices) es que existe una jerarquía comparable en la interconexión. Así, la interconexión de celdas lógicas dentro de un slice es más rápida que la interconexión de slice dentro de un CLB y la interconexión de estos últimos es aún más lenta. La idea es alcanzar un balance óptimo entre hacer fácil las conexiones y no incurrir en grandes demoras en las mismas.

Finalmente un FPGA consiste en una gran arreglo de CLBs interconectados entre sus vecinos, además de árboles de propagación del reloj global. También pueden contener bloques de memoria Ram (BRAMs) y estructuras complejas como multiplicadores y MACs (multiplicar y acumular) dependiendo de los modelos.

1.3. Lenguajes de descripción de Hardware

Los FPGA son programados y configurados utilizando un lenguaje de descripción de hardware (HDL). Existen muchos y variados HDLs propuestos por varias empresas y organizaciones. A continuación se hará un breve raconto histórico a modo de introducción de dos de los HDLs más extendidos y utilizados en la actualidad.

1.3.1. Verilog

En algún momento, a mediados de la década del '80, Phil Moorby (uno de los miembros originales del equipo que creó el famoso simulador lógico HILO) diseñó un nuevo HDL llamado Verilog. En 1985, la empresa para la que él trabajaba, Gateway Design Automation, presentó este lenguaje en el mercado junto con otro simulador lógico accesorio llamado Verilog-XL.

Un concepto muy interesante propio de Verilog y Verilog-XL era su interfaz de lenguaje de programación (application programming interface, API). Una API es una librería de funciones de software que permite que programas externos de software transmitan datos hacia una aplicación y accedan a datos de esa aplicación. Por lo tanto, la API de Verilog les permite a los usuarios extender las funcionalidades del lenguaje y el simulador Verilog.

Además, otra característica muy útil asociada con Verilog y Verilog-XL era la capacidad para contar con información temporal especificada en un archivo externo de texto conocido como formato estándar de demora (standard delay format, SDF). Esto permitía que herramientas como los paquetes de análisis de demoras post-place-and-route generen archivos SDF que el simulador puede utilizar para brindar resultados más precisos. Como lenguaje, el Verilog original era razonablemente poderoso en el nivel abstracción estructural (interruptores y compuertas) especialmente en lo concerniente a la capacidad de modelado de demoras; era muy poderoso en el nivel de abstracción funcional (ecuaciones Booleanas y RTL); y admitía algunas construcciones de comportamiento (algorítmicas)

En 1989, Gateway Design Automation, junto con el HDL Verilog y el simulador Verilog-XL fueron adquiridos por Cadence Design Systems. La posibilidad en aquel momento era que Verilog siguiera siendo sólo otro HDL propietario. Sin embargo, en una operación que tomó a la industria por sorpresa, Cadence abrió las especificaciones del HDL Verilog, la PLI Verilog y la SDF Verilog al dominio público en 1990.

Ésta era una operación muy valiente porque implicaba que cualquier usuario podía desarrollar un simulador Verilog y en consecuencia convertirse en un competidor potencial de Cadence. La razón de esta actitud tan generosa

de Cadence era que el lenguaje VHDL (que se presenta más adelante en esta sección) comenzaba a ganar una cantidad significativa de seguidores. El lado positivo de abrir Verilog al dominio público era que una amplia variedad de empresas que desarrollaban herramientas basadas en HDL, como aplicaciones de síntesis de lógica, a partir de ese momento aceptaron Verilog como lenguaje favorito.

Tener una representación de diseño único que puede utilizarse en simulaciones, síntesis y otras herramientas simplificó en gran medida la labor de los usuarios. Sin embargo, es importante recordar que Verilog se concibió originalmente pensando en la simulación como objetivo; las aplicaciones como la síntesis fueron una idea posterior. Esto significa que al crear una representación en Verilog para utilizarse para simulación y síntesis, el usuario se ve restringido a utilizar un subconjunto sintetizable del lenguaje.

La definición formal de Verilog está descripta en un documento conocido como manual de referencia del lenguaje (language reference manual, LRM), que detalla la sintaxis y la semántica del lenguaje. Verilog se volvió muy popular rápidamente. El problema era que distintas empresas comenzaron a extender el lenguaje en direcciones diferentes. Para limitar esta situación, se estableció en 1991 una corporación sin fines de lucro llamada Open Verilog International (OVI). Formada por representantes de todos los mayores proveedores de la época, el objetivo de la OVI era administrar y estandarizar el HDL Verilog.

La popularidad de Verilog continuó creciendo exponencialmente, y como resultado la OVI finalmente le solicitó a la IEEE que formara un comité de trabajo para establecer a Verilog como un estándar IEEE. Conocido como IEEE 1364, este comité se formó en 1993. En mayo de 1995, se emitió la primera versión oficial IEEE de Verilog, que se conoce formalmente como IEEE 1364-1995, y cuya denominación no oficial fue Verilog 95.

1.3.2. VHDL

En 1980, el Ministerio de Defensa de los Estados Unidos lanzó el programa de circuitos integrados de muy alta velocidad (very high speed integrated circuits, VHSIC), cuyo principal objetivo era el de llevar a un nivel más avanzado las tecnologías más recientes de circuitos integrados digitales. Este programa buscaba resolver, entre otros problemas, el hecho de que era difícil reproducir los circuitos integrados (y las placas de circuitos) durante el largo ciclo de vida útil de los equipos militares porque la función de cada pieza no estaba documentada en forma rigurosa. Más aún, los diferentes componentes que formaban un sistema a menudo se diseñaban y verificaban mediante lenguajes de simulación y herramientas de diseño diversas e incompatibles.

Para resolver estos problemas, en 1981 se lanzó un proyecto para desarrollar un nuevo lenguaje de descripción de hardware llamado VHSIC HDL (o, en forma abreviada, VHDL). Una característica única de este proceso era que se le dió participación a la industria desde las etapas iniciales. En 1983, un equipo compuesto por Intermetrics, IBM y Texas Instruments ganó un contrato para desarrollar el VHDL, cuya primera versión oficial se emitió en 1985. También es interesante el hecho de que, para estimular la aceptación de la industria, posteriormente el Ministerio de Defensa donó todos los derechos de la definición del lenguaje VHDL a la IEEE en 1986. Luego de realizarle algunas modificaciones para resolver algunos problemas ya conocidos, el VHDL se emitió como estándar IEEE 1076 oficial en 1987. El lenguaje se extendió aún más en una versión de 1993 y otra vez en 1999. Como lenguaje, el VHDL es muy poderoso en el nivel funcional (ecuaciones Booleanas y RTL) y de comportamiento (algorítmico) de abstracción, y también admite algunas construcciones de diseño a nivel de sistema. Sin embargo, VHDL es un poco débil en el nivel estructural (interruptores y compuertas) de abstracción, especialmente en lo concerniente a la capacidad de modelado de demoras. Rápidamente se advirtió que el VHDL tenía una precisión temporal insuficiente como para ser utilizado como simulador. Por esta razón, en la Design Automation Conference (DAC) de 1992 se lanzó la iniciativa VITAL. El resultado final combinaba una librería de funciones primitivas de ASIC y FPGA y un método asociado para realizar anotaciones posteriores de información de las demoras. En estos modelos de librería, este mecanismo de demora se basaba en el mismo formato tabular subyacente utilizado por Verilog.

Verilog	**VHDL**
Relativamente fácil de aprender	Relativamente difícil de aprender
Tipo de Datos fijo	Tipo de Datos abstracto
Construcciones Interpretadas	Construcciones Compiladas
Buen manejo temporal de compuertas	No tan buen manejo temporal de compuertas
Limitado reuso de diseños	Buen reuso de diseños
Sin replicación de estructuras	Soporta replicación de estructuras

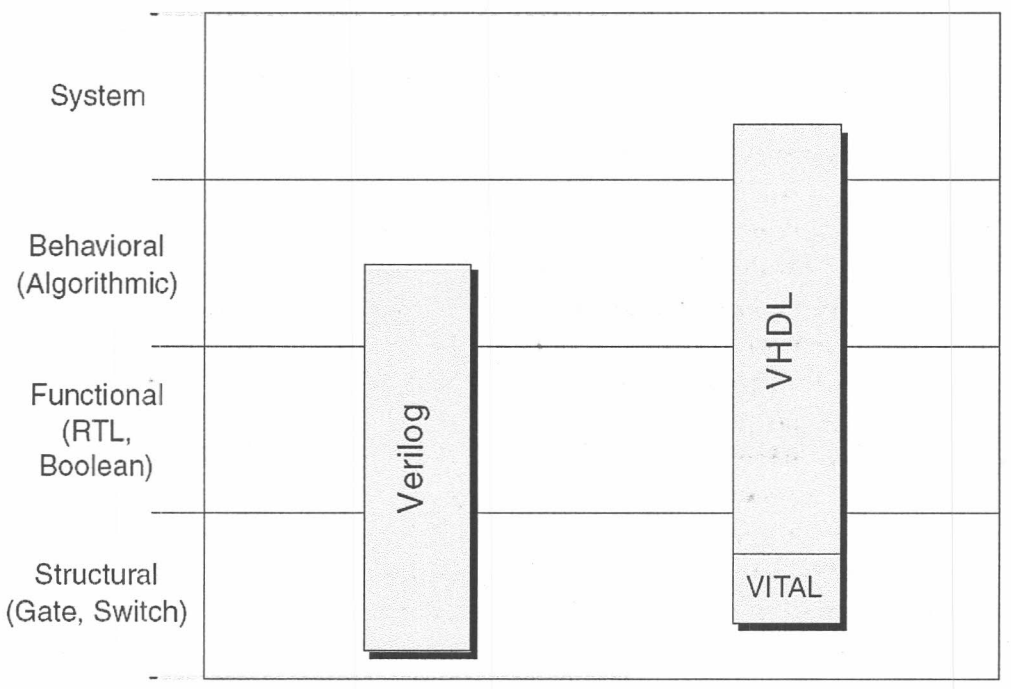


Figura 1.8: Diagrama comparativo de Verilog y VHDL

Capítulo 2

Algoritmo de detección de bordes ADM

La detección de bordes es una herramienta fundamental en las aplicaciones de procesamiento de imágenes debido a que caracterizan los límites de los objetos y permiten los procesos posteriores de segmentación e identificación y reconocimiento de los mismos. También el método es utilizado para mejorar la apariencia de objetos fuera de foco o borrosos. Otro de sus beneficios es que reduce la cantidad de datos de las imágenes, preservando únicamente la información útil.

Un borde dentro de una imagen es una zona que presenta un fuerte contraste en intensidad de un pixel a otro. Los metodos para detectarlos se basan en aproximar el método del gradiente (método de Sobel) o de la segunda derivada (Laplaciano) a los casos de imágenes (2 dimensiones). La idea es que, si hay un pixel que pertenece a un borde, el valor de su derivada excederá un cierto valor (sera un máximo). En el caso del laplaciano, si el pixel pertenece al borde, su segunda derivada sera cero.

El método de máscara de absolutas diferencias (ADM) propuesto por Alzahrani y Tom Chen [3], detecta bordes utilizando el método del gradiente. Nosotros lo hemos elegido para implementar pues es un algoritmo que requiere bajo costo computacional en relación a los algoritmos preexistentes. En este método, se asume que los bordes son pixels con un gradiente alto. La magnitud del gradiente es la intensidad del borde. El ángulo del gradiente, es la dirección del borde. Por lo tanto un pixel perteneciente al borde se define usando estas dos características: intensidad y dirección. El algoritmo sigue tres pasos. Durante el primer paso, se aplica un filtro semi-gaussiano para eliminar ruidos. Durante el segundo paso, se utiliza la máscara de absolutas diferencias para calcular en cada pixel la intensidad y la dirección del borde. Durante el ultimo paso, se produce el mapa de los bordes detectados.

2.1. Representación de bordes

Un borde ideal es aquel que presenta un cambio abrupto en la escala de grises. Sin embargo este no es el caso mas común en la práctica debido a varios factores tales como la naturaleza de la escena, reflejos, ruido, etc. Así pues, un borde es considerado como una cambio lineal o no lineal que puede ser abrupto o suave en la escala de grises. De esta manera, cada pixel de la imagen puede ser pensado como un pixel de borde midiendo la intensidad de borde.

Observando el caso de la Figura 2.1(a), que representa un borde en una imagen de $1D$. Se asume que el cambio en el tono de gris es de naturaleza lineal, también asumimos que el cambio se mide desde el centro del pixel $p_{(i)}$ al centro del pixel $p_{(i-k)}$. La altura del borde h_b, se define como la diferencia absoluta del tono de gris entre $p_{(i)}$ y $p_{(i-k)}$. El ancho del borde, w_b se define como el número de pixels a través de los cuales el cambio tiene lugar.

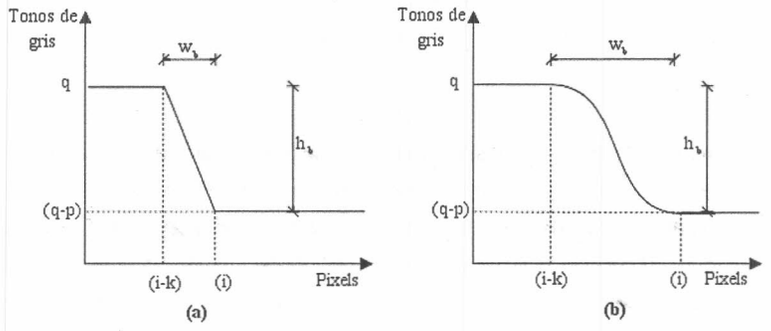


Figura 2.1: Representación de bordes: (a) Lineal; (b) No lineal

En la Figura 2.1(a), se ve como la intensidad del borde es función de su ancho y su alto. Cuanto más alto y más angosto, más intenso es el borde. Entonces, la intensidad de un borde lineal puede ser definida como:

$$S_{lb} = \left| \frac{h_b}{w_b} \right| \tag{2.1}$$

Mas precisamente, la intensidad del borde en el pixel j puede definirse como:

$$S_{lb} = \left| g_{(j+1)} - g_{(j-1)} \right| \tag{2.2}$$

donde $j = i-1, \ldots, i-k+1$ y g_j es el valor del tono de gris del pixel p_j. El lado derecho de ambas ecuaciones 2.1 y 2.2 son siempre positivos ya que la

intensidad de un borde es independiente de la dirección. Si el cambio en el tono de gris no es de naturaleza lineal, como se muestra en la Figura 2.1(b), entonces el cambio no sera uniforme. Como resultado de esto, la intensidad del borde sera diferente de un punto a otro a lo largo del borde. Esta es una representación mas realista de los bordes, especialmente si se ha efectuado un proceso de suavizado a la imagen antes de realizar la detección de bordes. El suavizado, en general, tiende a degradar los bordes lineales produciendo que estos pierdan su linealidad. Este efecto depende del diseño de la máscara de suavizado y de la naturaleza del borde antes de ser suavizado.

Considerando el caso de la Figura 2.1(b). Es obvio que el cambio en el tono de gris a lo largo del borde no es lineal. Entonces, la intensidad de un borde no lineal esta asociado al máximo cambio del tono de gris entre dos pixels vecinos a lo largo de borde. La ecuación 2.2 es utilizada para encontrar la intensidad de cada pixel j a lo largo del borde no lineal.

Si un borde no lineal ocurre entre los centros de los pixels $p_{(i)}$ y $p_{(i-k)}$, la intensidad del borde, S_{nb}, puede ser definida como:

$$S_{nb} = max[|S_{nb}(j)|] \tag{2.3}$$

donde $j = i-1, \ldots, i-k+1$

2.2. Proceso de suavizado

La técnica de suavizado de la imagen se aplica antes del proceso de detección de bordes con el objetivo de eliminar el ruido presente en la imagen. Este proceso consiste en ponderar los pixels de la imagen con sus vecinos, lo cual tiene el efecto de suprimir valores de alta frecuencia produciendo un suavizado en los bordes.

Si bien hay varias técnicas posibles para el suavizado, (media, mediana) la mayoría de las técnicas empleadas para detección de bordes utilizan los filtros gaussianos que producen una media ponderada de cada pixel dando mayor peso a los pixels mas cercanos al centro. De esta manera, se preservan mas los bordes que en el caso de otras técnicas que pesan uniformemente a todos los pixels.

El proceso de suavizado semi-gaussiano se aplica a los datos mediante una máscara de 5×5 elementos, donde el peso w_p de cada pixel se define como:

$$w_p = \frac{1}{d_p} \tag{2.4}$$

siendo d_p la distancia entre los pixels p y el centro. La distancia entre pixels vecinos horizontal y verticalmente es 1. La distancia entre pixels vecinos

diagonales ($p_{i,j}$ y $p_{i+1,j+1}$) es $\sqrt{2}$. De la misma manera, la distancia entre $p_{i,j}$ y $p_{i+2,j+1}$ es $\sqrt{5}$, y así sucesivamente. En el cuadro 2.1(a) se muestra la máscara del filtro semi-gaussiano.

0.353	0.447	0.5	0.447	0.353
0.447	0.707	1	0.707	0.477
0.5	1	2	1	0.5
0.447	0.707	1	0.707	0.477
0.353	0.447	0.5	0.447	0.353

(a)

Pd_{u_2}		V_{u_2}		Nd_{u_2}
	Pd_{u_1}	V_{u_1}	Nd_{u_1}	
H_{l_2}	H_{l_1}	$p_{(i,j)}$	H_{r_1}	H_{R_2}
	Nd_{l_1}	V_{l_1}	Pd_{l_1}	
Nd_{l_2}		V_{l_2}		Pd_{l_2}

(b)

Cuadro 2.1: Máscaras (a) semi-gaussiana, (b) deferencias absolutas

2.3. Intensidad y dirección de bordes

Para calcular la intensidad y la dirección de los bordes, el algoritmo ADM utiliza la imagen resultado de aplicar la máscara de suavizado y luego, para cada pixel, calcula la intensidad y la dirección.

Para calcular la intensidad se utilizan 4 pixels, la máscara utilizada para calcular la diferencia absolutas se muestra en el cuadro 2.1(b). Esta, esta centrada en el pixel de interés $p_{(i,j)}$ al que sera asignado la intensidad y dirección del borde. La intensidad de $p_{(i,j)}$ es medida en 4 direcciones, y la mas grande sera asignada al pixel.

1. Cálculo preliminar:

$$\begin{aligned} V_u &= V_{u_1}+V_{u_2} \\ V_l &= V_{l_1}+V_{l_2} \\ H_r &= H_{r_1}+H_{r_2} \\ H_l &= H_{l_1}+H_{l_2} \end{aligned} \qquad \begin{aligned} Pd_u &= Pd_{u_1}+Pd_{u_2} \\ Pd_l &= Pd_{l_1}+Pd_{l_2} \\ Nd_u &= Nd_{u_1}+Nd_{u_2} \\ Nd_l &= Nd_{l_1}+Nd_{l_2} \end{aligned}$$

2. Cálculo de todas las diferencia absolutas:

$$V = |V_u - V_l| \qquad Pd = |Pd_u - Pd_l|$$
$$H = |H_r - H_l| \qquad Nd = |Nd_u - Nd_l|$$

3. Cálculo de intensidad y dirección:

$$S_b = \frac{\text{máx}\{V,H,Pd,Nd\}}{2}$$

$$dir_b = dir(\text{mín}\{V, H, Pd, Nd\})$$

donde $dir(Nd) = 0, dir(V) = 1, dir(Pd) = 2, dir(H) = 3$

La dirección de borde es elegida como la más pequeña de las diferencias absolutas ya que generalmente la dirección de borde ocurre paralela a la misma. Para ilustrar este ejemplo, asumimos la presencia de un borde horizontal donde la diferencia en el tono de gris es de 100, como se muestra en la figura 2.2. El resultado de calcular las diferencias absolutas en las 4 direcciones es: $Nd = 200$, $V = 200$, $Pd = 200$, $h = 0$. Esto indica que el borde en general toma la misma dirección que el más pequeño de los valores de la diferencias absolutas. El mismo principio puede aplicarse para el resto de las direcciones.

100	100	100	100	100	100	100
100	100	100	100	100	100	100
100	100	100	100	100	100	100
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Figura 2.2: Ejemplo de calculo de dirección

Una vez que estos tres pasos fueron completados, el borde en $p_{(i,j)}$ sera representado por dos números que corresponden a la intensidad y la dirección. Por ejemplo $(100, 3)$ significa un borde horizontal de intensidad 100.

2.4. Detección y localización de bordes

En la sección anterior se mostró que los valores de intensidad a través del borde varía según la naturaleza del cambio de los niveles de gris. Además,

todos los bordes lineales perdieron su linealidad después de haberse aplicado el filtro de suavizado. Esto genera distintos valores de intensidad a lo ancho del borde, con un máximo local. Cada uno de estos valores tienen asignados una dirección, ambos datos son utilizados para determinar la ubicación del borde.

Un máximo local en los tonos de grises determina entonces la presencia de un borde. Este se detecta comparando la intensidad del pixel en análisis con la de sus vecinos en la dirección normal al borde.

$P1$	$P2$	$P3$
$P4$	$P5$	$P6$
$P7$	$P8$	$P9$

(a)

$dir(P5)$	Vecinos
Dn	$P3, P7$
V	$P4, P6$
Dp	$P1, P9$
H	$P2, P8$

(b)

Cuadro 2.2: (a) Mascara de pixels, (b) Vecinos según dirección del borde

Así, un pixel sera iluminado si su intensidad es mayor a la de sus vecinos correspondientes según la dirección del borde y además si su intensidad supera un umbral predeterminado.

Capítulo 3

Arquitectura

Al momento de comenzar este trabajo, que incluye la implementación de un algoritmo en hardware, se contaba con el Kit de desarrollo de Digilent *Spartan 3 Started Kit Board* donado por la empresa Xilinx como parte del programa de universidades [13]. En consecuencia, todo el desarrollo del presente trabajo esta basado en dicha placa. Las figuras 3.1 y 3.2 muestran respectivamente la vista superior e inferior de dicha placa. Además del FPGA Spartan 3, la placa dispone de otros dispositivos y puertos de comunicación. A continuación se enumeran aquellos que serán utilizados a lo largo de este trabajo. Para mayor profundidad sobre el contenido del Kit referirse a [11]

3.1. Kit de desarrollo

1 Xilinx Spartan-3 XC3S200 FPGA device (XC3S2OOFT256)

2 2M-bit Xilinx XCF02S flash PROM de configuración

4 Dos 256K-by-16 dispositivos SRAM asincronicos (ISSI IS61LV25616AL-lOT).

6 puerto serial RS-232

7 adaptador de voltaje RS-232-FPGA

8 Segundo puerto serial RS-232

10 Cuatro-dígitos displays siete-segmentos

11 Ocho switches desplazantes

12 Ocho salidas de LED

13 Cuatro botones de contacto

14 Oscilador a cristal de 50-MHz para fuente de reloj principal

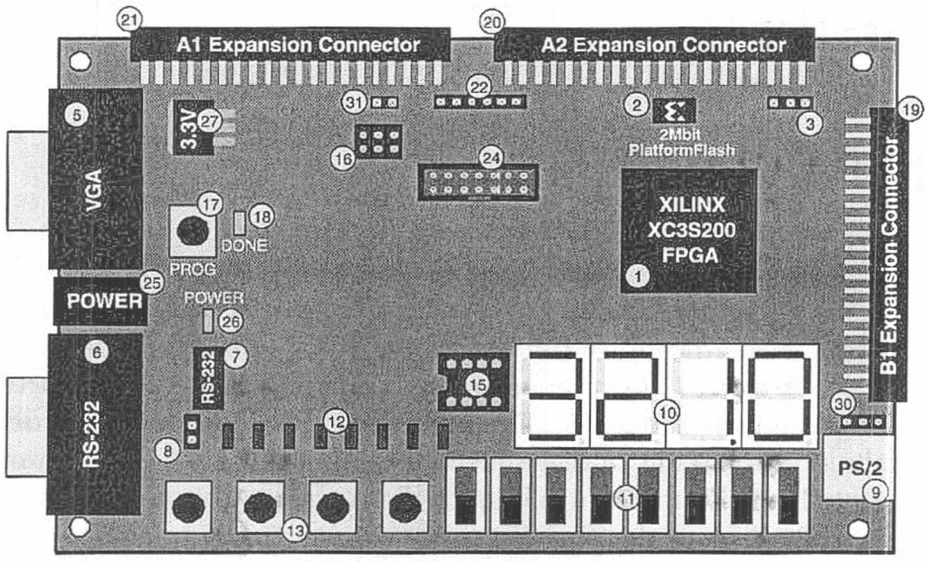


Figura 3.1: Vista superior de la placa Starter Kit board de Digilent

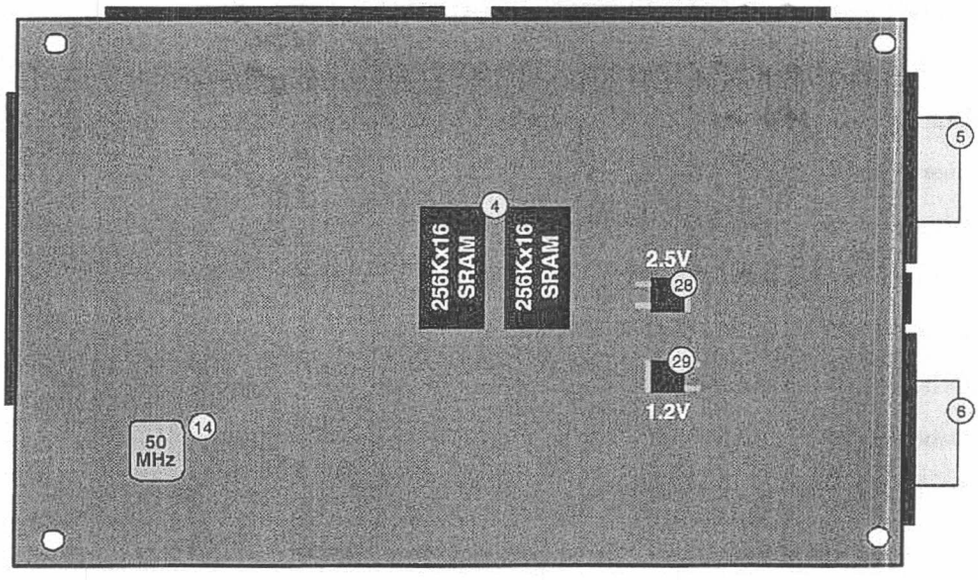


Figura 3.2: Vista inferior de la placa Starter Kit board de Digilent

3.1.1. Entrada y salida

La idea de esta implementación es poder detectar bordes utilizando el algoritmo ADM en una imagen procedente del exterior. El kit de Spartan 3 utilizado provee dos medios de comunicación con el exterior, un puerto serie y buses de expansión. Los buses de expansión tienen la ventaja de ser de alta velocidad, idoneo para la transferencia de imágenes, pero implica diseñar una interfaz electrónica por ejemplo para comunicarse con la PC. En consecuencia se decidió utilizar el puerto serial que es estándar en las PC, de fácil implementación y no afecta a los resultados buscados por esta tesis ya que la adquisición de la imagen en un medio real de uso puede variar ampliamente. El diseño de este módulo se vera en el Capitulo A

3.1.2. Memoria

Como se vio anteriormente en este capitulo, la placa dispone de dos chips de memoria SRAM en donde será almacenada la imagen proveniente de la PC a través del puerto serie. Para un sistema sincrónico es difícil acceder a una SRAM asincrónica directamente, en general se utiliza un controlador de memoria que toma los comandos del sistema principal sicrónico y genera las señales apropiadas asíncronas. El controlador de memoria entonces aisla el sistema principal de la problemática temporal de las señales haciendolas parecer sincrónicas. El desempeño de un controlador de memoria se evalúa según la cantidad de accesos que puede completar en un periodo de tiempo. Diseñar un controlador de memoria simple es bastante sencillo pero, alcanzar un buen rendimiento requiere ajustar con precisión las señales en el tiempo. Una operación de lectura o escritura requiere que los datos, dirección y señales de control sean asignadas en un orden específico y se mantengan estables durante un determinado periodo de tiempo.

En el Capitulo 4 se diseñará este módulo que en principio dispondrá de entradas independientes para la lectura y escritura de la memoria.

3.1.3. Accesorios

Además del diseño específico del algoritmo y de los módulos de memoria y entrada y salida, nos interesa también medir tiempos de procesamiento. Para esto, se generará un módulo encargado de contar los ciclos de reloj utilizados en la etapa de procesamiento y mostrar este resultado en los displays de siete segmentos que dispone la placa (10). El algoritmo que se pretende implementar posee algunos parametros ajustables que regulan el comportamiento del mismo, por ejemplo, el valor del umbral que sera considerado borde. Estos

parametros y otros serán ajustados mediante los switch deslizantes (11) que posee el Kit. También se hará uso de los leds (12), para indicar etapas del proceso (transferencia de imagen, filtrado de la imagen, etc)

3.2. Arquitectura

En lineas generales todo el mecanismo de filtrado de la imagen consta de tres etapas.

Inicialmente se recibirá la imagen desde el puerto serie y será almacenada en la memoria principal. Entonces, el módulo de entrada y salida estará conectado al pin *RX* del puerto serie y al módulo de administración de memoria (ver Fig 3.3). También deberá implementar algún protocolo de comunicación que permita identificar el comienzo y fin de una imagen durante la transmisión. La segunda etapa es el proceso de filtrado propiamente dicho. Este módulo estará conectado al administrador de memoria puesto que deberá cargar de esta, la imagen que deberá procesar y almacenar la imagen resultado. Finalmente se deberá transmitir la imagen resultado desde la memoria principal a la PC mediante el puerto serie y el módulo de entrada y salida.

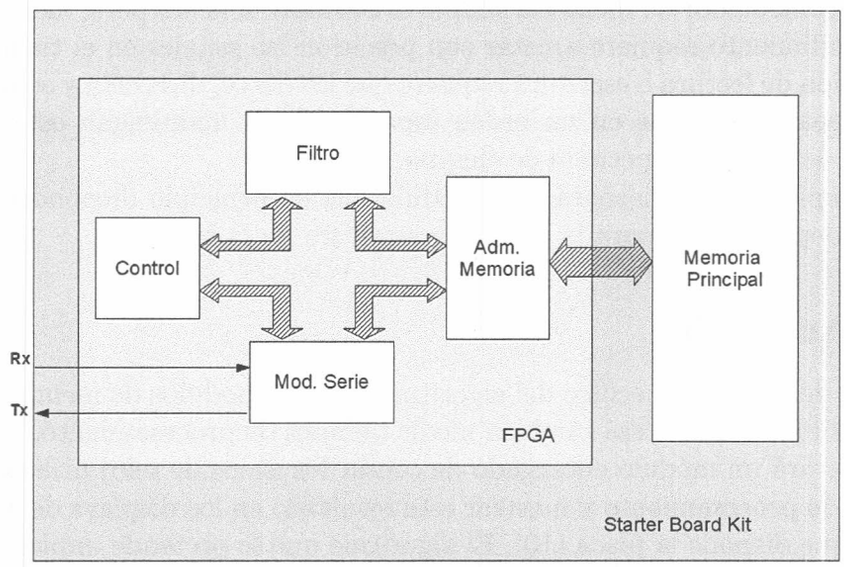


Figura 3.3: Estructura general de la implementación

El módulo de control será el encargado de coordinar cada una de estas

etapas. Así, durante la primera fase, el módulo de entrada y salida estará conectado al módulo de administración de memoria, mientras que en la segunda etapa lo estará el módulo de filtrado, manteniendo así la exclusión mutua de los módulos al administrador de memoria. Este módulo también sera el encargado de medir los tiempos de proceso y de mostrarlos en los displays.

Cada uno de los módulos mencionados podrán estar compuestos por submódulos interconectados que, en conjunto implementaran la funcionalidad completa. Cada una de estas unidades funcionales tiene una interfase, que consiste en un conjunto de señales de entrada y salida a través de las cuales se comunicará con otros módulos. En los capítulos que continúan, se desarrollará el diseño de los módulos presentados en este capítulo. Para cada uno de los módulos se dará su interfase y la descripción de su funcionalidad. Para la especificación del comportamiento se ha decidido adoptar el patrón de diseño que separa el camino de datos *(data path)* del camino de control *(control path)*. Esta metodología consiste en diseñar por un lado la lógica combinacional encargada del tratamiento de los datos y por otro lado, la lógica secuencial que determina el control del camino de datos [2]. Como regla general, podemos decir que el camino de datos implementa todas las posibles operaciones aplicadas a los datos (operaciones aritméticas, bifurcaciones y operaciones de bits.) y el camino de control determinará qué operaciones serán aplicadas a qué datos y en qué secuencia. El camino de datos podría ser un pipeline de 1 o mas etapas y el camino de control será casi siempre representado por una máquina de estados finitos que, dependiendo de las entradas determinará el tratamiento que se hará a los datos.

3.2.1. Máquinas de estado

Para la definición del camino de control de los módulos se utilizaran máquinas de estado finito. Existe una muy amplia bibliografía que trata en profundidad el tema de las máquinas de estado finito, a continuación se hará una breve referencia.

Las máquinas secuenciales son un tipo de autómata capaces de generar, a partir de una palabra de entrada, y transitando por los estados que la componen, una palabra de salida [9]. Según su salida se definen dos tipos: máquina secuencial de Mealy, donde las salidas dependen del estado y de las entradas y máquina secuencial de Moore, donde las salidas dependen únicamente del estado.

Formalmente se definen como una quíntupla

$$(\Sigma_E, \Sigma_S, Q, f, g)$$

donde:

- Σ_E es el alfabeto de símbolos de entrada
- Σ_S es el alfabeto de símbolos de salida
- Q es el conjunto finito no vacío de estados
- f es la función de transición definida como:

$$f : Q \times \Sigma_E \rightarrow Q$$

- g es la función de salida definida como:

  Salida de Mealy $g : Q \times \Sigma_E \rightarrow \Sigma_S$

  Salida de Moore $g : Q \rightarrow \Sigma_S$

El alfabeto de entrada entonces será la combinación de algunas de las señales de entrada y señales internas de control, mientras que el alfabeto de salida se corresponderá con las señales que controla el camino de datos y eventuales señales de salida que se deseen propagar a otros módulos.

Cabe destacar que toda máquina de Mealy se puede transformar en una máquina equivalente de Moore y viceversa. A lo largo del diseño se utilizara una u otra según resulte mas cómodo el modelado.

Por ejemplo, supongamos que necesitemos dividir por dos la frecuencia de reloj de un sistema y queramos hacerlo con una máquina de estado. Además queremos que este divisor funcione solo cuando esta habilitado. Definimos entonces el alfabeto de entrada como los valores de las señales *en* y *clk* que respectivamente representan la señal de habilitación y el reloj del sistema que queremos dividir.

Así el alfabeto de entrada sera $\Sigma_E = \{00, 01, 10, 11\}$ y el de salida $\Sigma_S = \{0, 1\}$ formado únicamente por la señal de salida *salida*. Definimos entonces el conjunto de estados como $Q = \{\text{idle}, \text{uno}, \text{cero}\}$. La función de transición f y la función de salida g se dan en la siguiente tabla:

$f : Q \times \Sigma_E \rightarrow Q$	**en=0 clk=0**	**en=0 clk=1**	**en=1 clk=0**	**en=1 clk=1**	$g : Q \rightarrow \Sigma_S$
idle	idle	idle	uno	uno	salida=0
uno	uno	cero	uno	cero	salida=1
cero	cero	idle	cero	uno	salida=0

En nuestro caso, las máquinas de estado siempre serán sincrónicas, es decir, las transiciones siempre se harán en un flanco del reloj y las salidas

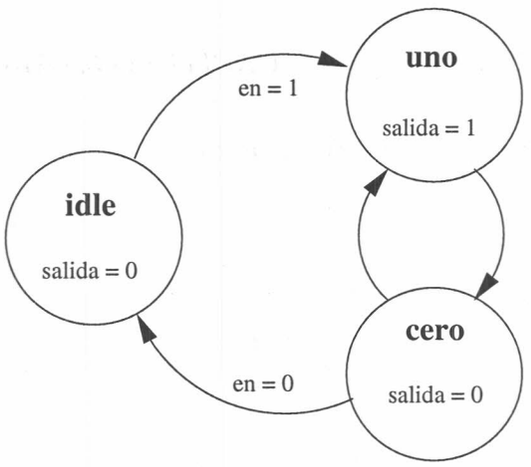


Figura 3.4: Representación de una máquina secuencial

serán registradas. Así, la figura 3.4 muestra una representación gráfica de la máquina de estado, en donde se suprimieron, para simplificar la gráfica, la señal de reloj y las transiciones de un estado a si mismo dadas por la señal de reloj. Los arcos marcados con una condición de entrada significan que se transitará por el arco si la entrada es la indicada. Los arcos sin condición serán transitados en el flanco de reloj, sin importar las entradas. Y en cada estado se marcan las salidas que cambiaron en referencia al estado anterior. En general, esta convención gráfica de representación de máquinas de estado de Moore será la que se utilizará a lo largo de este trabajo.

3.2.2. Pipelines

Los pipelines funcionan como una linea de ensamble, donde en cada etapa se calcula un resultado parcial que depende del resultado de la etapa anterior.

Un ejemplo del funcionamiento de un pipeline, se puede ver en la estructura de una sumatoria. Supongamos que necesitamos obtener la suma de 4 números (a, b, c, d) Puesto que el operador $+$ es binario, una manera de realizar esta operación involucra realizar 3 sumas, la primera $(a+b)$, la segunda $(c+d)$ y la tercera suma seria $(a+b)+(c+d)$ como lo muestra la figura 3.5(a)

Ahora supongamos que nuestro sumador tarda un tiempo t_+ para propagar el resultado a su salida. Entonces el esquema de la figura 3.5(a) demoraría $2t_+$ en darnos el resultado final. Este tiempo resulta de ejecutar la primer y la segunda suma en paralelo en un tiempo t_+ y luego la tercer suma (sumar ambos resultados). En el esquema de la figura 3.5(b) se realiza la misma ope-

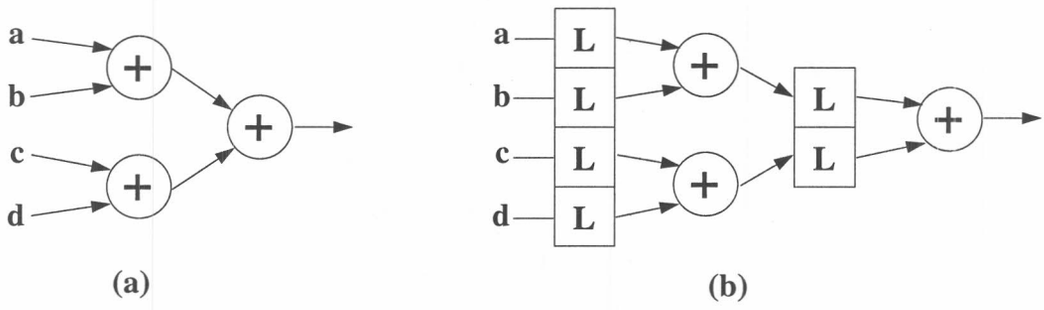


Figura 3.5: Implementación de sumas (a) combinacional. (b) en pipeline

ración pero se han introducido registros en la entrada de datos y en medio de los operadores. La inserción de registros implica también el agregado de una señal de reloj. Supongamos que los registros actualizan su salida en función de la entrada en cada flanco ascendente del reloj y que nuestro reloj tiene un periodo igual a t_+. Entonces, en el primer flanco ascendente se cargan los registros con los datos de entrada, los sumadores tardarán en realizar las suma un tiempo t_+, el mismo tiempo que dura un ciclo de reloj, así al llegar el segundo flanco de reloj, los registros actualizarán su salida con los valores de entrada. Los registros de entrada se actualizarán con datos nuevos si los hubiera y los registros intermedios con el resultado de las sumas parciales. La suma total estará disponible luego de transcurrir otro periodo t_+. En consecuencia, se demora el mismo tiempo en realizar la suma completa, para ambos esquemas (a) y (b).

Ahora veamos que ocurre si necesitamos realizar esta misma operación sobre una secuencia de números en cada una de las entradas. En el esquema (a), necesitaremos $2t_+$ para obtener el resultado $(a_0+b_0)+(c_0+d_0)$, otros $2t_+$ para obtener el resultado $(a_1+b_1)+(c_1+d_1)$ y así sucesivamente. Llamemos T_a a la cantidad de entradas que el esquema (a) puede procesar por segundo. Entonces T_a queda determinado por:

$$T_a = \frac{1}{2t_+} \tag{3.1}$$

En el esquema (b), los valores (a_0, b_0, c_0, d_0) se cargarán en los registros durante el primer flanco ascendente del reloj, transcurrido un tiempo t_+, llegará el segundo flanco del reloj en el cual los resultados parciales se cargarán en los registros intermedios y una segunda colección de valores podrá ser cargada en los registros de entrada (a_1, b_1, c_1, d_1). Transcurrido el segundo ciclo de reloj, tendremos el primer resultado a la salida $((a_0+b_0)+(c_0+d_0))$, las sumas parciales (a_1+b_1) y (c_1+d_1) en los registros intermedios y otra colección

de datos en los registros de entrada (a_2, b_2, c_2, d_2). En el tercer ciclo de reloj tendremos a la salida el resultado $((a_1 + b_1) + (c_1 + d_1))$ y así sucesivamente obteniendo un resultado nuevo en cada ciclo de reloj.

Un pipeline se caracteriza principalmente por 2 propiedades, su tamaño L_p (en cantidad de etapas) y su frecuencia de funcionamiento F_p. Las etapas de un pipeline siempre están separadas por elementos de memoria, en nuestro caso, registros. De estas dos propiedades se deriva una tercera que es el tiempo de llenado D_p que es el tiempo que demora el pipeline en entregar el primer resultado desde que ingreso el primer dato. Podemos entonces caracterizar nuestro ejemplo como un pipeline de 2 etapas y una frecuencia de $\frac{1}{t_+}$. La demora entonces queda determinada por $D_t = \frac{L_p}{F_p}$. Como se mostró, una vez que el pipeline esta lleno, es capaz de entregar un resultado por ciclo de reloj, así, el esquema (b) puede entregar T_b resultados por segundo. Donde:

$$T_b = \frac{1}{t_+} \tag{3.2}$$

Operando entre las ecuaciones 3.1 y 3.2 obtenemos que:

$$T_b = 2T_a$$

el esquema (b) entrega el doble de datos por segundo que el esquema (a). Si bien, la construcción de pipelines incurre en el agregado de recursos (registros) para efectuar la misma operación estos funcionan mucho mas rápido que sus equivalente combinacionales.

Nosotros utilizaremos la arquitectura de pipeline en este trabajo para la implementación del filtro.

Capítulo 4

Módulo de Administración de Memoria

Se decidió implementar un módulo de administración de memoria para aislar a los módulos de filtro y entrada y salida de la tarea de comunicarse con la memoria externa y para asegurar la exclusión mutua de las operaciones de escritura y lecturas en el acceso a la memoria. Con este fin, el administrador de memoria tendrá puertos independientes de lectura y escritura.

El kit S3 cuenta con 1Mbyte de memoria SRAM en la placa [11]. Esta consiste en dos chips IS61LV25616AL interconectados con el FPGA como lo muestra la figura 4.1. Cada chip puede accederse independientemente pero comparten el bus de direcciones.

4.1. Especificaciones del chip SRAM

El kit S3 tiene dos dispositivos 256Kx16 SRAM IS61LV25616AL fabricados por Integrated Silicon Solution, Inc. (ISSI)[5]. Este dispositivo tiene un bus de direcciones de 18 bits y un bus de datos bidireccional de 16 bits además de 5 señales de control. El bus de datos esta dividido en el byte más alto y el byte más bajo los cuales pueden ser accedidos individualmente. Las señales de control son las siguientes:

Señal		Activo
ce	Habilita o deshabilita el chip	0
we	Habilita o deshabilita la operación de escritura	0
oe	Habilita o deshabilita la salida	0
lb	Habilita o deshabilita la parte baja del bus de datos	0
ub	Habilita o deshabilita la parte alta del bus de datos	0

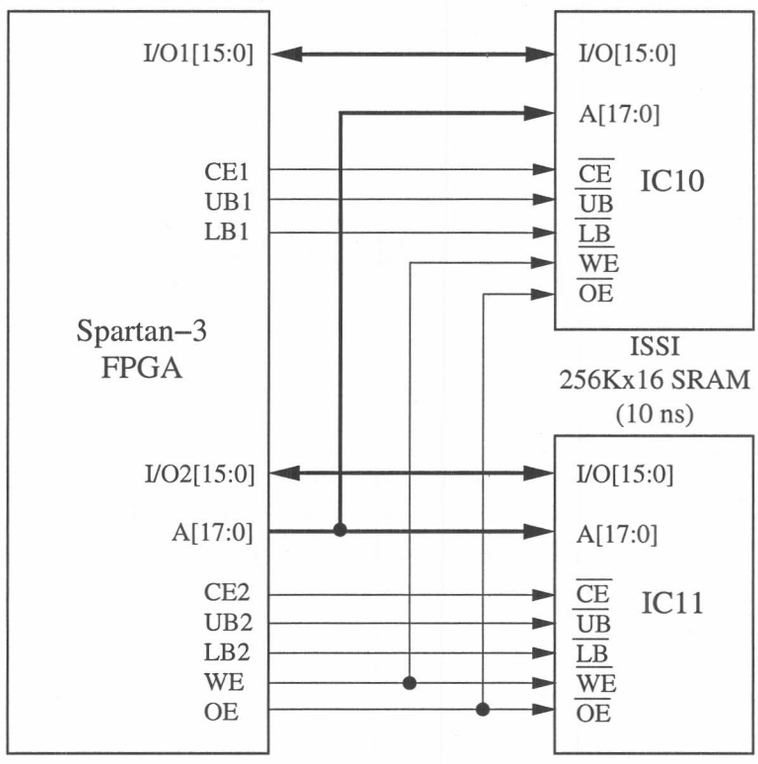


Figura 4.1: Conexión del FPGA y la memoria SRAM

Estas definen el siguiente conjuntos de operaciones que se pueden realizar sobre cada chip de memoria:

Operación	ce	we	oe	lb	up	Datos (baja)	Datos (alta)
Deshabilitado	1	-	-	-	-	Z	Z
	0	1	1	-	-	Z	Z
	0	-	-	1	1	Z	Z
Lectura	0	1	0	0	1	data	Z
	0	1	0	1	0	Z	data
	0	1	0	0	0	data	data
Escritura	0	0	-	0	1	data	Z
	0	0	-	1	0	Z	data
	0	0	-	0	0	data	data

Tanto el módulo de entrada y salida como el módulo del filtro realizarán sus accesos a memoria externa con palabras de 32 bits, así, ambos chips serán direccionados al mismo tiempo para completar la longitud de palabra tanto para lectura como para escritura. Podemos entonces dar un conjunto

de operaciones reducidas que serán las utilizadas en la construcción de este módulo

Operación	ce0,1	we0,1	oe0,1	lb0,1	up0,1	Bus datos
Deshabilitado	11	–	–	–	–	ZZ
Lectura	00	11	00	00	00	Data out
Escritura	00	00	–	00	00	Data in

Así queda entonces definidas el conjunto de señales y el valor de la mismas para las operaciones de lectura y escritura en memoria, pero aun falta definir la asignación del bus de direcciones y el esquema temporal de funcionamiento del chip.

4.1.1. Especificación temporal

La especificación temporal de un chip de memoria involucra muchos parámetros (señales, tiempos y niveles de tensión). Los diagramas de las figuras 4.2 y 4.3 muestran un esquema reducido a los parámetros mínimos para nuestro diseño y no incluyen especificaciones de tensión de las señales. Estos diagramas están construidos en base a la información brindada por los fabricantes del chip [5].

Los valores temporales mínimos están sujetos a los valores de tensión que se apliquen en las señales. Estos serán considerados en el momento de la implementación y síntesis de los módulos.

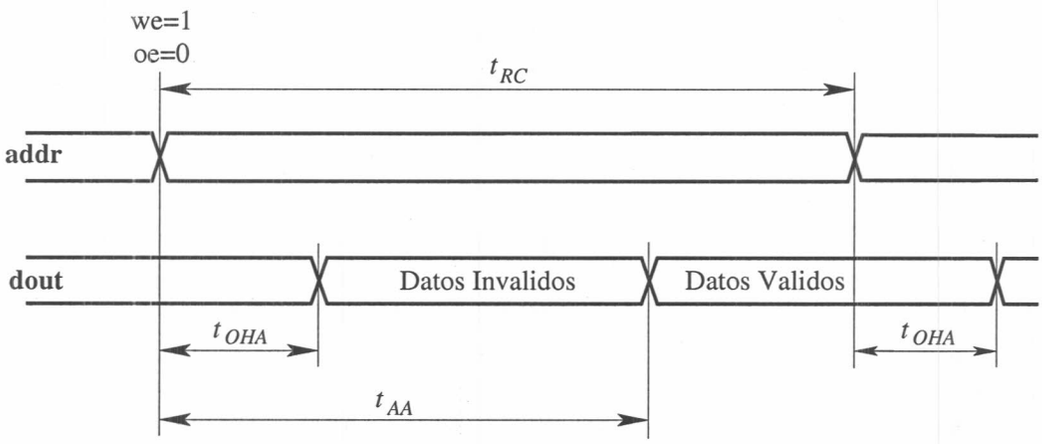


Figura 4.2: Diagrama temporal de un ciclo de lectura

Parametro		Min	Max
t_{RC}	Tiempo de ciclo de lectura	10	-
t_{AA}	Tiempo de acceso a la dirección	-	10
t_{OHA}	Tiempo de sostén de la salida	2	-

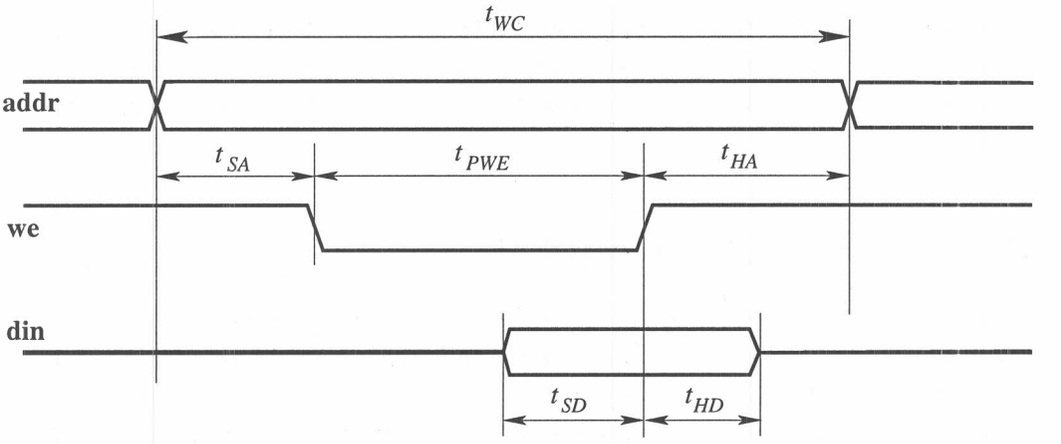


Figura 4.3: Diagrama temporal de un ciclo de escritura

Parametro		Min	Max
t_{WC}	Tiempo del ciclo de escritura	10	-
t_{SA}	Tiempo de establecimiento de dirección	0	-
t_{HA}	Tiempo de mantenimiento de dirección	0	-
t_{PWE}	Ancho del pulso *we*	8	-
t_{SD}	Tiempo de establecimiento de datos	6	-
t_{HD}	Tiempo de sostén de datos	0	-

4.2. Arquitectura

Se decidió implementar un módulo que administrará la exclusión mutua al bus de datos y direcciones de la memoria externa para no tener que distribuirlo y duplicarlo en los módulos del filtro y entrada y salida.

Si el canal de entrada y salida fuese mas rápido, se podría ir procesando la imagen a medida que esta se fuera almacenando en memoria y entonces ambos módulos competirían por el acceso a memoria. Pero como el canal es muy lento, se decidió que no se comenzará el procesamiento de la imagen hasta que esta no esté completamente cargada en memoria, y hasta que el filtro no haya terminado de procesar la imagen no se transmitirá el resultado nuevamente a la PC. Esto nos permite evaluar fácilmente el tiempo del procesamiento del filtro (sin intervención del canal de entrada y salida) y

asegurar el sincronismo que permite mantener al filtro siempre alimentado con datos. Así, el módulo de entrada y salida y el módulo del filtro no trabajan en tiempo solapado y no compiten entre sí por el acceso a la memoria. Además el módulo de entrada y salida no necesita hacer accesos de lectura y escritura consecutivos, ya que en una primera etapa, solo guardará en memoria la imagen que recibe de la PC (Escritura), y concluido el proceso de filtrado, transferirá el resultado nuevamente a la PC (Lectura). Así, su funcionalidad asegura que no habrá accesos simultáneos de lectura y escritura. El filtro, por otro lado, solo hará lecturas hasta llenar el pipeline de procesamiento, que una vez lleno, entregara un resultado que deberá almacenarse en memoria. Así, a lo largo de toda la imagen se realizarán operaciones de lectura y escritura simultáneas hasta terminar de leer la imagen. Finalmente solo habrá procesos de escritura hasta vaciar el pipeline. Como el tamaño del pipeline es significativamente menor que el tamaño de la imagen, la mayor parte del tiempo de filtrado se harán operaciones de lectura y escritura simultáneos y en consecuencia el módulo de administración de memoria debe asegurar en este caso la exclusión mutua al bus de datos y direcciones.

Como se ve en la figura 4.1, se dispone de dos chips de memoria con bus de control y datos independiente, pero bus de direcciones común. Esto nos permite diseñar al menos dos organizaciones de memoria diferentes. La opción (A) que contempla dos bancos con accesos independientes y la opción (B) que ve la memoria como un único espacio direccionable. Fig 4.4

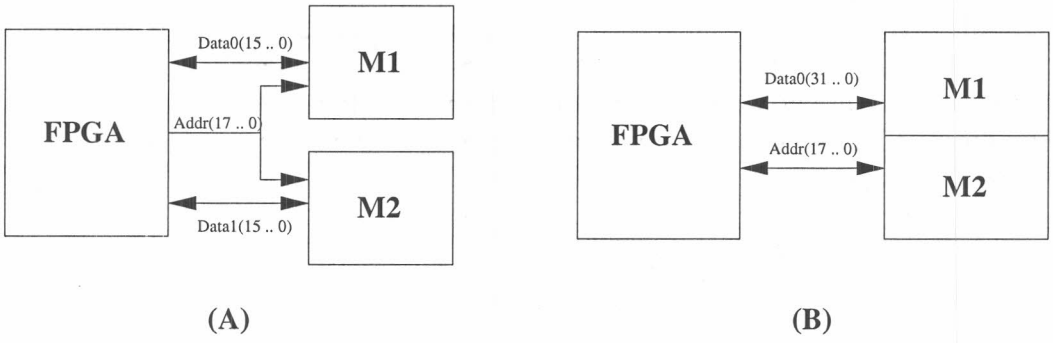


Figura 4.4: Organización de memoria. A) Bancos Independientes B) Único banco

4.2.1. Opción A

Esta organización de memoria nos permite realizar ciclos de lectura y escritura de palabras de 16 bits (2 pixels) en forma simultánea. Por ejemplo, podríamos almacenar la imagen a procesar en un banco, y la imagen resultado

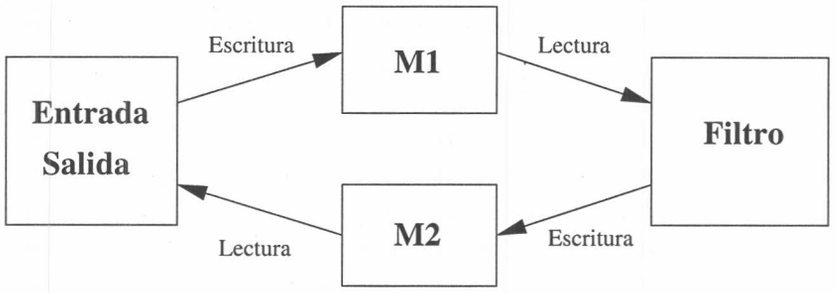


Figura 4.5: (Opción A)Flujo de datos entre módulos

en otro banco. Así, mientras el filtro lee datos del banco M1 manteniendo lleno el pipeline de procesamiento, va escribiendo la porción de la imagen que ya ha procesado en el banco M2 (ver Fig. 4.5) . Como el bus de direcciones es compartido por ambos bancos, para poder realizar ciclos de lectura y escritura simultáneos deberiamos almacenar la imagen resultado desfasada la longitud del pipeline del espacio de direcciones.

Llamamos CM_A al tiempo necesario para realizar una operación de memoria, como los accesos de lectura y escritura se realizan en forma simultánea, entonces CM_A representa un ciclo de memoria. Para evaluar el desempeño del módulo usamos la tasa de transferencia (cantidad de pixels transferidos por segundo). Así definimos a TT_{L_A} como la cantidad de pixels por segundo transferidos en operaciones de lecturas consecutivas y TT_{E_A} como escrituras consecutivas. TT_{A_A} es la tasa de transferencia para operaciones de lectura y escritura simultáneas

$$TT_{L_A} = \frac{2}{CM_A} \tag{4.1}$$

$$TT_{E_A} = \frac{2}{CM_A} \tag{4.2}$$

$$TT_{C_A} = \frac{2}{CM_A} + \frac{2}{CM_A} = \frac{4}{CM_A} \tag{4.3}$$

4.2.2. Opción B (Implementada)

En este caso disponemos de un espacio de direcciones unificado, los accesos a memoria se realizan con palabras de 32 bits (4 pixels), pero solo se puede realizar una operación a la vez. Como ya vimos, la única combinación de accesos simultáneos a memoria ocurre en el momento de filtrado y consiste en una lectura y una escritura, nunca ocurre el evento de requerir dos lecturas o dos escrituras simultáneas. Entonces, queremos obtener un comportamiento sincrónico de la memoria con accesos de lectura y escritura

simultáneos, es decir, que en un ciclo de memoria se resuelvan o un acceso de lectura o uno de escritura o uno de lectura y de escritura.

Llamamos Top_L y Top_E al tiempo requerido para realizar una operación de lectura y escritura respectivamente. Y definimos el ciclo de memoria CM_B como:

$$CM_B = Top_L + Top_E \tag{4.4}$$

De los diagramas temporales 4.2 y 4.3 se deduce que:

$$Top_L = Top_E$$

Como los tiempos de acceso están determinados por los chips de memoria independientemente del tamaño de palabra con que se acceda a los mismos, entonces:

$$Top_L = Top_E = CM_A \tag{4.5}$$

Reemplazando en 4.4 por CM_A:

$$CM_B = 2CM_A$$

Las transferencias en este caso se realizan con el doble de pixels por ciclo de memoria CM_B. Entonces:

$$\begin{aligned} TT_{L_B} &= \frac{4}{CM_B} \\ TT_{E_B} &= \frac{4}{CM_B} \\ TT_{C_B} &= \frac{8}{CM_B} \end{aligned}$$

Si comparamos ahora la tasa de transferencias de ambos esquemas A y B, reemplazado CM_B por $2CM_A$ y simplificando:

$$\begin{aligned} TT_{L_B} &= \frac{4}{2CM_A} = TT_{L_A} \\ TT_{E_B} &= \frac{4}{2CM_A} = TT_{E_A} \\ TT_{C_B} &= \frac{8}{2CM_A} = TT_{C_A} \end{aligned}$$

Se concluye entonces que ambos esquemas tienen el mismo desempeño en referencia a la cantidad de pixels que pueden entregar al filtro. Pero se decidió implementar la opción B ya que permite independencia en el direccionamiento de datos y resulta más sencillo sincronizar los accesos a memoria, puesto que en la opción A, los accesos simultáneos debe estar sincronizados (comparten el bus de direcciones) mientras que la opción B implementa accesos secuenciales en el misma ventana de tiempo (CM_B)

4.2.3. Diseño del módulo

Se define a continuación la interfaz del módulo:

Señal	Descripción	Dirección	Bus
addrW	Bus de direcciones del puerto de escritura	entrada	18
addrR	Bus de direcciones del puerto de lectura	entrada	18
dataR	Bus de datos del puerto de lectura	salida	32
dataW	Bus de datos del puerto de escritura	entrada	32
rqR	Pedido inicio ciclo de lectura	entrada	-
rqW	Pedido inicio ciclo de escritura	entrada	-
addrBus	Bus de direcciones a memoria SRAM	entrada	18
dataBus	Bus de datos a memoria SRAM	entrada/salida	32
we	Habilitación de escritura en SRAM	salida	2
lb	Habilitación byte bajo del bus de datos SRAM	salida	2
ub	Habilitación byte alto del bus de datos SRAM	salida	2
oe	Habilitación salida del datos del SRAM	salida	2
ce	Habilitación de chip SRAM	salida	2

El puerto de lectura esta formado por las señales *addrR*, *dataR* y *rqR*. Una lectura en memoria se realiza colocando la dirección del dato que se desea acceder en el bus de direcciones *addrR* y solicitando la búsqueda *rqR*. A partir de este momento se deberá esperar el tiempo CM_B a partir del cual el dato estará disponible en *dataR*.

El puerto de escritura esta conformado por las señales *addrW*, *dataW* y *rqW*. Una escritura en memoria se realiza colocando la dirección del dato que se desea guardar en el bus de direcciones *addrW*, el dato en el bus de datos *dataW* y solicitando el almacenamiento *rqW*. Se deberá esperar el tiempo correspondiente a un ciclo de memoria CM_B antes de solicitar una nueva escritura.

Tanto para la lectura como para la escritura, las señales *rqR* y *rqW* deben estar activadas durante la primer mitad del ciclo de memoria y desactivadas durante la segunda mitad.

Las señales *addrBus*, *dataBus*, *we*, *lb*, *ub*, *oe* y *ce* pertenecen a la interfaz de la memoria externa. El bus *dataBus* a diferencia del resto de las señales es bidireccional, por lo tanto hay que administrar de manera muy precisa el acceso a este bus por parte de las señales *dataR* y *dataW* para no generar un corto circuito solapando ambas señales. La figura 4.6 muestra como se implementa este control. Las señales *lb*, *ub* y *ce* están fijas en '0' (activas) manteniendo de esta manera, los bancos de memoria siempre habilitados (*ce*) y el bus de datos accedido siempre con palabras de 16 bits (*lb* y *ub*).

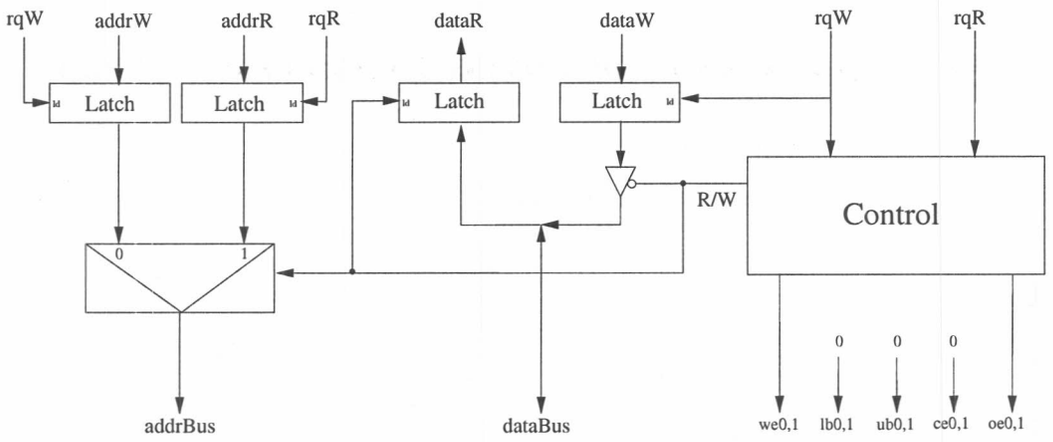


Figura 4.6: Arquitectura del módulo de memoria

Para solicitar (o almacenar) un dato en memoria, se deber colocar la dirección del dato en cuestión en el bus *addrR* (o *addrW*) y activar la señal *rqR* (o *rqW*). En ese momento la dirección (y el dato en caso de escritura) es registrada en el latch correpondiente para ser utilizada en el momento que el módulo de control inicie el ciclo de lectura (o escritura).

Cuando el módulo de control inicia un ciclo de escritura, se desactiva la señal *R/W* y en consecuencia se activa la compuerta de alta impedancia permitiendo que la salida del latch de escritura se propague a la señal *dataBus*, al mismo tiempo se desactiva la carga del latch de lectura y se selecciona la dirección del puerto de escritura en el multiplexor conectado al bus de direcciones de la memoria *(addrBus)*.

Cuando el módulo de control inicia un ciclo de lectura, activa la señal *R/W* propagando al bus de direcciones de la memoria externa *addrBus* la dirección *addrR* contenida en el latch a través del multiplexor. Al mismo tiempo, se desactiva la compuerta de alta impedancia permitiendo la carga de datos en el latch de lectura mientras el multiplexor propaga la señal *addrR* al bus *addrBus*

La figura 4.7 muestra el diagrama de transiciones de estados del módulo de control. La función de salida se muestra en la cuadro a continuación 4.1.

En el diagrama de estados y la tabla de salida, se puede observar la aparición de dos señales internas denominadas *rrqR* y *rrqW*, su utilidad se mencionará más adelante en esta sección.

Esta máquina tiene dos estados de reposo, **Read** y **Write**, a partir de los cuales se puede iniciar tanto un ciclo de lectura como uno de escritura. La razón por la cual hay dos estados de reposo es que cada uno prioriza una operación de lectura o escritura sobre la otra, promoviendo la alternancia

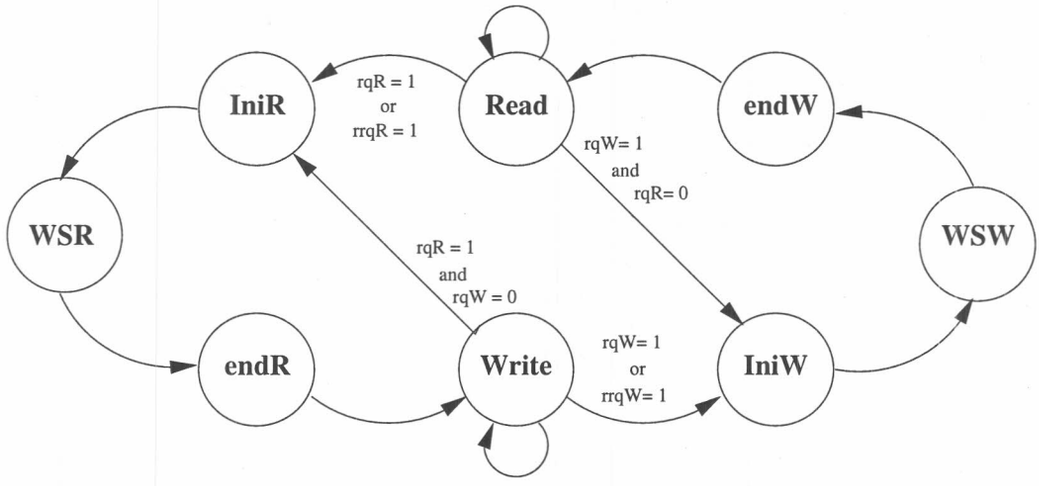


Figura 4.7: Diagrama de estados del módulo de control

Estado	R/W	oe	we	rrqR	rrqW
read	1	1	1	-	-
IniR	1	0	1	-	-
WSR	1	0	1	-	-
endR	1	0	1	0	rqW
Write	0	1	1	-	-
IniW	0	0	0	-	-
WSW	0	0	0	-	-
endW	0	0	0	rqR	0

Cuadro 4.1: Función de salida

de ciclos de lectura y escritura. Supongamos que la máquina reposa en el estado **Read** y llegan dos requerimientos simultáneos uno de lectura y uno de escritura (se activan ambas señales rqR y rqW), al estar en el estado **Read** se prioriza la operación de lectura y se comienza en consecuencia un ciclo de lectura. De haber estado en el estado **Write**, el ciclo que se hubiera iniciado seria el de escritura.

Un ciclo de lectura puede iniciarse en cualquier estado de reposo (**Read** o **Write**) y siempre termina en el estado **Write**. Las posibles transiciones son:

$$\{Read \rightarrow IniR \rightarrow WSR \rightarrow endR \rightarrow Write\}$$

$$\{Write \rightarrow IniR \rightarrow WSR \rightarrow endR \rightarrow Write\}$$

Análogamente, un ciclo de escritura siempre termina en el estado de reposo **Read** y puede estar formado por cualquiera de las siguientes transiciones:

$$\{Read \rightarrow IniW \rightarrow WSW \rightarrow endW \rightarrow Read\}$$

$$\{Write \rightarrow IniW \rightarrow WSW \rightarrow endW \rightarrow Read\}$$

Como se mencionó anteriormente, las señales de requerimiento de operaciones en memoria (rqR y rqW) deben estar activas durante la mitad del ciclo de memoria y desactivas durante la segunda mitad. Esto fomenta que no se ejecute una operación repetida 2 veces en memoria ya que cuando el módulo de control termina de realizar la operación indicada y vuelve al estado de reposo, la señal que le dió origen se encuentra inactiva. Pero también genera un problema (ver Fig: 4.8) cuando dos operaciones se solicitan en el mismo instante.

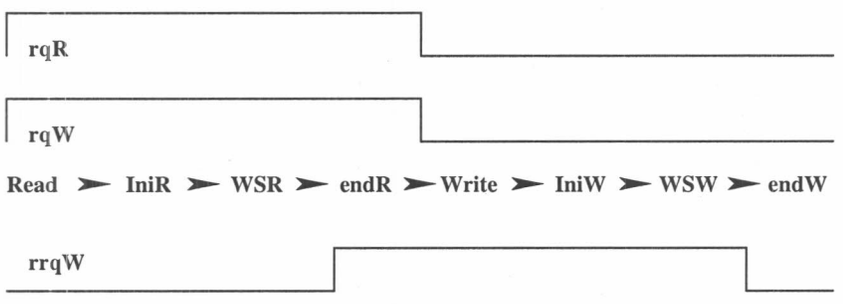


Figura 4.8: Diagrama de representación de accesos simultáneos

Supongamos que la máquina se encuentra en el estado de reposo **Read**, y se activan simultáneamente las señales rqR y rqW. Al estar en el estado **Read**, la máquina da prioridad al requerimiento de lectura e inicia la operación transitando por los estados $\{Read \rightarrow IniR \rightarrow WSR \rightarrow endR \rightarrow Write\}$. Pero al llegar al estado **Write** para iniciar la operación de escritura, la señal rqW se encuentra inactiva y en consecuencia no se inicia la correspondiente operación y esta se pierde. Si la misma situación se hubiera dado estando la máquina en el estado de reposo Write, se habría realizado la operación de escritura y perdido la de lectura.

Para solucionar este inconveniente, se incorporaron las señales $rrqR$ y $rrqW$ que registran el valor de las señales rqR y rqW en los estados **endW** y **endR** respectivamente. Así, al llegar al estado **endR** en la figura 4.8 se censa el valor de la señal rqW y se registra en la señal $rrqW$ y al llegar al estado **Write**, se inicia una operación de escritura si cualquiera de las señales **rqW** o **rrqW** están activas.

El comportamiento de las señales rrqR y rrqW se muestra en el cuadro 4.1, donde los guiones indican que la señal no se actualiza y mantiene su valor anterior. Para no repetir una operación dos veces, al transitar por los estados **endR** o **endW** se inactivan respectivamente las señales *rrqR* y *rrqW* ya que la operación correspondiente ya fue realizada.

Capítulo 5

Filtro

En el capítulo 2 se analizó el algoritmo ADM, veremos ahora como construir este algoritmo que consta de 3 etapas:

1. *Suavizado* de la imagen

   Entrada: Imagen original
   Salida: Imagen suavizada

2. *Cálculo* de la intensidad y dirección de bordes

   Entrada: Imagen suavizada
   Salida: Matriz $C_{i,j}$ = (Intensidad $pixel_{(i,j)}$, Dirección $pixel_{(i,j)}$)

3. *Mapeo* - Detección y localización de bordes

   Entrada: Matriz C
   Salida: Imagen de bordes

Como cada una de las etapas del algoritmo necesita de la salida de la etapa anterior, podemos suponer que el tiempo necesario para procesar una imagen completa es la suma del tiempo de cada una de las etapas. Pero también sabemos del análisis del algoritmo que cada una de las etapas necesita una porción muy pequeña de la imagen para procesar un dato de salida. Por ejemplo, la etapa de suavizado utiliza una máscara de 5×5 pixels para producir un pixel suavizado; la etapa de cálculo también utiliza una máscara de 5×5 pixels para producir una tupla (Intensidad, dirección) a la salida. De esta manera, podemos ir resolviendo una etapa a medida que la anterior haya procesado todos los datos necesarios para poder procesar un dato de salida, así todas ellas pueden trabajar en forma solapada en el tiempo como lo muestra la figura 5.1

Los módulos cargador y almacenador se encargan de traer datos de la memoria externa y de almacenarlos en la misma respectivamente.

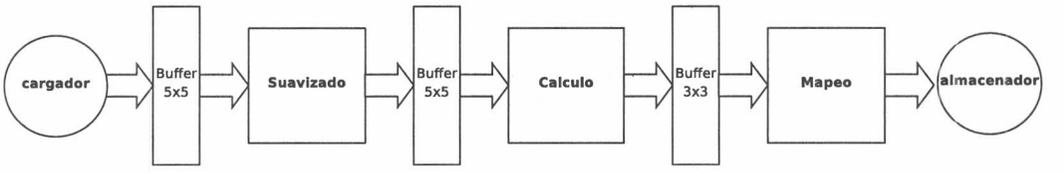


Figura 5.1: Estructura en pipeline del algoritmo ADM

5.1. Procesamiento

5.1.1. Suavizado

Como se menciono en el capítulo 2, el primer paso del algoritmo consiste en el suavizado de la imagen. Cada pixel suavizado es la suma ponderada de sus vecinos aplicando una máscara semi-gaussiana que prioriza al pixel central de la imagen como se muestra en el cuadro 5.1(a). Analizando esta máscara, podemos ver que su utilización implica realizar 24 sumas, 25 productos y 1 división, para obtener un pixel suavizado. En [3] se propone utilizar la máscara del cuadro 5.1(b) para reducir las operaciones aritméticas a solo sumas y desplazamientos de bits. Los pesos de los pixels en el filtro semi-

0.353	0.447	0.5	0.447	0.353
0.447	0.707	1	0.707	0.477
0.5	1	2	1	0.5
0.447	0.707	1	0.707	0.477
0.353	0.447	0.5	0.447	0.353

(a)

0.25	0.5	0.5	0.5	0.25
0.5	0.75	1	0.75	0.5
0.5	1	2	1	0.5
0.5	0.75	1	0.75	0.5
0.25	0.5	0.5	0.5	0.25

(b)

Cuadro 5.1: (a) Máscara semi-gaussiana, (b) Máscara aproximada

gaussiano se han elegido de forma tal que todos sean representables como potencias de 2 o sumas de potencias de 2 y que su suma total también sea potencia de 2 como se muestra a continuación

Coeficiente	Operación	
.0.25	2^{-2}	Desplazar 2 posiciones a la derecha
0.5	2^{-1}	Desplazar 1 posición a la derecha
0.75	$2^{-1} + 2^{-2}$	Suma de dos bytes desplazados
1	2^{0}	
2	2^{1}	Desplazar 2 posiciones a la izquierda
$\frac{1}{16}$	2^{-4}	Desplazar 4 posiciones a la derecha

De esta manera, todos los productos se transformaron en desplazamiento de bits y lo mismo ocurre con la división final del promedio.

La Figura 5.2 muestra el diseño de la máscara del cuadro 5.1(b). Así, las entradas a los sumadores son los pixels de la máscara desplazados según la tabla anterior. En el diseño se observan columnas de latches (L), de manera que cada resultado de una suma es guardado en un latch. Si los sumadores hubieran estado conectados unos con otros directamente, el resultado hubiera sido el mismo pero el tiempo de procesamiento seria la suma de los tiempos de propagación de cada sumador dispuesto en cascada, mientras de esta manera la suma se realiza en un pipeline, que una vez lleno, se obtiene un resultado en el tiempo de propagación de un solo sumador.

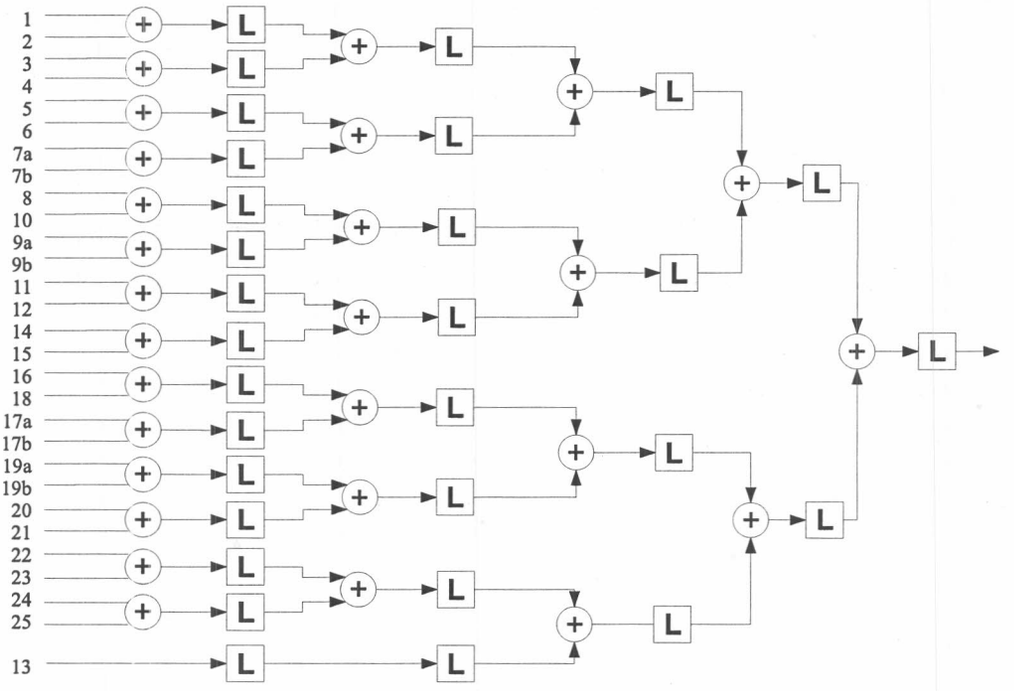


Figura 5.2: Implementación de máscara de suavizado

Se deberá tener en cuenta al momento de implementar, la cantidad de bits que serán utilizados por los sumadores, puesto que al haber desplazado los bits, no necesariamente todos los datos de entrada tiene el mismo ancho de bits.

5.1.2. Cálculo

Como se explicó en el capitulo 2 para realizar el cálculo de la intensidad y dirección del borde también se necesita una subregión de la imagen de 5x5 pixels. La figura 5.3 muestra la arquitectura del procedimiento,también en este caso se han separado las etapas de cálculo con latches para ir almacenando los resultados parciales y de esta manera reducir el tiempo total de cálculo distribuyéndolo en una estructura de pipeline. El procedimiento completo del cálculo pude ser dividido en tres etapas como se muestra a continuación:

1. Cálculo preliminar:

   Se realizan las sumas previas al cálculo de deferencias absolutas, todas las sumas se efectúan en paralelo y sus resultados son almacenados en la primer columna de latches.

$$
\begin{aligned}
V_u &= V_{u_1}+V_{u_2} \\
V_l &= V_{l_1}+V_{l_2} \\
H_r &= H_{r_1}+H_{r_2} \\
H_l &= H_{l_1}+H_{l_2} \\
Pd_u &= Pd_{u_1}+Pd_{u_2} \\
Pd_l &= Pd_{l_1}+Pd_{l_2} \\
Nd_u &= Nd_{u_1}+Nd_{u_2} \\
Nd_l &= Nd_{l_1}+Nd_{l_2}
\end{aligned}
$$

Pd_{u_2}		V_{u_2}		Nd_{u_2}
	Pd_{u_1}	V_{u_1}	Nd_{u_1}	
H_{l_2}	H_{l_1}	$p_{(i,j)}$	H_{r_1}	H_{R_2}
	Nd_{l_1}	V_{l_1}	Pd_{l_1}	
Nd_{l_2}		V_{l_2}		Pd_{l_2}

2. Cálculo de todas las diferencia absolutas:

   Con los resultados de las sumas del paso anterior se realizan las diferencias y estas son almacenadas en una segunda columna de latches.

$$
\begin{aligned}
V &= |V_u - V_l| \\
H &= |H_r - H_l|
\end{aligned}
\qquad\qquad
\begin{aligned}
Pd &= |Pd_u - Pd_l| \\
Nd &= |Nd_u - Nd_l|
\end{aligned}
$$

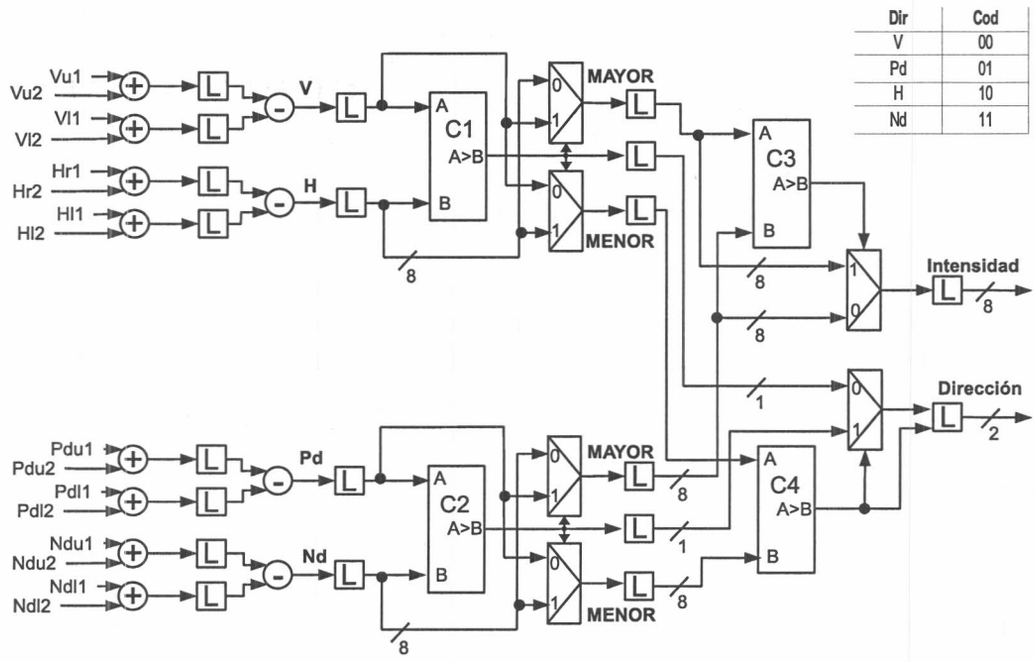


Figura 5.3: Cálculo de Intensidad y dirección del borde

3. Cálculo de intensidad y dirección:

   Para calcular la intensidad del borde es necesario calcular el máximo de todas las diferencias absolutas.

$$S_b = \frac{\text{máx}\{V, H, Pd, Nd\}}{2}$$

   Como los comparadores toman únicamente 2 valores de entrada, se necesitan 3 comparadores para obtener el máximo absoluto. Así el comparador **C1** compara los valres H y V mientras que el comparador **C2** compara Pd y Nd. A la salida de cada comparador los multiplexores ordenan el **Mayor** y el **Menor** de los valores comparados y los almacenarán en sendos latches. De esta manera tenemos ordenados en los latches los máximos y los mínimos parciales. El comparador **C3** toma los valores máximos parciales y calcula el máximo total que se corresponde a la intensidad del borde. Mientras que el comparador **C4** compara los mínimos parciales para obtener el mínimo total. El cálculo de la dirección se realiza mediante la siguiente formula:

$$dir_b = dir(\text{mín}\{V, H, Pd, Nd\})$$

Entonces la dirección del borde se calcula identificando en mínimo total y aplicando la transformación indicada en la tabla de la figura 5.3. Esta tabla esta implementada mediante el multiplexor controlado por **C4** y su resultado.

5.1.3. Mapeo

De la etapa anterior tenemos para cada pixel de la imagen un valor de intensidad de borde y su dirección, ambos datos son utilizados para determinar la ubicación del borde.

Un máximo local en los tonos de grises determina entonces la presencia de un borde. Este se detecta comparando la intensidad del pixel en análisis con la de sus vecinos en la dirección normal al borde.

$P1$	$P2$	$P3$
$P4$	$P5$	$P6$
$P7$	$P8$	$P9$

(a)

$dir(P5)$	Vecinos
Dn	$P3, P7$
V	$P4, P6$
Dp	$P1, P9$
H	$P2, P8$

(b)

Cuadro 5.2: (a) Máscara de pixels, (b) Vecinos según dirección del borde

Así, un pixel sera iluminado si su intensidad es mayor a la de sus vecinos correspondientes según la dirección del borde (ver cuadro 5.2) y además si su intensidad supera un umbral predeterminado. En la figura 5.4 los multiplexores **M1** y **M2** son los encargados de seleccionar, en función de la dirección del borde, los vecinos correspondientes. Los comparadores **C1** y **C2** determinan si el pixel en análisis es mayor que sus dos vecinos y el comparador **C3** determina si la intensidad del borde supera un umbral predeterminado *Th.* El control *bin* determina si la imagen resultado se generará binaria (blanco y negro) o en tonos de grises en función de la intensidad del borde.(**M3**)

5.2. Entrada y salida

Los módulos **almacenador** y **cargador** son los encargados respectivamente de guardar la imagen resultado en memoria y de ir copiando la imagen

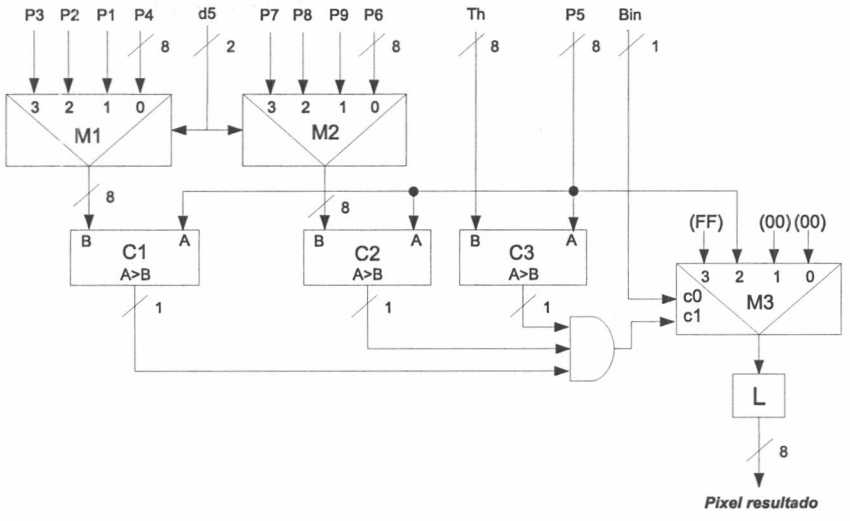


Figura 5.4: Implementación del Mapeo de Bordes

original en los buffers internos para su procesamiento. El comportamiento de estos módulos es muy simple y consiste en una máquina de estados que una vez iniciada, cicla constantemente hasta agotar los datos a transferir.

5.2.1. Cargador

El cargador deberá ir trayendo la imagen almacenada en memoria y cargarla en el buffer. Se describe su funcionamiento en la figura 5.5 y la función de salida en la tabla a continuación. Nótese que las salidas están registradas y las señales que no aparecen en la función de salida mantienen su valor anterior.

Cuando la máquina es habilitada *Start=1* inicia un ciclo de memoria *mrq=1* y transita del estado **idle** al estado **b1** en el cual solamente mantiene activa la señal *mrq* para luego transitar al estado **b2**. En este estado se desactiva la señal *mrq* y se incrementa la señal *addr* en 1 preparándose para el próximo ciclo de lectura en memoria. En el estado **b3** solo se mantiene la *mrq* inactiva aguardando a que el ciclo de memoria iniciado termine (recordar que se debe mantener la señal *mrq* activa durante la mitad del ciclo de memoria e inactiva la segunda mitad). Finalmente se transita al estado **b0** cuando el ciclo de memoria esta finalizado y se dispone del dato recuperado que es registrado en la palabra de salida *do=din* y se comunica que hay un dato disponible **drout=1**. También en este estado se inicia un nuevo ciclo de

memoria *mrq=1*. Si se hubieran recuperado todos los pixels de la imagen *addr=ImFin*, la máquina vuelve al estado **idle** habiendo transferido al buffer la imagen completa, sino, transita al estado **b1** repitiendo el proceso hasta finalizar. Nótese que cuando la máquina inicia, transita del estado **Idle** → **b1**, salteándose **b0**, esto es porque **b0** habilita la escritura en el buffer *drout=1* y durante el primer ciclo no se dispone aún de datos desde la memoria.

5.2.2. Almacenador

El Almacenador deberá tomar cada uno de los pixels entregados por el módulo de Mapeo, agruparlos en palabras de 4 bytes y almacenarlos en la memoria externa. Se describe su funcionamiento en la figura 5.5 y la función de salida en la tabla a continuación.

Cuando el módulo de Mapeo tiene el primer pixel de la imagen resultado disponible, activa la señal *drout* conectada a la entrada *drin* del Almacenador. Así comienza el funcionamiento de este módulo que toma la señal de entrada y la almacena en un registro interno *data0=din* para su posterior utilización y transita del estado **idle** al estado **b1**. En este estado registra el siguiente dato entregado por el módulo de Mapeo *data1=din*. El módulo de Mapeo irá entregando en cada ciclo de reloj un pixel resultado (byte), y estos deberán ser acumulados hasta completar los 4 bytes necesarios que serán transferidos a memoria externa. Así, obtenido el segundo pixel, se transita al estado **b2** donde se registrarán el tercer pixel y la palabra de dirección *addr=addri*. La señal *addri* es un registro interno que mantiene la próxima dirección que será utilizada. En el estado **b3** llega el cuarto byte y se completa la palabra de salida de 32 bits concatenando este y los bytes previamente registrados *do=din&data2&data1&data0*, también se incrementa el registro interno *addri=addr+1*. Finalmente se transita al estado **b0** donde se inicia el primer ciclo de memoria *mrq=1* con los datos previamente registrados *addr* y *do* Así se continúa ciclando hasta que se transfieren todos los datos a memoria addr=ResFin. Nótese que cuando la máquina inicia, transita del estado **Idle** → **b1**, salteándose **b0**, esto es porque **b0** habilita la escritura en memoria externa *mrq=1* y durante el primer ciclo no se dispone para almacenar. El registro addr1 fue incorporado ya que se necesita incrementar la dirección de almacenamiento en todos los ciclos de memoria menos en el primero.

La figura 5.5 muestra la interconexión de los módulos que componen el pipeline completo del proceso de filtrado.

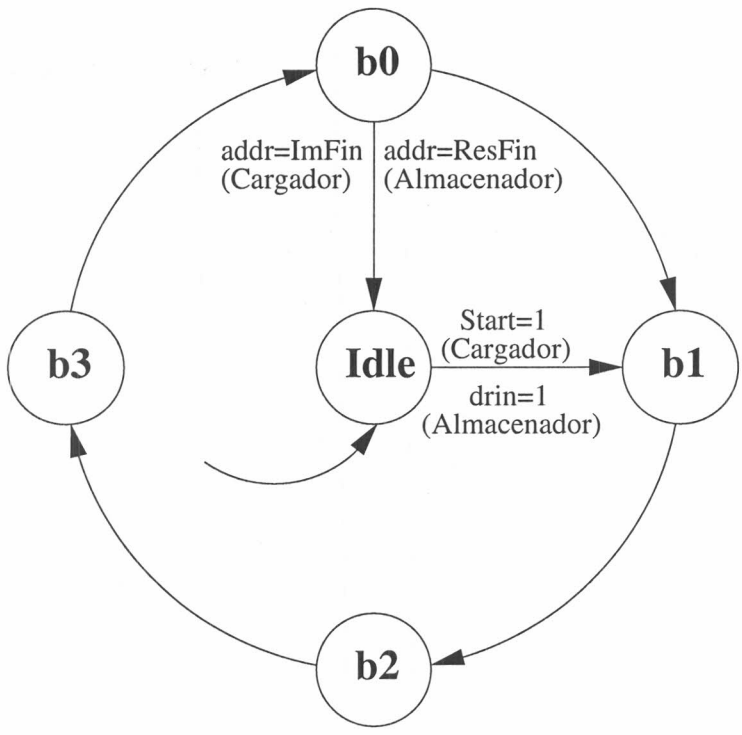


Figura 5.5: Máquina de estados de los módulos Almacenador y Cargador

Estado	Almacenador	Cargador
Idle	mrq=0 addri=0 data0=din	mrq=Start addr=0 drout=0
b0	data0=din mrq=1	mrq=1 do=din drout=1
b1	data1=din	mrq=1
b2	mrq=0 data2=din addr=addri	mrq=0 addr=addr+1 drout=0
b3	addri=addri+1 do=din&data2&data1&data0	mrq=0

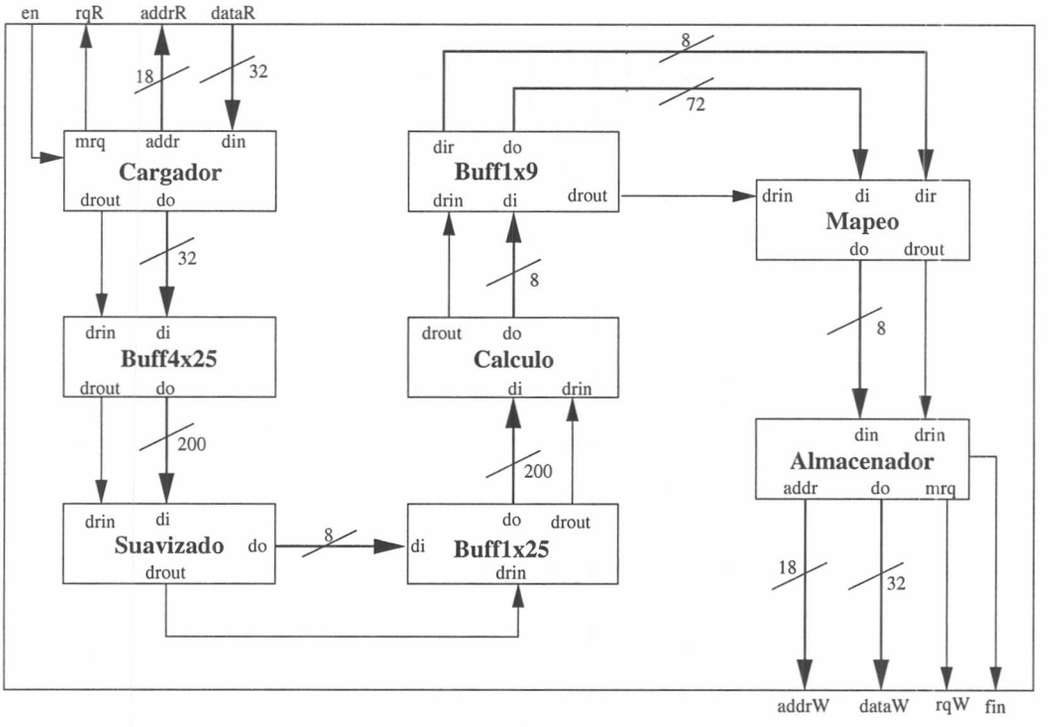


Figura 5.6: Diagrama de interconexión de módulos

5.3. Análisis temporal

En esta sección analizaremos las propiedades del pipeline que hemos diseñado hasta el momento. Así, se analizarán las tasas de transferencia necesarias para mantener el pipeline lleno, y la demora que este genera según su longitud.

5.3.1. Tasas de transferencia

Para poder mantener el pipeline lleno debemos saber con cuantos datos hay que alimentarlo y con que frecuencia. Así, sabemos por el diseño que para procesar un pixel, la primer etapa necesita 25 pixels de entrada, pero lo cierto es que 24 de esos pixels son reutilizados para el procesamiento de pixels futuros, y solo uno es descartado. Esto se explicará con mas detalle en el capitulo 6. Así independientemente de los datos necesarios para procesar, sabemos que en cada ciclo de reloj se descarta un pixel y en consecuencia se utiliza un pixel nuevo que reemplaza al anterior. De esta manera, el pipeline necesita de un pixel nuevo por ciclo de reloj para no vaciarse. Del capitulo 4 sabemos que la memoria es capaz de entregar 4 pixel por ciclo de memoria (CM). Entonces podemos determinar la frecuencia exacta a la que debería

funcionar el pipeline mediante la siguiente ecuación:

$$f_p = \frac{4}{CM} \tag{5.1}$$

donde f_p es la frecuencia del pipeline medida en Hz y CM el tiempo necesario para completar un ciclo de memoria medido en segundos. Como el pipeline entrega como resultado también 1 byte por ciclo de reloj, y el módulo de memoria fue diseñado para poder realizar en un ciclo de memoria una lectura y una escritura, entonces durante un ciclo de memoria se puede alimentar al filtro y guardar su resultado en memoria. Así la ecuación 5.1 determina la frecuencia en la que deberá operar el pipeline en función del ciclo de memoria para que este no se vacíe ni se pierdan datos de su salida.

5.3.2. Etapas del filtro

En cada ciclo de reloj, cada etapa del filtro toma datos de entrada y entrega datos a la salida, así, desde que el primer pixel entra al pipeline hasta que el primer pixel resultado sale, existe una demora que nos interesa calcular. Como cada etapa procesa en el tiempo correspondiente a un ciclo de reloj, entonces, la demora total del pipeline se puede calcular como la cantidad de etapas dividido la frecuencia de operación. Si llamamos n a la cantidad de etapas del filtro y d_F a la demora del mismo, nos queda:

$$d_f = \frac{n}{f_p}$$

Las etapas dentro del pipeline están claramente identificadas por la presencia de elementos de memoria, así dentro de cada uno de los módulos de procesamiento, una etapa esta separada de la siguiente mediante latches y cada módulo de procesamiento esta separado del siguiente por buffers. Si vemos los diseños de cada uno de los módulos e identificamos los latches, sabremos entonces cuantas etapas tiene ese módulo. Así determinamos que:

- **Suavizado** $\rightarrow$ 5 etapas (ver Figura 5.2) $\rightarrow d_{f1} = \frac{5}{f_p}$
- **Cálculo** $\rightarrow$ 4 etapas (ver Figura 5.3) $\rightarrow d_{f2} = \frac{4}{f_p}$
- **Mapeo** $\rightarrow$ 1 etapa (ver Figura 5.4) $\rightarrow d_{f3} = \frac{1}{f_p}$

Hasta el momento conocemos las etapas del pipeline que corresponden al procesamiento, en el capítulo siguiente se verán y analizarán los buffers intermedios.

Capítulo 6

Buffers

Como se mostró en la figura 5.1 se necesita una estructura de buffer entre la memoria externa y el filtro y también entre las distintas etapas del filtro. Estos buffers se encargan de adaptar los datos de salida de una etapa a los de entrada de la siguiente.

Por ejemplo, la memoria externa es capaz de transferir una palabra de 4 bytes por ciclo de memoria, mientras que la entrada al componente de suavizado requiere una estructura compleja de 25 bytes, quien a su vez solo devuelve como resultado un único byte mientras la siguiente etapa de filtrado requiere como entrada, también una estructura de 25 elementos.

En este capítulo analizaremos como deben ser estos buffer para poder realizar la tarea requerida en el tiempo necesario para no vaciar al pipeline de proceso. Primero se hará una introducción a los Blocks Ram que serán el componente de memoria que utilizarán los buffers.

6.1. Block Ram

El FPGA Spartan 3 posee bloques de memoria interna que pueden ser utilizados para almacenamiento de grandes cantidades de datos [15]. En particular, se dispone de 12 bloques los cuales puedes ser accedidos independientemente ya sea en la modalidad de único puerto o doble puerto (fig 6.1). Cada puerto puede ser configurado individualmente, esto es, cada puerto puede poseer distinto ancho de palabra y frecuencia de reloj. Además estos bancos de memoria disponen de bits adicionales utilizados para almacenar paridad

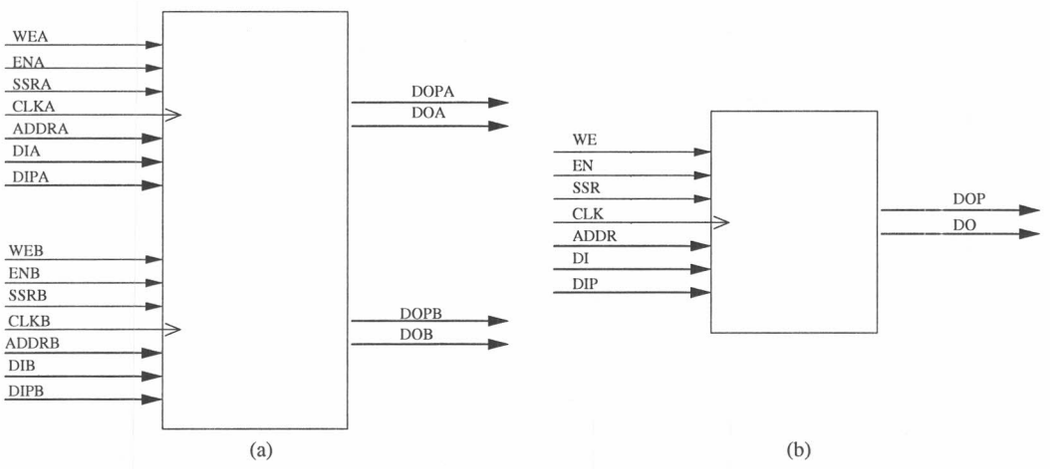


Figura 6.1: Diagrama de Block Rams (a) Doble puerto, (b) puerto simple

		Doble Puerto		
Descripción	**Puerto Único**	**Puerto A**	**Puerto B**	**Dirección**
Bus de datos	DI	DIA	DIB	Entrada
Bus de paridad	DIP	DIPA	DIPB	Entrada
Bus de datos	DO	DOA	DOB	Salida
Bus de paridad	DOP	DOPA	DOPB	Salida
Bus de direcciones	ADDR	ADDRA	ADDRB	Entrada
Habilitación de escritura	WE	WEA	WEB	Entrada
Habilitación de reloj	EN	ENA	ENB	Entrada
Set/Reset Sincrónico	SSR	SSRA	SSRB	Entrada
Reloj	CLK	CLKA	CLKB	Entrada

Como se mencionó anteriormente, cada puerto del bloque de memoria puede ser configurado independientemente, no obstante, la configuración de los mismos no es libre y debe ajustarse a alguna de las configuraciones que se detallan a continuación [15].

Configuración	**Celdas**	**Ancho de Palabra**	**Bits de Paridad**
16Kx1	16k	1	-
8Kx2	8k	2	-
4Kx4	4k	4	-
2Kx8	2k	8	-
2Kx9	2k	8	1
1Kx16	1k	16	-
1Kx18	1k	16	2
512x32	512	32	-
512x36	512	32	4

6.2. Análisis de la arquitectura

Según los requerimientos de arquitectura mencionados en el capítulo anterior, necesitaremos tres buffers, cada uno de características diferentes (fig 5.1). La funcionalidad buscada principalmente, es adaptar los datos de salida de una etapa con los datos de entrada de la siguiente etapa en el filtro. También se desea poder manejar esta estructura como una cola donde se vaya almacenando píxel a píxel las líneas de la imagen y se puedan ir recuperando en el mismo orden, estos y sus vecinos correspondientes.

La siguiente tabla muestra los distintos tipos de buffers según su origen y destino de datos y el tamaño de palabra de los mismos.

Nombre	Dato origen	Dato destino	Palabra origen	Palabra destino
buff4x25	Memoria Externa	1er Etapa del filtro	32 bits	200 bits
buff1x25	1er etapa del filtro	2da etapa del filtro	8 bits	200 bits
buff1x9	2da etapa del filtro	3er etapa del filtro	8 bits	72 bits

(a)

(i-2,j-2)
(i-1,j-2)
(i,j-2)
(i+1,j-2)
(i+2,j-2)

(b)

(i-2,j-1)	(i-2,j)	(i-2,j+1)	(i-2,j+2)
(i-1,j-1)	(i-1,j)	(i-1,j+1)	(i-1,j+2)
(i,j-1)	**(i,j)**	**(i,j+1)**	(i,j+2)
(i+1,j-1)	(i+1,j)	(i+1,j+1)	(i+1,j+2)
(i+2,j-1)	(i+2,j)	(i+2,j+1)	(i+2,j+2)

(c)

(i-2,j+3)
(i-1,j+3)
(i,j+3)
(i+1,j+3)
(i+2,j+3)

Cuadro 6.1: Vecindad de distancia 2 para el píxel (i,j) y (i,j+1)

En los buffer *buff4x25, buff1x25* y *buff1x9*, la estructura de salida requerida es una matriz cuadrada centrada en el píxel que se está procesando y sus vecinos (ver capítulo 2). En el caso del buffer *buff1x9* su salida es, además del píxel que se está procesando, su vecindad con distancia menor o igual a uno, mientras que los buffers *buff4x25, buff1x25* se necesita además del píxel que

se desea procesar, toda su vecindad de distancia menor o igual a 2. El cuadro 6.1 muestra este último caso, donde la vecindad del píxel (i,j) está dada por la unión de (a) y (b) y la vecindad del siguiente píxel (i,j+1) está dada por (b) y (c). Vemos entonces que ambas vecindades tienen pixels en común (b) y para generar los vecinos del siguiente píxel solo es necesario obtener 5 nuevos pixels y desplazar la vecindad anterior descartando 5 pixels. De esta manera, podemos mediante una estructura de ventana deslizante obtener la matriz de 25 elementos actualizando solo 5 de ellos por píxel a procesar.

La figura 6.2 muestra un esquema inicial de la arquitectura de los buffers.

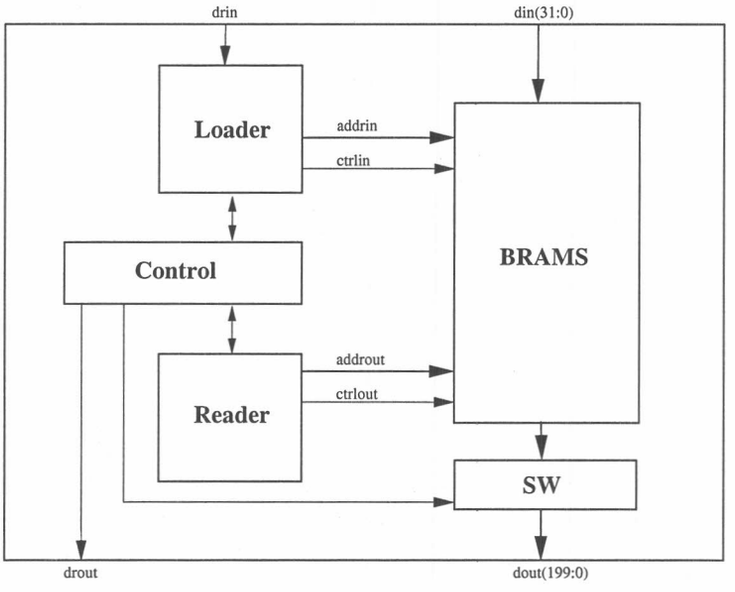


Figura 6.2: Interfase y estructura de buffer

Donde el módulo **loader** se encarga de cargar los datos en los bancos de Block Ram *BRAMS*, el módulo **reader** extrae los datos de los BRAMS, el módulo **SW** implementa la estructura de ventana deslizante que entregara a la salida la matriz de 25 elementos requerida y el módulo control es el encargado de controlar y sincronizar los módulos anteriores .

Opción A

Podríamos pensar en almacenar las 5 líneas necesarias para procesar, en 5 bancos de memorias diferentes. De esta forma, en un ciclo de reloj podemos direccionar un píxel de cada banco y entonces obtendríamos el vector columna que necesitamos para alimentar al filtro. La figura 6.3(a)

muestra la distribución de pixels por línea. Entonces si estamos procesando la línea N y columna 3, en un ciclo de reloj obtendríamos el vector $[(N-2,3),(N-1,3),(N,3),(N+1,3),(N+2,3)]$. Pero como vimos anteriormente, cada banco de memoria puede almacenar hasta 4 líneas completas. Entonces las primeras 20 líneas de la imagen quedarían almacenadas como lo muestra las figura 6.3(b).

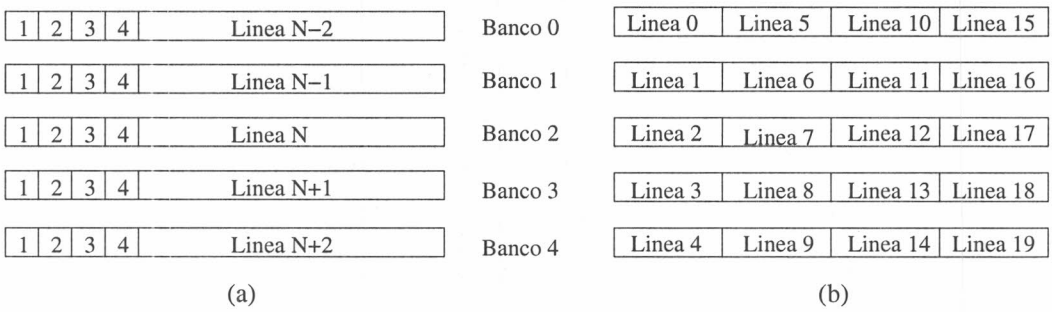


Figura 6.3: Configuración de memoria (a) Pixels por línea, (b) Lineas por banco

Ahora bien, cuando procesemos por ejemplo la línea 4, necesitaremos obtener los pixels correspondientes a las líneas 2,3,4,5,6. Supongamos que estamos analizando la columna 1, entonces la dirección de memoria para las líneas 2,3,4 será 1, pero para las líneas 5 y 6 será 513. Las siguientes formulas muestran como se calcula la dirección; donde *banco* es el banco de memoria que contiene el pixel $(fila, columna)$, y el $offset$ es la dirección dentro del banco seleccionado.

$$\begin{aligned} \text{banco} &= linea \text{ mód } 5 \\ \text{offset} &= [(linea \text{ mód } 4) * 512] + columna \end{aligned}$$

El primer problema que encontramos es que las FPGA no tienen implementado un módulo divisor y en consecuencia no se puede realizar la función módulo a menos que se implemente. Pero el mayor problema es que, como se mostró en la figura 5.1, se necesitan 2 buffers que almacenen al menos 5 líneas completas y uno de 3 líneas completas. Siguiendo la estructura antes mencionda, necesitariamos entonces $5+5+3=13$ bancos de memoria. Pero el Spartan 3 solo posee 12 bancos. En conclusión, se deberá pensar en otra forma de implementar la estructura de los buffers.

Opción B (implementada)

En la sección anterior se mostró la necesidad de reducir la cantidad de bancos de memoria para poder implementar el diseño sobre el Spartan 3. Se intentó entonces, reducir la cantidad de bancos utilizados para cada buffer de forma tal que alcance con el hardware disponible. También se decidió que la cantidad de bancos utilizados fuera potencia de 2 para evitar la implementación de un módulo divisor.

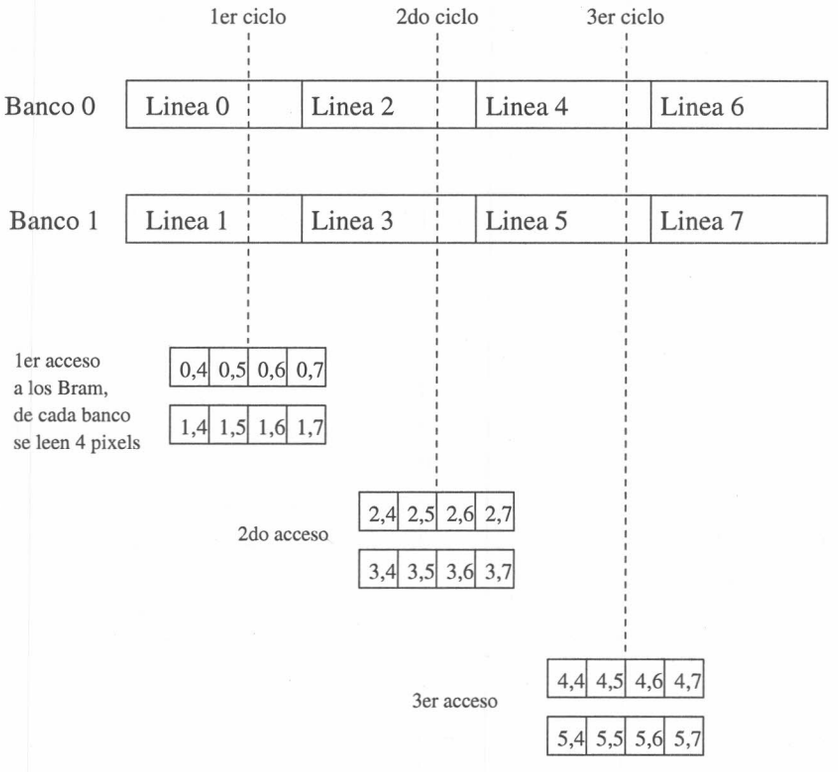


Figura 6.4: Accesos a los pixels en cada Block Ram

Dadas las condiciones de diseño, como cada banco de memoria puede contener hasta 4 líneas completas de la imagen, se utilizarán 2 bancos de memoria pudiéndose almacenar hasta 8 líneas. Se platea entonces la siguiente organización de memoria. Se dispondrá a las líneas pares de la imagen en un bram (banco 0) y las impares en otro (banco 1), de manera de poder siempre acceder a dos líneas consecutivas simultáneamente. Si bien necesitamos obtener 1 píxel de cada una de las 5 líneas, solo podemos acceder a dos de ellas al mismo tiempo. Por ejemplo si quisiéramos procesar el pixel $(2, 4)$ necesitamos el vector columna $[(0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]$. Para poder completar la columna de 5 pixels necesitaremos hacer siempre 3 accesos consecutivos a los

2 bancos simultáneamente, obteniendo de esta manera 6 pixels. Así, durante el primer acceso se leerán los pixels $(0,4)$ y $(1,4)$ del banco 0 y del banco 1 respectivamente. En el segundo acceso los pixels $(2,4)$ y $(3,4)$ y finalmente en el tercer acceso los pixels $(4,4)$ y $(5,4)$ de los cuales se deberá descartar el primero o el último dependiendo si la fila que se esta procesando es impar o par respectivamente. En este caso se esta procesando la fila 2 y en consecuencia se descartará el pixel $(5,4)$.

Para lograr el objetivo de poder procesar un píxel por cada ciclo de reloj y en consecuencia poder obtener una columna de 5 pixels por ciclo de reloj, en cada acceso a los brams recuperaremos 4 pixels consecutivos de una línea (recordar que el ancho de palabra debe ser potencia de 2), así despues de 3 ciclos obtendremos los datos necesarios para armar 4 columnas de 5 pixels cada una como se muestra en la figura 6.4. Si bien hemos encontrado una manera de poder cumplir con el requisito de entregar los datos en el tiempo necesario, también debemos entregarlos en el formato necesario. Hasta el momento disponemos de 20 bytes ordenados como 5 filas de 4 bytes cada una. Pero lo que el desplazador de ventana necesita es una columna de 5 elementos. Necesitamos encontrar entonces una estructura que reordene los datos y los entregue en tiempo y forma requeridos.

6.2.1. Transponedor

Llamaremos transponedor a la estructura capaz de recibir 5 vectores filas de 4 elementos y los transforme en 4 vectores columna de 5 elementos. La figura 6.5 muestra un esquema del funcionamiento del transponedor. Así mientras los Brams van entregando ciclo a ciclo un par de vectores fila de 4 elementos, el transponedor se va cargando con ellos. En la figura, las celdas blancas representan a las celdas que aun no han sido cargadas por los Brams. Las celdas gris claro son aquellas que están siendo accedidas (en este caso se están llenando con datos de los brams) y las celdas gris obscuro tiene datos válidos ya almacenado. El proceso de carga de los transponedores transcurre en 3 ciclos de reloj. La descarga ocurre en 4 ciclos de reloj, igual que el caso anterior, las celdas gris obscuro son las que poseen datos aun no descargados, las gris claro son las que se están accediendo en ese ciclo, y las celdas blancas son aquellas que ya han sido descargadas.

Del funcionamiento del transponedor se vé que se necesitarán al menos dos de ellos, puesto que un transponedor no entregara los datos requeridos hasta no estar totalmente cargado y esto sucede en tres ciclos de reloj. Con el uso de dos transponedores, mientras que uno se carga el otro se descara y entonces podremos entregar contantemente un vector columna por ciclo de reloj.

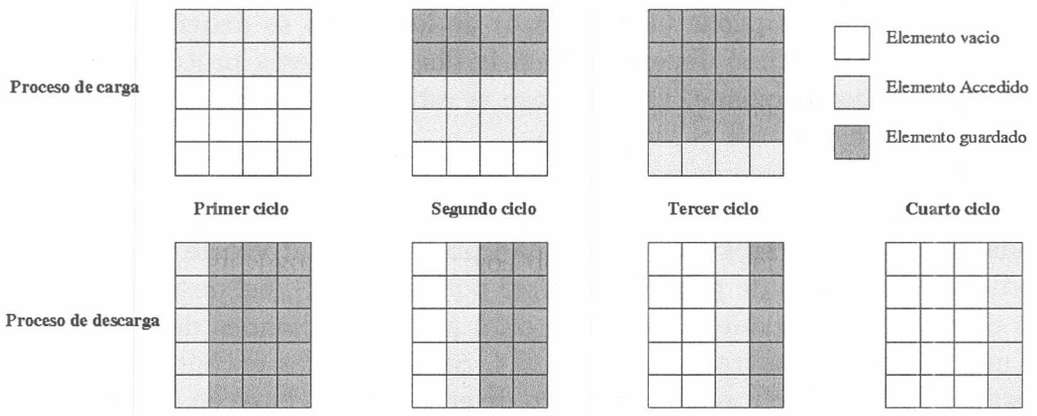


Figura 6.5: Proceso de carga y descarga del transponedor

Para almacenar los pixels que se van extrayendo de los brams necesitamos una estructura de latches que puedan habilitarse independientemente tanto para la carga como para la descarga de datos. Así, los pixels serán cargados como sucesión de pixels consecutivos de una línea y serán accedidos como pixels consecutivos en una columna.

Se decidió que los transponedores almacenaran los 6 vectores filas entregados por los Brams y que al ser accedidos se descartara el dato de mas, ya que de esta manera resulta mas sencilla la implementación al costo de utilizar mas latches. La figura 6.6 muestra la arquitectura del transponedor, las señales *ld(x)* cargan los datos en los latch y las señales *enout(x)* habilitan las salidas. La señal *hi/low* determina cual de los 6 pixels de la columna de salida será descartado. Siguiendo el ejemplo de la figura 6.4, durante el primer ciclo se activará la señal *ld0*, durante el segundo la señal *ld1* y en el tercero *ld2*. Y cuando se necesite procesar el columna 4 se activará la señal *enout0*, para la columna 5 *enout1* y así sucesivamente hasta terminar con la columna 7. El multiplexor de la salida está controlado por la señal *hi/low* y descartará el pixel correspondiente entregando a la salida el vector columna de 5 elementos.

A continuación se presenta la interfase del módulo transponedor.

Señal		E/S	Bus
clk	Reloj del sistema	entrada	
reset	Reset del módulo	entrada	
enout0	Habilitación de salida de Latch	entrada	
enout1	Habilitación de salida de Latch	entrada	
enout2	Habilitación de salida de Latch	entrada	
enout0	Habilitación de salida de Latch	entrada	
ld0	Habilitación de entrada de Latch	entrada	
ld1	Habilitación de entrada de Latch	entrada	
ld2	Habilitación de entrada de Latch	entrada	
hi/low	Control se selección de Bytes	entrada	
din	Bus de datos de entrada	entrada	64
dout	Bus de datos de salida	salida	40

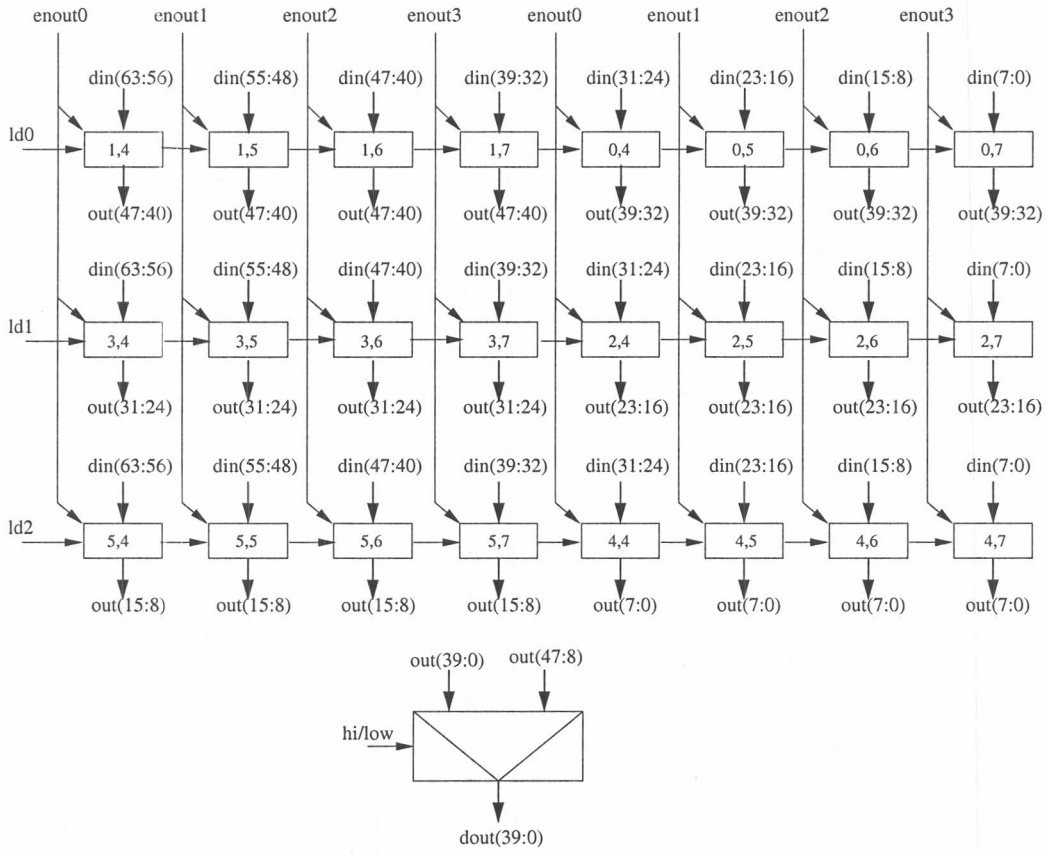


Figura 6.6: Diagrama del transponedor

6.2.2. Desplazador de ventana

El desplazador de ventanas es el ultimo módulo en el camino de datos del buffer. Su función es tomar cada columna de 5 pixels (40 bits) entregada por los transponedores y construir la matriz de 25 elementos (200 bits) para ser enviada al filtro.

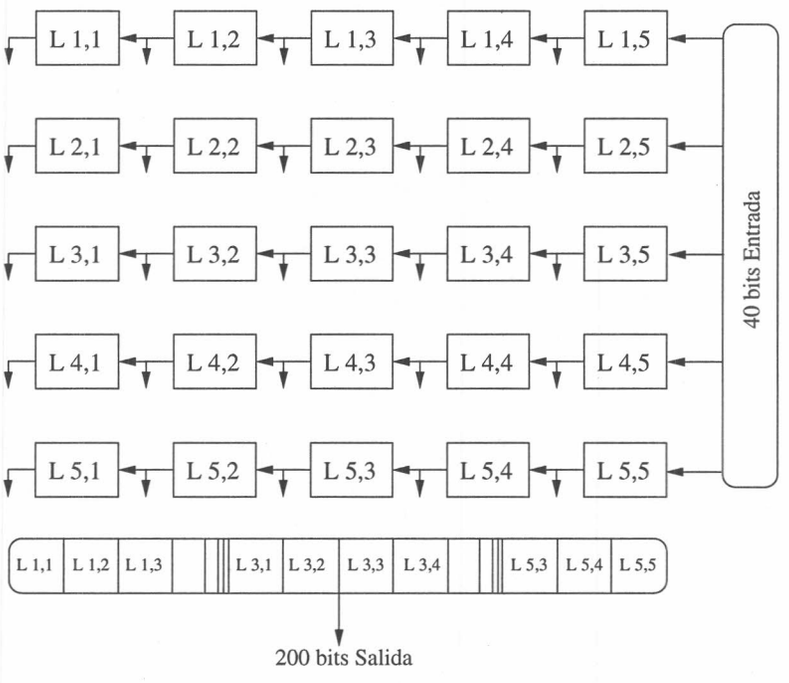


Figura 6.7: Arquitectura del desplazador de ventana

Este módulo tiene una única señal de control *endp* que lo habilita. Si la señal esta activa, se leerán los datos de entrada y se desplazará el contenido del módulo una columna para dar lugar a los nuevos datos. Así, por cada ciclo de reloj, si la entrada está disponible, todos los latch cargarán los valores de entrada y propagarán sus nuevos datos a la salida como lo muestra la figura 6.7. La salida de este módulo es una tira de 200 bits ordenada por fila, esto es, los 40 bits mas significativos serán el contenido de los latches *L1,1; L1,2; L1,3; L1,4; L1,5* los siguientes 40 bits serán *L2,1; L2,2; L2,3; L2,4; L2,5* y así sucesivamente hasta llegar a los 40 bits menos significativos con el contenido de los latches *L5,1; L5,2; L5,3; L5,4; L5,5*, esta disposición de los bits es arbitraria y el único requisito obvio es que se corresponda con el formato de entrada del módulo que alimenta.

Señal		E/S	Bus
clk	Reloj del sistema	entrada	
reset	Reset del módulo	entrada	
din	Datos de entrada	entrada	40
dout	Datos de salida	salida	200
endp	Datos disponibles en la entrada	entrada	

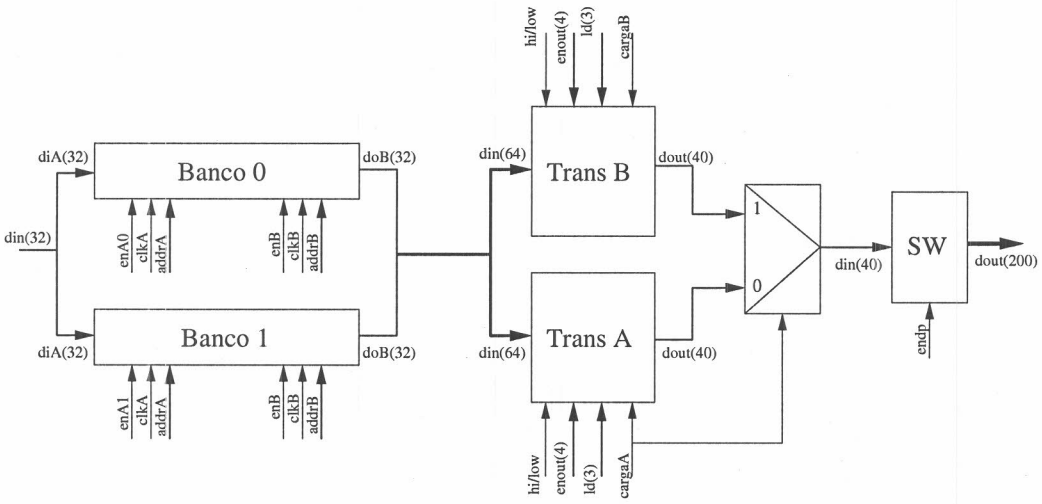


Figura 6.8: Arquitectura del camino de datos del buffer

La figura 6.8 muestra el camino de datos completo del buffer. Los datos ingresados desde memoria externa (32 bits) son almacenados en los BRAMS, para luego ser accedidos y enviados al transponedor correspondiente, mientras el otro módulo transponedor esta entregando columnas al desplazador de ventana que, a su vez, entrega a la salida una tira de 200 bits que constituye la matriz de vecindad del píxel que se está procesando.

A continuación analizaremos los módulos *loader*, *reader* y *control* que corresponden al camino de control del buffer y son los encargados del funcionamiento y sincronismo del camino de datos.

6.2.3. Loader

En el capítulo anterior, se mostró como el filtro tenía un módulo cargador que recuperaba datos de la memoria y los presentaba al buffer, entonces, la tarea de este módulo es ir guardando estos datos dentro de los brams. La señal *addrA*, es el bus de direcciones de los brams y como la memoria entrega palabras de 4 bytes, se configuró a los brams para que su bus de

datos también fuera de 4 bytes. La siguiente tabla muestra como se realiza el direccionamiento.

Bit	BRAM	Control
10	-	bit 3 del contador de fila
9	bit 8 de dirección	bit 2 del contador de fila
8	bit 7 de dirección	bit 1 del contador de fila
7	selecciona BRAM	bit 0 del contador de fila selecciona pixel descartado
6	bit 6 de dirección	-
5	bit 5 de dirección	-
4	bit 4 de dirección	-
3	bit 3 de dirección	-
2	bit 2 de dirección	-
1	bit 1 de dirección	-
0	bit 0 de dirección	selecciona Transponedor

Los datos se guardarán ordenados por filas, y cada fila consiste en 128 posiciones de 4 bytes cada una. Por lo tanto se necesitan 7 bits para direccionar cada paquete de 4 columnas y 2 bits para direccionar las filas Así, los bits $0 \cdots 6$ (offset) conforman la parte baja de la dirección (addrA) y direccionan las columnas dentro de las filas en los BRAMS, el bit 7 de la tira selecciona el bram que recibirá en dato, mientras que los bits 8 y 9 forman la parate alta de la dirección (addrA) y establecen la fila que se está direccionando (recordar que cada BRAMS puede almacenar hasta 4 filas completas). Finalmente los bits $7 \cdots 10$ son el contador real de fila de la imagen que se está almacenando y servirá para que el módulo de control calcule cuando se dispone de datos suficientes para comenzar a trabajar. La arquitectura del **loader** es muy simple y solo consiste en un registro *addrA* y un incrementador, sincronizados por la señal *drin*.

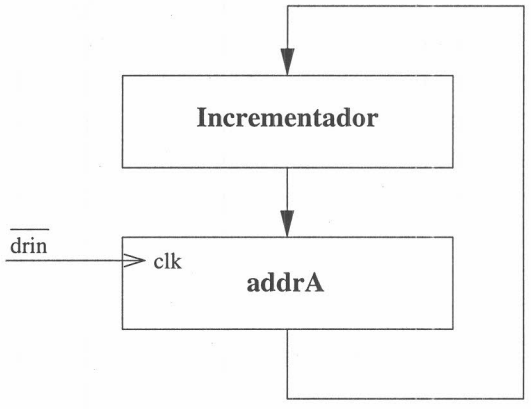


Figura 6.9: Diseño del módulo loader

6.2.4. Reader

El reader por su parte, direccionará el puerto B de los BRAMS. Este deberá realizar 3 accesos cada 4 ciclos de reloj, pero en este caso, no se deberán acceder a direcciones consecutivas. En cada acceso hay que direccionar una fila diferente que deberá alimentar a los transponedores que están controlados por el módulo de control. Se decidió entonces delegar esta tarea al módulo de control y entonces, es control quien se encarga de extraer los datos de los brams y cargarlos en los transponedores.

6.2.5. Control

El módulo de control, tendrá como tarea el control de la salida de los transponedores (*enout*) y el de la carga (*ld*), la selección de bytes de los mismos (*hi/low*) y el control del desplazador de ventana (*endp*),además de direccionar los datos de los brams (*addrB*).

El mecanismo de control deberá coordinar los módulos del camino de datos de manera sincronizada. Los bordes de la imagen necesitan un tratamiento especial dado que al estar por ejemplo, procesando la línea 0, no se dispone de 2 líneas anteriores para completar la vecindad requerida, de manera similar ocurre con las filas 1, 510 y 511 que necesitan un tratamiento especial para que la función no se indefina. Pero se decidió no realizar ningún tratamiento especial y llenar la vecindad para los bordes superior e inferior con el contenido de la memoria cualquiera sea este, y para los bordes laterales se trabajo como si la imagen fuera un cilindro, continuando el borde lateral derecho con el izquierdo. Esta decisión reduce drasticamente la lógica de control simplificándola a solo realizar siempre la misma secuencia de pasos. La siguiente tabla muestra las señales de control que serán activadas en cada paso, las casillas vacías significan que dicha señal no esta activada.

Comportamiento general del módulo de control																
Paso	**BRAMS**	**Transponedores**														**SW**
	addrB	enoutA				enoutB				ldA			ldB			endp
		1	2	3	0	1	2	3	0	1	2	0	1	2		
1	filas-1/-2	1														1
2	filas 0/1		1										1			1
3	filas +2/+3			1										1		1
4					1										1	1
5	filas-1/-2					1										1
6	filas 0/1						1			1						1
7	filas +2/+3							1			1					1
8									1			1				1

Asumiremos para la explicación que el transponedorA está cargado con datos listos para ser entregados a la salida. Durante el primer paso se deberá direccionar el puerto B de los BRAMS *(enB y fila-1/-2)* para obtener los datos correspondientes durante el próximo ciclo. También se habilita la primer salida del transponedorA *(enoutA0)*, como tenemos un dato a la salida del transponedor, debemos desplazar la ventana *(endp)*. En el segundo paso, tenemos disponibles los datos del BRAM correspondientes a la fila-1/-2 por lo tanto se habilitará la entrada del transponedorB *(ldB0)* y se direccionarán los BRAMS con los siguientes datos *(enB y fila 0/1)*. Se continúa con la siguiente salida del transponedorA *(enoutA1)* y mientras haya salidas disponibles de los transponedores, se desplazará la ventana *(endp)*. El tercer paso es similar al anterior y recién en el paso 4 ocurre que no se direccionarán los BRAMS puesto que el último dato necesario para cargar al transponedorB esta disponible durante este ciclo *(ldB2)* y el transponedorA tiene aún datos almacenados. En este paso es donde se compensa el hecho que se necesitan tres ciclos para llenar los transponedores y 4 ciclos para vaciarlos. Los pasos del 5 al 8 son exactamente igual a los pasos 1 al 4 con la diferencia que están intercambiados en funcionalidad los transponedores A y B.

Cabe destacar el hecho que por cada rotación completa de los 8 pasos se extraen de los BRAMS 8 pixels consecutivos de cada línea accedida. Así, para completar un línea se necesitan 64 iteraciones completas de la máquina, en consecuencia, al iniciar una nueva línea, el transponedorA es el que se encuentra cargado cumpliendo la suposición inicial del ejemplo para todas las líneas siguientes. Sin embargo, al iniciar su funcionamiento, la máquina dispone de los dos transponedores vacíos, por lo tanto habrá que tratar este primer caso separadamente.

Comportamiento inicial del módulo de control																
Paso	**BRAMS**	**Transponedores**														**SW**
	addrB	enoutB				enoutA				ldB			ldA			endp
		1	2	3	0	1	2	3	0	1	2	0	1	2		
1	filas-1/-2															
2	filas 0/1									1						
3	filas +2/+3										1					
4	filas-1/-2	1										1				1
5	filas 0/1		1										1			1
6	filas +2/+3			1										1		1
7					1										1	1

En los pasos 1, 2 y 3 se realiza la carga del transponedorB, como no se dispone de datos para entregar al desplazador de ventana, la señal *endp* esta desactivada. Durante los siguientes 4 pasos, se descarga el transponedorB y se carga el transonedorA, en estos pasos la señal *endp* se activa puesto que se están entregando datos a la salida. Terminado el paso 7, se continua con el procedimiento general antes mencionado teniendo el transponedorA cargado como se había supuesto. La señal *drout* que no figura en las tablas se activa en el paso 7 del procedimiento inicial, que es el momento en que el desplazador de ventana tiene el pixels $(0, 0)$ en la posición central de la matriz de salida. Y no se desactiva hasta terminar de procesar todos los pixels de la imagen.

Ahora tenemos el comportamiento completo de la estructura de control con respecto al camino de datos. Cada una de las tablas mencionadas anteriormente puede traducirse en una máquina de estados. Así, los 7 pasos del comportamiento inicial corresponden a 7 estados que iniciarán la máquina para luego ciclar entre los 8 estados (8 pasos) corresponedites al comportamiento general. La máquina comienza su funcionamiento cuando el loader haya cargado las primeras 4 líneas a los brams y termina cuando se ha estregado el ultimo pixel de la imagen.

Hasta ahora se ha explicado el funcionamiento del buffer *buff4x25* que alimenta al módulo de **suavizado**, el buffer *buff1x25* que recibe los datos del módulo de **suavizado** y alimenta al módulo de **cálculo**, es exactamente igual a buff4x25 con la diferencia que los Brams se configurarán con un puerto de escritura de 8 bits y un puerto de lectura de 32 bits. El buffer *buff1x9* es conceptualmente igual, pero escalado a una matriz de $3x3$ (el desplazador de ventana y los transponedores son mas chicos), y solo se necesita un Bram para almacenar los datos, puesto que la vecindad requerida en este caso necesita solo tres líneas, pero como este módulo debe guardar también la dirección del borde detectado además de la intensidad del mismo, se utiliza un segundo bram para este propósito.

6.3. Análisis temporal

Los buffers entregan entonces información a la tasa requerida por los módulos de procesamiento, pero necesitan cargarse antes de poder empezar e entregar datos, esto genera una demora en el llenado del pipeline. Así, los buffers *buff4x25* y *buff1x25* necesitan llenar 4 líneas, un transponedor y el desplazador de ventana antes de comenzar su funcionamiento, mientras que el buffer *buff1x9* necesitara llenar 2 líneas, un transponedor y el desplazador de ventana.. Así, la demora de cada uno de los buffers esta determinada por el tiempo que le lleva a la etapa anterior completar las líneas correspondientes. De esta manera, el buff4x25 tardará en llenar 4 líneas completas de 512 bytes cada una, un tiempo proporcional a la tasa de trasferencia de la memoria externa que lo alimenta. En el capítulo 4 se mostró que la tasa de transferencia de la memoria es de $\frac{4}{CM}$, donde CM representa el ciclo de memoria. Operando nos queda que la demora D_1 es:

$$D_1 = \frac{CM(512 \times 4)}{4} + \frac{7}{f_p}$$

El resto de los buffers recibe 1 bytes por ciclo de reloj del filtro. Del capítulo 5 sabemos que la frecuencia del filtro es $f_p = \frac{4}{CM}$ por lo tanto las demoras D_2 del buffer *buff1x25* y D_3 del *buff1x9* serán:

$$D_2 = \frac{CM(512 \times 4)}{4} + \frac{7}{f_p}$$

$$D_3 = \frac{CM(512 \times 2)}{4} + \frac{6}{f_p}$$

Teniendo las demoras de los buffers y habiendo calculado en el capítulo 5 las demoras de cada etapa de procesamiento, podemos ahora calcular la demora total D_F del filtro completo:

$$D_P = D_1 + d_{f1} + D_2 + d_{f2} + D_3 + d_{f3}$$

Reemplazando y operando nos queda:

$$D_P = \frac{CM((512 \times 4) + 7 + 5 + (512 \times 4) + 7 + 4 + (512 \times 2) + 6 + 1)}{4}$$

Finalmente nos queda:

$$D_P = 1287,5 \times CM$$

Hemos expresado la demora del pipeline en función de CM ya que es el valor que determina la frecuencia a la que debe funcionar el filtro (ecuación 5.1). Así, no solamente el tiempo de transferencia de la imagen desde y hacia la memoria externa queda determinado por el desempeño logrado por el administrador de memoria, sino que todo el sistema incluido el procesamiento dependen de este desempeño.

Capítulo 7

Implementación

Se presentó hasta aquí, un modelo de diseño que separa el camino de datos del camino de control. Esta metodología se basa fuertemente en la construcción de máquinas de estado y pipelines, ocultando todo lo posible los detalles pertenecientes al tratamiento de señales eléctricas y tiempos de propagación de señales. Se propuso así una metodología centrada en la problemática algorítmica y no en cuestiones eléctricas y temporales de las señales. Obteniéndose así, un balance entre la utilización de recursos y la performace general del sistema. En este capítulo veremos como implementar los diseños discutidos hasta ahora, en un código que sea sintetizable y nos garantice el comportamiento descripto de los módulos. Para ellos utilizaremos muchas de las herramientas bridadas por Xilinx para el desarrollo e implementación de sistemas con sus Kits de FPGA. Principalmente utilizaremos la versión libre de su entorno de desarrollo llamada ISEWebPack. Este paquete cuenta con todo lo necesario para implementar un sistema completo mediano.

7.1. Codificación

Hasta el momento hemos realizado el diseño de los principales módulos necesarios para procesar una imagen desde una PC. Así, cada módulo fue presentado con su interfase y se describió el comportamiento que debe tener. Además se mostró como estos módulos debían interconectarse. En esta sección se explicará como estos diseños pueden ser utilizados para configurar un FPGA y los pasos y métodos aplicados para este fin. El primer paso es elegir un lenguaje de descripción de hardware que pueda ser traducido (sintetizado) a componentes de hardware de un FPGA. Se eligió VHDL puesto que abarca en tu totalidad el modelo de abstracción algorítmico (ver fig. 1.8) y pareció mas idoneo para el ámbito de computación.

7.1.1. Estructura del código

A continuación se explicarán la estructura y los conceptos básicos de un programa VHDL. En general, un sistema está compuesto por varios módulos interconectados entre sí. En VHDL los módulos se denominan entidades (*Entity*) y están compuesto, en principio, por dos partes. Por un lado, la descripción de su interfaz, que lista los puertos de entrada y salida de la entidad. Y por otro lado, la descripción de su arquitectura (*Architecture*)

Entidad

La declaración de una entidad involucra básicamente, un nombre que será asignado a la entidad y una lista de puertos de entrada y salida. Notemos entonces que la declaración de entidades describe la vista externa de la entidad. Tomemos como ejemplo el módulo de administración de memoria descripto en el capítulo 4. En la sección 4.2.3 se describe la interfaz diseñada para este módulo, el siguiente código muestra como esa interfaz es codificada en VHDL para generar el módulo de administración de memoria (memmgr).

```
entity memmgr is
    Port ( clkin : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           addrW : in  STD_LOGIC_VECTOR (17 downto 0);
           addrR : in  STD_LOGIC_VECTOR (17 downto 0);
           dataW : in  STD_LOGIC_VECTOR (31 downto 0);
           dataR : out  STD_LOGIC_VECTOR (31 downto 0);
           rqR : in  STD_LOGIC;
           rqW : in  STD_LOGIC;
           addrBus : out  STD_LOGIC_VECTOR (17 downto 0);
           dataBus : inout  STD_LOGIC_VECTOR (31 downto 0);
           we : out  STD_LOGIC;
           oe : out  STD_LOGIC;
           lb0 : out  STD_LOGIC;
           lb1 : out  STD_LOGIC;
           ub0 : out  STD_LOGIC;
           ub1 : out  STD_LOGIC;
           ce0 : out  STD_LOGIC;
           ce1 : out  STD_LOGIC);
end memmgr;
```

El nombre que se quiere asignar a la entidad va entre las palabras claves **entity** y **is**, en nuestro caso *memmgr*. Y la definición de todos sus puertos, en la sección **Port** ($\cdots$). Los puertos se definen con: un nombre, por ejemplo *clkin*; una direccion, si es de entrada (in) o salida (out) y determinando el tipo de datos que transfieren (VHDL es un lenguaje fuertemente tipado, véase [6] para una referencia de los distintos tipos)

Arquitectura

En la parte de arquitectura se define el comportamiento del módulo. Esta sección consta de dos partes, en la primera se declaran todas las señales con sus tipos que serán utilizados para describir el comportamiento en la segunda parte. La descripción de la arquitectura comienza con:

```
architecture Behavioral of memmgr is
```

Inmediatamente después viene la sección de declaración de señales. La entidad *memmgr* utilizará las siguientes señales internas:

```
type estados is (mread, mwrite, iniW, wsW, endW, IniR, wsR, endR);
signal estado, prox_estado: estados;
signal reg_dataread, prox_dataread, datawrite: std_logic_vector(31 downto 0);
signal addressW, addressR: std_logic_vector(17 downto 0);
signal rw, rrqR, rrqW, prox_rrqR, prox_rrqW : std_logic;
```

En la primer sentencia (**type**) se define el tipo enumerado *estados* que será utilizado a continuación para declarar (**signal**) las señales *estado* y *prox_estado*.

La descripción del comportamiento va a continuación, entre las palabras claves **begin** y **end Behavioral;**. El comportamiento se describe mediante sentencias concurrentes y procesos. En VHDL existen dos tipos de sentencias, las concurrentes y las secuenciales, estas últimas serán utilizadas para describir procesos que se explicarán mas adelante en esta sección. Las sentencias concurrentes son aquellas que se están ejecutando todo el tiempo, todas juntas (recordar que estas instrucciones son en realidad componentes de hardware). Así por ejemplo:

```
dataR   <= prox_dataread;
databus <= datawrite when rw='0' else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
addrbus <= addressW when rw='0' else addressR;
```

la señal *addrbus* valdrá *addressW* siempre que la señal *rw* valga 0, sino, tomará el valor de la señal *addressR*. Como se ve en el ejemplo, la señal *databus* también depende del valor de la señal *rw*. Si el valor de *rw* llegara a cambiar, ambas señales *databus* y *addrbus* cambiarán en el mismo instante.

La sentencia **when** se sintetiza como un multiplexor combinacional, en este caso, de dos entradas (*addressW* y *addressR* de 17 bits) utilizando a la señal *rw* como selector. Así ambas sentencias modelan el comportamiento del multiplexor del bus de direcciones y la compuerta de tres estados (el valor 'Z' representa el estado de alta impedancia) indicada en la figura 4.6. Obviamente está prohibido y da error al intentar sintetizar una doble asignación paralela, es decir dar valor a una misma señal desde dos fuentes distintas.

Procesos

Los procesos funcionan como una sentencia concurrente y conceptualmente representan una forma de "crear" nuevas sentencias concurrentes. El comportamiento de un proceso se describe en forma secuencial mediante el uso de sentencias secuenciales. Así por ejemplo:

```
process(rqR)
begin
    if rqR='1' and rqR'event then
        addressR<=addrR;
    end if;
end process;
```

define un proceso que sera sensible a la señal *rqR* (entradas) y su comportamiento esta descripto por la ejecución secuencial de las sentencias que lo componen. En este caso es únicamente un *if*. Particularmente este *if* tiene como condición la estructura (*señal=valor* ***and*** *señal****'event***) que describe a una señal de reloj. Así, el proceso sera sincrónico y todas las señales que fueran asignadas dentro del mismo se sintetizarán como registros. Este proceso implementa el registro de la dirección de escritura indicado en la figura 4.6

7.1.2. Máquinas de estado

El diagrama en bloques de la figura 7.1 muestra la implementación de una máquina de estados genérica donde se han discriminado las salidas tipo Moore y Mealy.

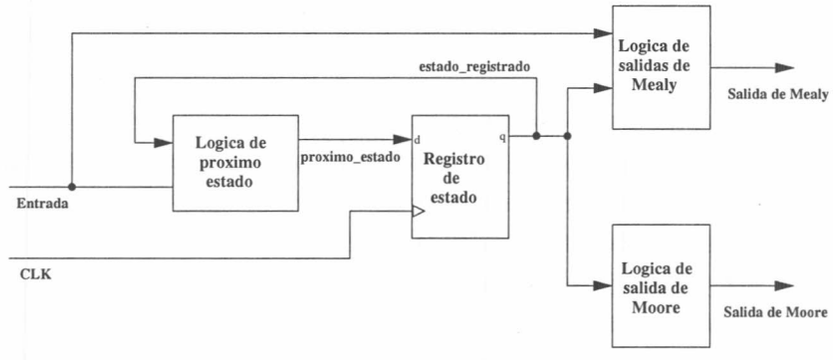


Figura 7.1: Diagrama de bloques de una máquina de estados sincrónica

En la figura se puede observar claramente tres partes: una el registro de estados, otra la lógica combinacional que implementa la función de cambio de estado y la tercera parte donde se configuran las salidas. Así, la herramienta de síntesis XST [14] reconoce y sintetiza un máquina de estados si esta está codificada en cualquiera de las siguiente tres maneras:

3 Procesos Donde en cada proceso se implementa:

1. Registro de estado
2. Lógica de cambio de estado
3. Salidas

2 Procesos Donde en cada proceso se implementa:

1. Registro de estado y Lógica de cambio de estado
2. Salidas

1 Proceso Todo se implementa en el mismo proceso.

Es importante notar que, el sintetizador solo reconoce máquinas de estado sincrónicas comandadas por un reloj. Esto genera en el caso de implementar la máquina con un único proceso que sus salidas sean registradas pudiendo no ser el efecto deseado. Siguiendo con el ejemplo, el módulo de administración de memoria fue codificado utilizando la configuración de tres procesos. Un listado de código completo puede verse en el apéndice B.1

A continuación se muestra el proceso de registro de estado:

```
process(clkin)
begin
    if reset='1' then
        estado<=mread;
        rrqR<='0';
        rrqW<='0';
    elsif clkin='1' and clkin'event then
        reg_dataread<=prox_dataread;
        estado<=prox_estado;
        rrqR<=prox_rrqR;
        rrqW<=prox_rrqW;
    end if;
end process;
```

Esta descripción de proceso genera registros con *reset* asicrónico (el *reset* puede activarse en cualquier momento independientemente de la señal de reloj (*clkin*). Además de la palabra de estado (*estado*), se han agregado al proceso tres señales más que se deseaban registrar (*reg_dataread*, *rrqR*, *rrqW*)

El segundo proceso, es el que implementa la función de cambio de estado y se muestra a continuación.

```
process (estado,rqW,rqR)
begin
    case estado is
        when mread  =>  prox_estado<=mread;
                        if rqR='1' or rrqR='1' then
                            prox_estado<=IniR;
                        elsif rqW='1' or rrqW='1'then
                            prox_estado<=IniW;
                        end if;
```

```
        when mwrite =>  prox_estado<=mwrite;
                        if rqW='1' or rrqW='1' then
                            prox_estado<=IniW;
                        elsif rqR='1' or rrqR='1' then
                            prox_estado<=IniR;
                        end if;
        when IniR   =>  prox_estado<=wsR;
        when wsR    =>  prox_estado<=endR;
        when endR   =>  prox_estado<=mwrite;
        when IniW   =>  prox_estado<=wsW;
        when wsW    =>  prox_estado<=endW;
        when endW   =>  prox_estado<=mread;
    end case;
end process;
```

y finalmente el proceso donde se implementan las salidas para cada estado. Nótese que en el código a continuación hay salidas de Moore (que solo dependen del estado actual) y salidas tipo Mealy (que dependen del estado actual y de las entradas).

```
process(estado,rqR,rqW)
begin
    prox_dataread<=reg_dataread;
    prox_rrqR<=rrqR;
    prox_rrqW<=rrqW;
    case estado is
        when mread  =>  oe<='1';
                        we<='1';
                        rw<='1';
                        if rqR='1' or rrqR='1' then
                            rw<='1';
                        elsif rqW='1' or rrqW='1' then
                            rw<='0';
                        end if;
        when mwrite =>  oe<='1';
                        we<='1';
                        rw<='0';
                        if rqW='1' or rrqW='1' then
                            rw<='0';
                        elsif rqR='1' or rrqR='1' then
                            rw<='1';
                        end if;
        when IniR   =>  oe<='0';
                        we<='1';
                        rw<='1';
        when wsR    =>  oe<='0';
                        we<='1';
                        rw<='1';
                        prox_rrqR<='0';
                        prox_rrqW<=rqW;
        when endR   =>  oe<='0';
                        we<='1';
                        rw<='1';
                        prox_dataread<=databus;
        when IniW   =>  oe<='1';
                        we<='0';
                        rw<='0';
        when wsW    =>  oe<='1';
                        we<='0';
                        rw<='0';
```

```
                    prox_rrqR<=rqR;
                    prox_rrqW<='0';
        when endW   => oe<='1';
                    we<='0';
                    rw<='0';
    end case;
end process;
```

Es importante comprender que VHDL es un lenguaje de descripción de hardware, por lo tanto, el código que uno escribe no se ejecuta, sino que se traduce a elementos de hardware. Existen muchas maneras de describir un mismo hardware y muchas maneras de implementar un mismo comportamiento con distinto hardware. Esto genera un poco de desconcierto al momento de escribir el código porque parece que no hay manera de saber a priori que hardware inferirá el sintetizador.

Pues bien, esto no es tan así, ya que el sintetizador infiere según la estructura del código, un componente de hardware especifico (véase [14] para una lista de código y hardware asociado). Así, un mismo comportamiento descripto de distintas maneras genera distinto hardware. Esto nos da la posibilidad de, si nos interesa, elegir exactamente que componentes conformarán nuestro sistema codificando de manera que el sintetizador infiera lo que nosostros queremos.

Las máquinas de estado presentan una alternativa adicional ya que, codificándola de cualquiera de las tres maneras descriptas anteriormente, el sintetizador infiere que se está modelando una máquina de estados, y si no agregamos restricciones, el sintetizador elegirá la mejor estrategia para sintetizarla.

7.1.3. Pipelines

Vimos que en general un pipeline esta formado por etapas de cálculo separadas por registros. Así la construcción de un pipeline se implementó generando un proceso por etapa y registrando su resultado todo en el mismo proceso. Tomando como ejemplo el diseño realizado para la etapa de cálculo del filtro (ver fig. 5.3).

El código a continuación representa las dos primeras etapas de las cinco que involucran dicho módulo.

```
--
-- Nivel 1
--
process()
begin
    if clkin='1' and clkin'event then
        vu <= ('0'&datain (119 downto 112))+('0'&datain (111 downto 104));
        vl <= ('0'&datain ( 95 downto  88))+('0'&datain ( 87 downto  80));
        hu <= ('0'&datain (183 downto 176))+('0'&datain (143 downto 136));
```

```
        hl <= ('0'&datain (63 downto 56))  +('0'&datain (23 downto 16));
        pdu <= ('0'&datain(199 downto 192))+('0'&datain (151 downto 144));
        pdl <= ('0'&datain (55 downto 48)) +('0'&datain ( 7 downto  0));
        ndu <= ('0'&datain (39 downto 32)) +('0'&datain (71 downto 64));
        ndl <= ('0'&datain(135 downto 128))+('0'&datain (167 downto 160));
        dato1<=datain(103 downto  96);
        rl1<=readyin;
    end if;
end process;
```

En esta primer etapa se realizan las sumas de cada par de elementos dispuestos en cada sección de las diagonales (cálculo preliminar ver sección 5.1.2). Así cada señal asignada (registro), recibe el resultado de una operación suma o en el caso de la señal *rl1*, simplemente se pasa el dato de *readyin* desde la entrada hacia la salida.

La siguiente etapa realiza todas las diferencias absolutas y también almacena los resultados en registros.

```
--
-- Nivel 2
--
process()
begin
    if clkin='1' and clkin'event then
        if vu > vl then
            v<=vu-vl;
        else
            v<=vl-vu;
        end if;

        if hu > hl then
            h<=hu-hl;
        else
            h<=hl-hu;
        end if;

        if pdu > pdl then
            pd<=pdu-pdl;
        else
            pd<=pdl-pdu;
        end if;

        if ndu > ndl then
            nd<=ndu-ndl;
        else
            nd<=ndl-ndu;
        end if;

        dato2<=dato1;
        rl2<=rl1;
    end if;
end process;
```

El código completo de este módulo puede verse en el apéndice B.2

7.2. Proceso de Síntesis

Una versión sencilla del flujo de desarrollo de sistemas basados en FPGA se muestra en 7.2. La porción derecha del flujo representa el proceso de programación y depuración, en la cual, un sistema es transformado de una descripción HDL abstracta a una configuración a nivel de celdas que finalmente es transferida al FPGA. La porción derecha es el proceso de validación que verifica que el sistema cumpla la especificación funcional (Testing). Los pasos principales en este flujo son;

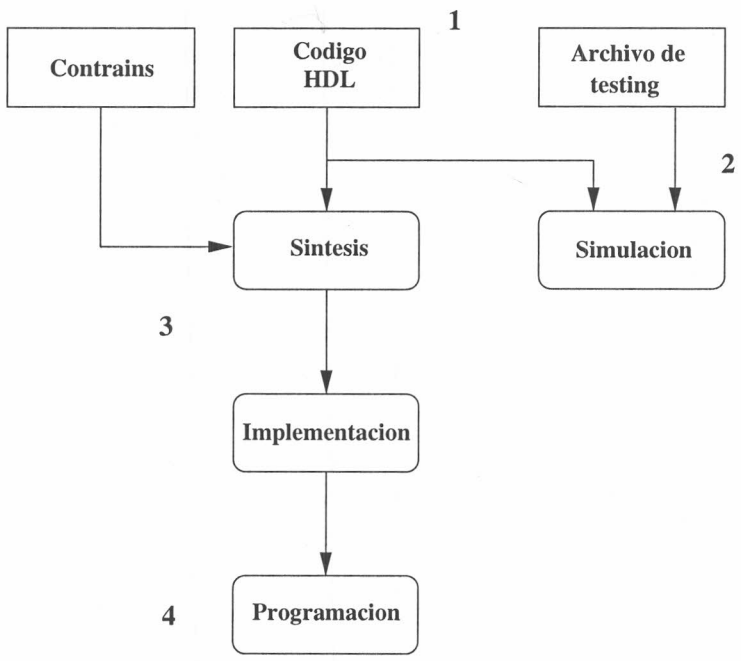


Figura 7.2: Flujo del proceso de desarrollo

1. Diseño del sistema y construcción de los archivos HDL derivados, junto con un archivo donde se indican las restricciones del diseño.

2. Desarrollo de los bancos de prueba para simular el comportamiento a nivel de transferencia de registro (RTL)

3. Síntesis e Implementación. El proceso de síntesis, también denominado síntesis lógica, es donde se transforman las estructuras descriptas en HDL a circuitos. El proceso de implementación consiste en general en tres partes:

Translate El proceso de traducción crea a partir de múltiples archivos HDL un único archivo *netlist*,que consiste en una lista de circuitos genéricos interconectados.

Map El proceso de mapeo transforma los circuitos listados en el archivo de *netlist* a celdas lógicas y puertos de entrada y salida del FPGA.

Place and Route . Durante este proceso, se establece la disposición física del diseño dentro del chip FPGA. Esto es, se ubican las celdas lógicas en un lugar físico y se determinan la rutas de interconección entre las señales.

Luego de la implementación se realiza el análisis temporal estático que determina varios parametros temporales tales como el tiempo máximo de propagación de las señales, la máxima frecuencia de reloj, etc.

4. Generación y descarga del archivo de configuración. Este es el proceso que genera y descarga el archivo que contiene la configuración con la que será programado el FPGA.

7.2.1. Herramienta de Síntesis

Xilinx provee con sus placas, un entorno de desarrollo llamado ISE (integrated software environment) que controla todos los aspectos del flujo de desarrollo. El navegador de proyectos es una interfase gráfica para que los usuarios accedan a todos los archivos relevantes asociados al proyecto y a las herramientas de software (síntesis e implementación). Las descripciones presentadas a continuación corresponden a la versión de distribución gratuita ISEWebPack 10.1.03 para linux. La figura 7.3 muestra la pantalla por defecto del ISE dividida en cuatro secciones:

Ventana de Fuentes Esta ventana es usada principalmente para mostrar los archivos asociados con el proyecto activo. La lista desplegable etiquetada *source for:* especifica la vista del diseño. Si se desea sintetizar, se deberá seleccionar *synthesis/implementation.* En cambio, si se desea simular un diseño, se deberá seleccionar *Behavioral Simulation.*

Ventana de Procesos Esta ventana muestra los procesos disponibles según el contexto. Así, dependiendo del tipo de fuentes seleccionado en la ventana de fuentes, nos permitirá elegir entre las opciones disponibles. Algunos de los procesos están encadenados y deben ser ejecutados en la secuencia correcta, por ejemplo: primero sintetizar y luego implementar. En el caso que uno decida implementar sin haber sintetizado antes,

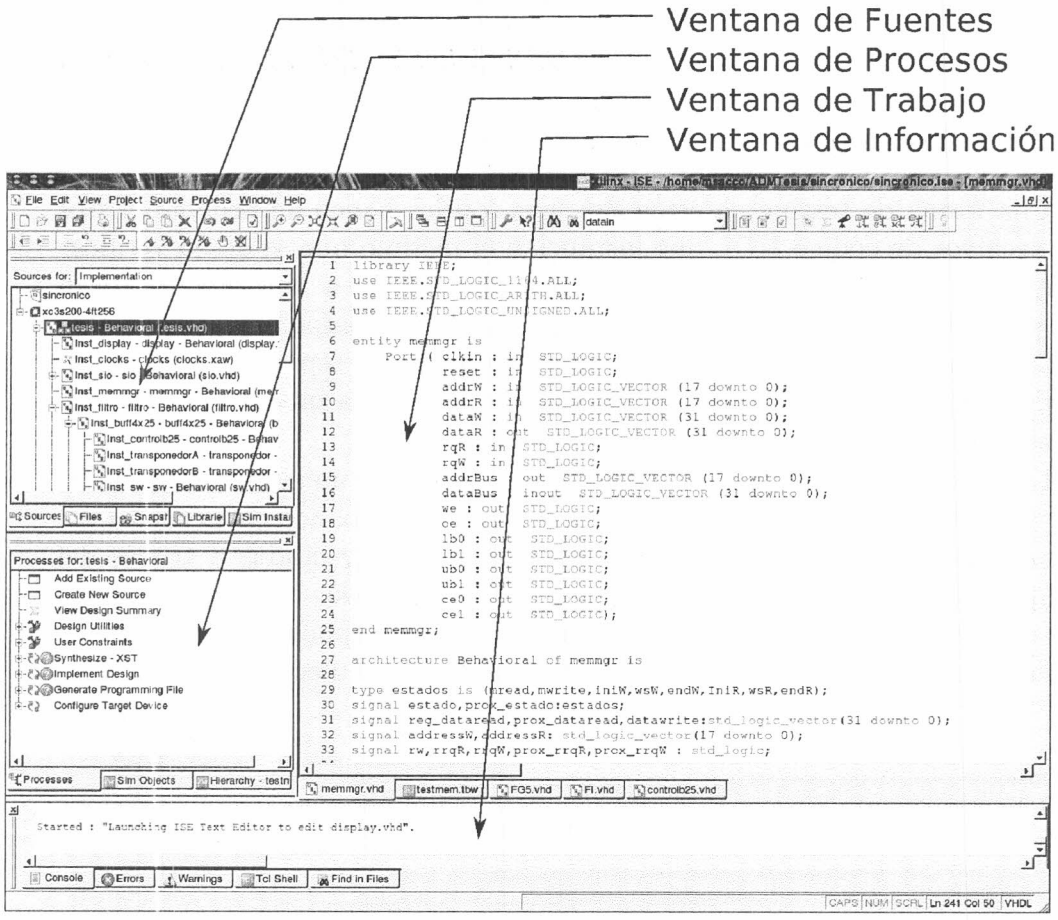


Figura 7.3: Entorno de desarrollo ISEWebPack

la herramienta realiza todos los pasos necesarios para cumplir con el proceso indicado, entonces, antes de implementar, correrá el proceso de síntesis.

Ventana de Información Esta ventana muestra el progreso del proceso que se esté ejecutando. La solapa de consola muestra los errores, advertencias y mensajes de información general. Las distintas solapas muestran la misma información discriminada por tipo.

Ventana de Trabajo Esta ventana tiene dos objetivos principales, el primero es ver y editar todos los archivos relacionados al proyecto. El segundo es la visualización de reportes generados por los procesos y el resumen de diseño.

7.2.2. Testing

Una forma de testing en estos sistemas son los banco de pruebas. Estos son archivos de código HDL que instancian el módulo que se desea verificar e inyectan señales en los puertos entrada del mismo a la vez que sensan sus puertos de salida. La salida sensada puede se comparada con valores esperados de la misma y de existir una deferencia se reporta el error o se puede visualizar la forma de onda de estas salidas en la interfase gráfica del simulador.

La herramienta de desarrollo de Xilinx cuenta con un simulador sencillo que permite de manera gráfica generar el archivo de banco de prueba (Ver figura 7.4) dibujando la forma de onda que serán inyectadas a través del tiempo a las entradas del módulo que se esta testeando. Una vez generado este archivo, se corre en el simulador y se obtienen la respuestas de los puertos de salida a lo largo del tiempo especificado para la simulación (ver fig. 7.5)

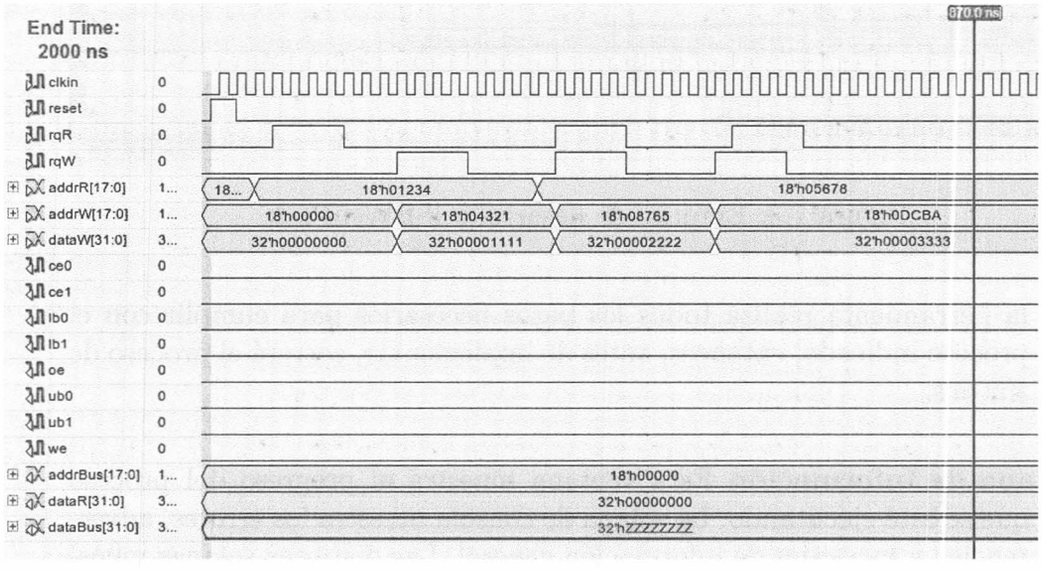


Figura 7.4: Edición del archivo de pruebas para simulación

Esta simulación se realiza a nivel de código HDL y, en general, no se toma en cuenta el tiempo de propagación de las señales y otras cuantas cuestiones temporales. Existen simuladores mucho más completos que admiten una configuración mucho más exhaustiva, pero para este trabajo, se ha utilizado el que provee la herramienta, que si bien no captura las sutilezas temporales del sistema, permite al menos detectar errores de diseño y/o codificación en relación al comportamiento y funcionalidad esperados.

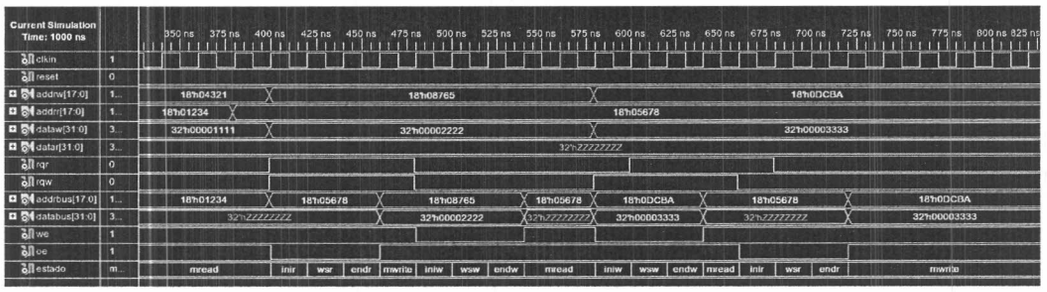


Figura 7.5: Resultado de la simulación del banco de prueba

7.3. Análisis temporal

En los capítulos de diseño, para cada módulo se realizó un análisis temporal donde todos los tiempos quedaron expresados en función del ciclo de memoria CM.

Por lo tanto, una vez determinado este tiempo, podremos calcular el tiempo teórico que insumirá el proceso completo de filtrado. Las especificaciones del fabricante del chip de memoria nos dice que un acceso a memoria requiere un tiempo mínimo de $10nseg$ o $12nseg$ [5] dependiendo de las tensiones y corrientes aplicadas a los pines de entrada. Considerando además, que las señales del FPGA tienen una demora para salir del chip a la placa y finalmente a la memoria, podemos suponer entonces que un acceso a memoria desde el punto de vista del FPGA (desde que el FPGA asigna las señales de control hasta que obtiene el resultado) será mayor a estos $10nseg$. Para determinar exactamente cual será entonces este tiempo, se realizó un test empírico que consiste en miles de transferencias a memoria con diferentes tiempos de acceso utilizando el módulo de administración de memoria (capítulo 4).

En nuestro diseño, habíamos determinado que un ciclo de memoria CM era el tiempo necesario para realizar una operación de lectura y una de escritura. Para poder realizar esto, nuestro módulo de administración de memoria debe transitar por 8 estado diferentes (ver fig. 4.7) y cada transición se hará en un tic de reloj. Podemos entonces establecer la siguiente relación:

$$F_{AM} = \frac{8}{CM}$$

donde F_{AM} es la frecuencia del reloj principal del módulo.

El resultado de test empírico determinó que todas las transferencias se podían realizar sin errores a una frecuencia de 200MHz. Por lo tanto, el ciclo de memoria resulta ser de $CM = 40nseg$ realizando un acceso a memoria en $20nseg$ (el doble del mínimo establecido por el fabricante.) La frecuencia del

filtro determinada en la ecuación 5.1 también dependía del valor de CM, y ahora queda determinada como:

$$f_p = \frac{4}{CM} = 100\text{MHz}$$

Ahora bien, transferir la imagen completa (512×512 pixels) de memoria a FPGA y de vuelta a la memoria, implica realizar $\frac{512 \times 512}{4}$ lecturas y escrituras (recordar que la transferencia a memoria se realiza con palabras de 4 bytes). Si en un ciclo de memoria se puede realizar una lectura y una escritura, entonces transferir la imagen ida y vuelta a memoria tarda

$$D_T = CM\frac{512 \times 512}{4} = 2621440\text{nseg}$$

siendo D_T el tiempo de transferencia.

En el capítulo 6 se había determinado que la demora ocasionada por el pipeline de procesamiento era:

$$D_P = 1287,5 \times CM = 51500\text{nseg}$$

Y finalmente, el tiempo teórico (T_{PT}) necesario para obtener una imagen completa procesada y almacenada en memoria principal es:

$$T_{PT} = D_T + D_P = 2672940\text{nseg}$$

Con estos valores, podríamos procesar mas de 370 imágenes por segundo que es un orden de magnitud superior a lo necesario para procesar video.

7.4. Síntesis e Implementación

En esta sección se analizará el resultado de la síntesis e implementación del diseño. Cada ítem en el procesos de síntesis descripto en la figura 7.2 genera un reporte que puede ser revisado para determinar si todos los supuestos realizados en el diseño fueron interpretados por el sintetizador y en consecuencia saber si se configuro el hardware esperado. Estos reportes además, proveen estadísticas temporales y de ocupación de área que nos permiten saber si nuestro diseño se implementará correctamente en el FPGA, o habrá que hacer modificaciones o ajustes.

Archivo de restricciones

Como se mostró anteriormente el la figura 7.2, para poder implementar el diseño se necesita no solo la codificación de todos los módulos en algún lenguaje HDL, sino también informar al sintetizador como se conectaran los puerto del diseño a los pines físicos del FPGA para poder interactuar con el exterior (memoria externa, puerto RS232, displays de 7 segmente, etc), y si fuera el caso, indicar restricciones temporales. Para esto se utiliza un archivo de restricciones (constrains). En el apéndice C.1 se lista el archivo completo utilizado para la implementación de este trabajo.

Señales de reloj

En la sección anterior, se vio que el módulo de administración de memoria funciona con una frecuencia de reloj de 200MHz, y que el módulo del filtro deberá entonces trabajar con una frecuencia de 100MHz. Ahora bien, el FPGA Spartan 3 posee una única entrada de reloj conectada, en la placa, a un cristal que oscila a una frecuencia de 50MHz. Necesitaremos entonces alguna manera de alterar la señal del reloj de entrada para obtener nuevas señales de reloj a la frecuencia deseada.

Los FPGA de la familia Spartan3 disponen internamente de cuatro dispositivos llamados DCMs (Digital Clock Managers) que permiten, entre otras cosas, dada una señal de reloj de entrada, multiplicar y/o dividir su frecuencia para obtener una nueva señal de reloj. Así configurando un DCM se consiguieron dos nuevas señales de reloj de 200Mhz y 100Mhz para el módulo administrador de memoria y el módulo de procesamiento respectivamente. El funcionamiento y configuración de los DCM se describe en [8]

De esta manera, la implementación del diseño involucra el manejo de tres señales de reloj diferentes:

50MHz Sincroniza los módulos de Entrada y salida, el temporizador y los displays de 7 segmentos.

100MHz Sincroniza todo el funcionamiento del filtro (etapas de procesamiento y buffers)

200MHz Sincroniza el Modulo administrador de memoria.

Utilizar los DCM es muy sencillo y simplemente requiere generar el componente (el paquete ISE provee una herramienta gráfica para generar y configurar los DCMs) y luego instanciarlos en algún módulo.

Reporte de síntesis

Durante el proceso de síntesis, se transformará nuestro diseño en un conjunto de circuitos interconectados. Para poder llevar a cabo esta tarea, el sintetizador interpretara el código HDL e inferirá del mismo las estructuras subyacentes que serán optimizadas y convertidas en circuitos lógicos. Terminado el proceso de síntesis, obtendremos un reporte que resumen el trabajo realizado por el sintetizador, este reporte puede ser dividido en 3 partes. En la primera, se resume para cada módulo codificado, cuales fueron las estructuras inferidas.

Por ejemplo, a continuación se muestra la sección del reporte generado por el sintetizador para el módulo de administración de memoria.

```
Synthesizing Unit <memmgr>.
    Related source file is "/home/msacco/ADMTesis/sincronico/memmgr.vhd".
    Found finite state machine <FSM_0> for signal <estado>.
    -----------------------------------------------------------------------
    | States             | 8                                              |
    | Transitions        | 16                                             |
    | Inputs             | 4                                              |
    | Outputs            | 10                                             |
    | Clock              | clkin (rising_edge)                            |
    | Reset              | reset (positive)                               |
    | Reset type         | asynchronous                                   |
    | Reset State        | mread                                          |
    | Power Up State     | mread                                          |
    | Encoding           | automatic                                      |
    | Implementation     | LUT                                            |
    -----------------------------------------------------------------------
    Found 32-bit tristate buffer for signal <dataBus>.
    Found 18-bit register for signal <addressR>.
    Found 18-bit register for signal <addressW>.
    Found 32-bit register for signal <datawrite>.
    Found 32-bit register for signal <reg_dataread>.
    Found 1-bit register for signal <rrqR>.
    Found 1-bit register for signal <rrqW>.
    Summary:
    inferred   1 Finite State Machine(s).
    inferred 102 D-type flip-flop(s).
    inferred  32 Tristate(s).
Unit <memmgr> synthesized.
```

Si comparamos el reporte con la arquitectura diseñada en la figura 4.6 vemos que el sintetizador encontró los 4 latches diseñados para los buses de datos y los de direcciones, también reconoció la compuerta de 3 estados (tristate) e infirió una máquina de estados para el módulo de control. Así podemos ir revisando para cada módulo si se sintetizó la estructura esperada.

Una vez que todos los módulos fueron sintetizados, la segunda parte del resumen informa todos los recursos del FPGA que serán utilizados para implementar el diseño. A continuación se lista el resumen generado para nuestro sistema.

```
Device utilization summary:
---------------------------

Selected Device : 3s200ft256-4

 Number of Slices:                       2237  out of   1920   116% (*)
 Number of Slice Flip Flops:             2742  out of   3840    71%
 Number of 4 input LUTs:                 3075  out of   3840    80%
    Number used as logic:                3030
    Number used as Shift registers:        45
 Number of IOs:                           86
 Number of bonded IOBs:                   86  out of    173    49%
    IOB Flip Flops:                       32
 Number of BRAMs:                          6  out of     12    50%
 Number of GCLKs:                          5  out of      8    62%
 Number of DCMs:                           1  out of      4    25%

WARNING:Xst:1336 -  (*) More than 100% of Device resources are used
```

Este resumen nos indica que nuestro diseño necesita más recursos de los que el FPGA Spartan 3 dispone. Así que tendremos que ver como solucionar este incidente.

Una posible solución sería repasar todos los módulos intentando reducir la cantidad de recursos utilizados. Pero uno de los objetivos de este trabajo es presentar una metodología de diseño y codificación que, aunque no genere un diseño óptimo, esconda los detalles temporales y a nivel de compuertas que de otra manera requeriría del programador fuertes conocimientos de electrónica. Por lo tanto buscaremos una solución alternativa que nos permita implementar nuestro diseño.

Si bien no es nuestra intención lidiar con metodos de optimización a nivel de compuertas, en nuestro caso es necesario que alguien lo haga, porque sino, no podremos implementar nuestro diseño. Así delegaremos esta tarea al proceso de implementación. En la siguiente sección veremos como configurar al proceso de implementación para que optimice de manera adecuada nuestro diseño.

El tercer punto importante dentro del reporte de síntesis es el resumen temporal. Así, el sintetizador nos informa, según la estructura inferida, cual será la velocidad máxima que podrá alcanzar nuestro diseño. A continuación se muestra el resultado obtenido.

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 20.144ns (Maximum Frequency: 49.643MHz)
   Minimum input arrival time before clock: 8.957ns
   Maximum output required time after clock: 14.919ns
   Maximum combinational path delay: No path found
```

En este caso el sintetizador nos informa que la frecuencia máxima de nues-

tro sistema es de 49,643MHz. La frecuencia a la que esta haciendo referencia es a la del reloj de entrada y no a las generadas por los DCM. El problema es que la frecuencia de entrada es fija y esta determinada por el cristal. Por los tanto, en este caso también será necesario configurar al proceso de implementación para que realice un esfuerzo extra para cumplir con los requisitos tanto de área como temporales.

Si bien el proceso de síntesis no logró cumplir con los requisitos temporales y de área, este no generó ningún error ya que el sintetizador no es el encargado de implementar el diseño. Por otro lado, los valores tanto de área como los temporales reportados son estimaciones, y hasta no tener un mapa definitivo de como será implementado el diseño no será posible saber con certeza cuales serán estos valores.

Reporte de implementación

El proceso de implementación es el encargado de convertir el archivo de circuitos interconectados que entrega como salida el proceso de síntesis, en una configuración para el FPGA. Así este proceso es el que tendrá que resolver los problemas encontrados durante la síntesis.

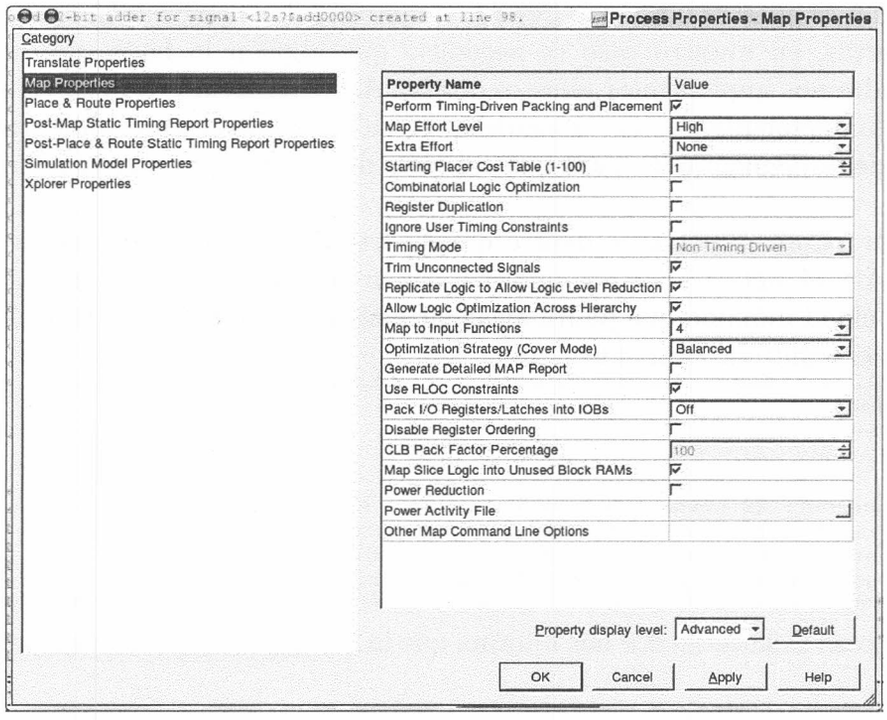


Figura 7.6: Ventana de propiedades del proceso de implementación

Al correr el proceso de implementación, este termina con errores informando que no hay recursos necesarios para implementar el diseño y que además no se cumplieron las restricciones temporales (en el archivo de restricciones se indica que el reloj sera de 50MHz). Pero existen muchos parametros que se pueden modificar en la configuración de este proceso que pueden ayudar a superar este problema. La figura 7.6 muestra la ventana de configuración del proceso de implementación.

Una propiedad interesante es *Map Slice Logic into Unused Block RAMs*, que habilita al proceso de implementación a ubicar lógica dentro de los Blocks Rams que no fueron utilizados. Por extraña que parezca esta propiedad, tiene sentido si pensamos que toda función lógica tiene una tabla de verdad asociada. Y buscar en una tabla de verdad es análogo a buscar en un banco de memoria. Así pues, activando esta propiedad confiamos que se pueda resolver el problema de área, mapeando la lógica excedente en bancos de memoria.

Otras propiedades que utilizaremos son *Optimization Estrategy* y *Map effort Level* que configuraremos respectivamente con los valores *SPEED* y *HIGH* para resolver los problemas temporales.

A continuación se muestran dos secciones del reporte final del proceso de implementación indicando que, se ha logrado implementar el diseño completo dentro del FPGA.

```
Device Utilization Summary:

   Number of BUFGMUXs                    5 out of 8      62%
   Number of DCMs                        1 out of 4      25%
   Number of External IOBs              86 out of 173    49%
      Number of LOCed IOBs              85 out of 86     98%

   Number of RAMB16s                    12 out of 12    100%
   Number of Slices                   1894 out of 1920   98%
      Number of SLICEMs                 45 out of 960    4%
```

Nótese que el proceso de síntensis indicaba que se necesitaban 6 Block Rams para implementar el diseño y el proceso de implementación indica que se utilizaron 12 Blocks Rams; pero el número de Slices necesarias paso de ser 116 % a utilizar el 98 % esto se debe a la propiedad *Map Slice Logic into Unused Block RAMs* activada en la configuración.

La siguiente tabla muestra el resultado del análisis temporal donde no se han detectado errores.

Constraint	Period Requirement	Actual Period		Timing Errors	
		Direct	Derivative	Direct	Derivative
TS_clkin	20.000ns	N/A	19.976ns	0	0
TS_Inst_clocks_CLK0_BUF	20.000ns	19.760ns	N/A	0	0
TS_Inst_clocks_CLK2X_BUF	10.000ns	9.988ns	N/A	0	0
TS_Inst_clocks_CLKFX_BUF	5.000ns	4.555ns	N/A	0	0

Así, el proceso de implementación ha logrado resolver los problemas encontrados durante el proceso de síntesis mediante una mejor utilización de los recursos disponibles (Blocks Rams) y dedicando una esfuerzo extra en la ubicación de los componente para disminuir el tiempo de propagación de las señales dentro del FPGA, logrando que el sistema cumpla con las restricciones temporales (Reloj principal de 50MHz)

El paso final es la generación del archivo de configuración del FPGA y su posterior descarga. Para esto se utiliza una herramienta que también provee el paquete IseWebPack llamada IMPACT.

Capítulo 8

Resultados y conclusiones

En este capítulo presentaremos los resultados obtenidos por nuestro diseño luego de haber sido sintetizado, implementado y programado en un Spartan 3 sobre la placa Starter Board Kit.

Para valorar los resultados obtenidos en este trabajo, también se citarán otros trabajos similares realizados en distintas tecnologías de hardware y software.

Y finalmente, se presentarán las conclusiones correpondientes y se dará una vision sobre posibles trabajos futuros para continuar abordando esta nueva tecnología de los FPGA.

8.1. Resultados

La implementación y ejecución del algoritmo ADM de detección de bordes nos entrega como resultado las imágenes de la figura 8.1b. Las tres imágenes seleccionadas para probar el diseño son: la tradicional Lena, un dibujo sintético realizado por ordenador y una fotografía con un fondo arbolado muy irregular (ver fig. 8.1a).

El análisis teórico del tiempo de procesamiento realizado en el capítulo 7 nos indicaba que este debía estar en el orden de los milisegundos, por lo tanto se decidió implementar la medición de tiempos con precisión de microsegundos, así el máximo error cometido en la medición es de 1μseg equivalente a 50 ciclos de reloj. El tiempo de procesamiento de las imágenes fue medido utilizando un contador de ciclos de reloj de 50Mhz y mostrando el resultado en los displays de 7 segmentos que provee la placa. Obteniéndose como resultado al procesar imágenes de 512×512 un tiempo total de 2673μseg. Esto nos permite procesar 374 imágenes por segundo.

Comparando el resultado teórico (2672940nseg) con el real medido en

(a)

(b)

Figura 8.1: (a) Imágenes originales. (b) Imágenes procesadas en la placa

la placa, se observa una diferencia de al menos 0,06μseg. Para facilitar el análisis, en el momento de calcular el tiempo teórico no se tuvieron en cuenta algunos factores temporales que se producen al comenzar y finalizar el proceso. Por ejemplo, cuando una señal se activa en un ciclo de reloj, esta es reconocida (en las máquinas sincrónicas) al siguiente ciclo, además las máquinas comienzan a funcionar al salir de los estados reposo, es decir un ciclo despues al reconocimiento de la señal de comienzo. Siendo la diferencia observada de apenas unos cuantos ciclos de reloj, la atribuimos entonces a estos ciclos no considerados.

8.2. Comparación de resultados

A continuación se presentaran resultados obtenidos en detección de bordes con diferentes tecnologías (Procesador, ASIC, FPGA). En cada trabajo, los algoritmos y el objetivo final del trabajo son diferentes y en consecuencia

es muy difícil comparar los desempeños alcanzados. Pero nuestra intención no es realizar una comparación rigurosa, sino obtener un valor que nos pueda ayudar a valorar el desempeño alcanzado por nuestra implementación. Así, en general, las imágenes procesadas en los siguientes trabajos son de tamaños distintos y las implementaciones en hardware funcionan con diferentes frecuencias de reloj. Se decidió entonces escalar los resultados obtenidos en cada trabajo a una configuración compatible con nuestro diseño, es decir, imágenes de 512×512 pixels de resolución y frecuencia de pipeline de 100MHz.

8.2.1. Procesamiento en Procesadores

Para evaluar el desempeño en un procesador se decidió correr un programa desarrollado en python, utilizando la Python Image Library (PIL), una librería de procesamiento de imágenes que, entre otras cosas, posee una función que nos permite detectar bordes. La detección de bordes que realiza es muy sencilla y simplemente consiste en la convolución de un kernel de 3×3. A continuación se muestra la clase que implementa dicho filtro:

```
##
# Edge-finding filter.

class FIND_EDGES(BuiltinFilter):
    name = "Find Edges"
    filterargs = (3, 3), 1, 0, (
        -1, -1, -1,
        -1,  8, -1,
        -1, -1, -1
        )
```

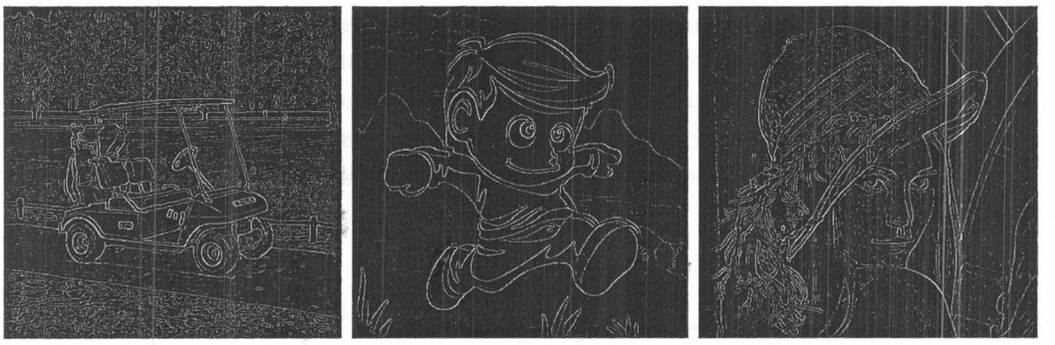
El filtro se corrió en una máquina con procesador *Intel(R) Xeon(R) CPU E5450 @ 3.00GHz* y 4Gb de memoria Ram obteniendo un resultado promedio en 100 corridas de: 0,00813734054565 segundos. La figura 8.2.1 muestra una comparación de las imágenes resultados obtenidas por nuestro diseño y por la PC. Así, esta aplicación puede procesar más de 122 imágenes por segundo.

8.2.2. Procesamiento en ASIC

En [3] se presenta el algoritmo ADM y su correspondiente implementación en un ASIC. En este caso, el algoritmo implementado es exactamente el mismo que el nuestro, ya que nosotros basamos nuestra implementación en ese trabajo.

A continuación se citan las conclusiones obtenidas en dicho trabajo.

"Presentamos un nuevo algoritmo de detección de bordes basado en una máscara de diferencias absolutas (ADM). El algoritmo es capaz de detectar bordes débiles y producir un borde localizado de un único pixel. El algoritmo tiene



(a)

(b)

Figura 8.2: Resultados de procesar (a) Sobre la placa. (b) en la PC

un estructura computacional muy regular que permite procesamiento paralelo y solapado (pipeline). Además el costo computacional es muy bajo si se compara con otros algoritmos de detección de bordes ya que no se utilizan operaciones de alta complejidad.

El algoritmo fue mapeado a una arquitectura VLSI implementada sobre lógica CMOS y fue testeado a diferentes niveles de frecuencia. El circuito es capaz de procesar 30 cuadros por segundo usando una frecuencia de reloj tan baja como 10Mhz. El chip ADM fue establecido y fabricado usando el proceso CMOS de doble-metal de $0,8\mu$m de Samsung."

Intentando escalar los resultados obtenidos podemos pensar que si el pipeline puede entregar 30 cuadros por segundo funcionando a una frecuencia de 10MHz, funcionando a 100MHz podría ser capaz de entregar alrededor de 300 imágenes por segundo.

8.2.3. Procesamiento en FPGA

En [4] miembros de la compañía Altera (fabricante de FPGAs) realizaron una implementación del algoritmo de detección de bordes de Canny utilizando comparativamente la misma cantidad de recursos. A continuación se citan sus conclusiones.

"La arquitectura presentada se implementa usando DSP Builder. Esta herramienta traduce la representación del diseño creado usando MATLAB/Simulink a código VHDL que luego será tomado por la herramientas de desarrollo Altera Quartus II para su implementación en un FPGA de Altera.

Los resultado preliminares muestran el diseño corriendo en un FPGA Stratix II a una frecuencia de 264Mhz. Usamos una imagen de prueba con resolución de 256×256. El módulo está totalmente construido en pipeline donde un pixel resultado es calculado en cada ciclo de reloj. Con esta tasa y ciclo de reloj, es posible procesar más de 4000 cuadros de 256×256. El diseño es escalable para manejar imágenes de mayor tamaño. Adaptando la frecuencia de reloj, se puede utilizar tanto para procesar señales de video estándar con una tasa de 30 cuadros por segundo, como para aplicaciones de visión donde se requiere más de 100 cuadros por segundo. Es posible aumentar la frecuencia más allá de 264Mhz introduciendo aún más etapas en el pipeline a costo de incrementar los recursos utilizados en el FPGA."

Analizando estos resultados, vemos que las imágenes que se procesan en este trabajo son 4 veces más pequeñas que las que nosotros utilizamos. Así, de procesar imágenes de 512×512 a 264MHz, obtendrían 1000 imágenes por segundo. Pero escalando también la frecuencia de reloj y llevándola a 100MHz, se procesarían 378 imágenes por segundo.

8.3. Conclusiones

A lo largo de este trabajo se ha construido un sistema embebido completo sobre un FPGA Spartan 3. Para cumplir con los requerimientos temporales del diseño y al no disponer de recursos internos de memoria suficientes, se diseñó y construyó una estructura de buffers alternativa que logre los objetivos deseado con el hardware disponible. Nuestro diseño fue publicado en el *IV Southern Programmable Logic Conference* [7]

También se evaluó el desempeño de otros trabajos y tecnologías donde se implementaban algoritmos de detección de bordes. Sus resultados fueron escalados a imágenes de 512×512 pixels de resolución y 100MHz de frecuencia de pipeline logrando así un valor estimativo teórico que muestra que el

resultado obtenido en nuestro desarrollo es competitivo.

La siguiente tabla muestra un resumen de estos resultados.

Tecnología	Cuadros por seg.	Algoritmo
PC	122	pasa altos
Asic	300	ADM
FPGA	378	Canny
Nuestro Diseño	*374*	*ADM*

Así la metodología propuesta, que sugiere implementar todos los módulos como pipelines o máquinas de estado separando el camino de datos del camino de control, nos condujo a un diseño que es sintetizable y que implementa un dispositivo capaz de competir y superar en tiempo de procesamiento a otras tecnologías del mercado actual.

En nuestro caso, el diseño ocupó casi el 100 % del área y la memoria de un Spartan 3, logrando así una utilización óptima de los recursos. Si por el contrario, la detección de bordes fuera solo uno de los varios procesos que se quiera aplicar a la imagen, entonces el resto de los módulos no podrían ser implementados. Actualmente Xilinx comercializa la familia Spartan 6 en la cual nuestro diseño solo ocuparía el 1,28 % del área disponible [12] sobrando más del 98 % del área para la implementación de más módulos de procesamiento. Entonces, siendo tan amplia la gama de FPGAs disponibles en el mercado, es importante saber elegir cual de ellas se ajusta a las necesidades de nuestro proyecto.

La aplicación de la metodología de diseño propuesta logra un buen balance entre la utilización de recursos y desempeño temporal, al mismo tiempo que consigue esconder en gran medida los aspectos relacionados con toda la electrónica subyacente, acercando esta nueva tecnología al ámbito computacional, pudiéndose de esta manera, encarar un proyecto desde una vision meramente algorítmica.

8.3.1. Trabajos Futuros

Existen aun muchas áreas de esta tecnología abiertas a la investigación. Los nuevos FPGA tienen embebidos microprocesadores dentro de su área de lógica que permiten mezclar software y hardware en los diseños. Las herramientas que asisten en el codiseño de hardware y software están en continuo desarrollo, por ejemplo el EDK de Xilinx o SystemC. Así, la posibilidad de construir o mejorar algoritmos que puedan optimizar o asistir en la generación de código HDL a partir de otros lenguajes que permiten modelar desde una vision sistémica.

Muchos algoritmos tradicionales pueden ser adaptados y reorganizados para ser embebidos aprovechando el gran paralelismo que ofrece esta tecnología.

Otro tema que no se ha abordado en profundidad en este trabajo, pero que resulta muy importante, es el de verificacion y simulacion de diseños. Ya que mas del 70 % del tiempo de desarrollo de una aplicación embebida se dedica a testing [1].

Entonces tanto desde el área de generación de nuevas herramientas como desde la construcción y adaptación de algoritmos se tiene por delante un vasto campo de investigación y desarrollo que espera ser explorado.

Apéndice A

Módulo de Entrada y Salida

La UART (*Universal Asynchronous Receiver and Transmitter*) es un circuito que envía y recibe datos paralelos a través de una linea serie. Las UARTs son frecuentemente usadas en conjunto con el estándar EIA (*Electronic Industries Alliance*) RS-232, el cual especifica las características eléctricas, funcionales y procedurales de dos dispositivos de comunicación de datos. El kit Spartan 3 tiene un puerto RS-232 con un conector estándar de 9 pines, debido a que los valores de tensión definidos en el protocolo RS-232 difieren de los niveles de tensión manejados por el FPGA, también incluye un chip de conversión de tensiones y manejo de señales de control de manera que solo quede por diseñar el circuito de la UART.

Una UART incluye un transmisor y un receptor. El transmisor es esencialmente un desplazador de registro que carga datos en paralelo y los va desplazando bit a bit a la salida en un período de tiempo determinado. El receptor, por otro lado, desplaza la entrada de datos bit a bit y ensambla el byte paralelo. La linea serie transmite '1' cuando esta inactiva, La transmisión comienza con un *start bit* que es un '0', seguido de los bits de datos y, opcionalmente bits de paridad, terminando con un *stop bit* que es un '1'. El número de bits de datos puede ser 6, 7 u 8. Los bits de paridad son utilizados para detectar errores de transmisión.

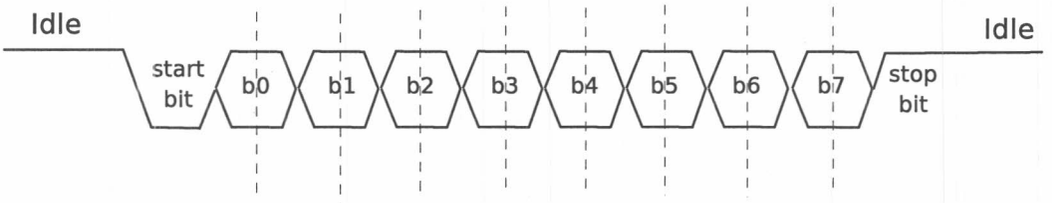


Figura A.1: Transmisión serie de datos

La figura A.1 muestra el diagrama de una transmisión serie de 8 bits de datos, sin paridad y 1 *stop bit*. Nótese que primero se transmiten los bits menos significativos

En la linea serie no se incluye información de sincronismo (reloj) por lo cual la tasa de transmisión de datos y otro parametros como la paridad y la cantidad de bits transmitidos deben ser acordados de antemando entre emisor y receptor.

A.1. Arquitectura

Este módulo debe encargarse de transferir una imagen desde la PC a la memoria principal y viceversa.

Una primera aproximación al diseño, podría ser ir procesando la imagen a medida que esta es transferida desde el exterior de manera de solapar el tiempo de adquisición con el de procesamiento. Pero, como se mencionó anteriormente, la interfaz serie no es una vía de comunicación adecuada para la adquisición de imágenes debido a su muy baja tasa de transferencia. En un caso real de uso, otras vías de adquisición serán requeridas y en consecuencia este módulo sería reemplazado. Por esta razón el módulo de entrada y salida se diseño de manera aislada, sin interacción con el resto del diseño.

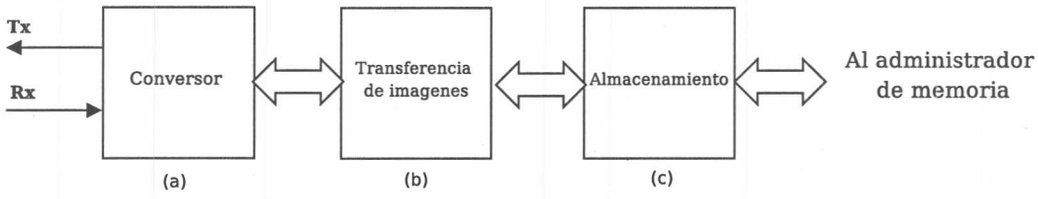


Figura A.2: Esquema del módulo Serie

Se propone entonces, transferir la imagen desde la PC hacia la memoria, una vez terminada la transferencia comienza el procesamiento de la imagen, y cuando este haya terminado, la imagen resultado guardada en memoria es transferida nuevamente a la PC. Esta arquitectura supone dos ventajas, primero este módulo es fácilmente reemplazable por otro que implemente una vía de transferencia mas adecuada. Segundo, facilita la medición de tiempos del módulo de procesamiento ya que este es independiente del tiempo de transferencia.

La funcionalidad requerida entonces consiste en a) convertir la información de serie a paralelo y viceversa, b) un protocolo de transferencia de imágenes y c) almacenamiento en memoria principal. La figura A.2 muestra el esquema del módulo de entrada y salida. Como se dispone de canales

independientes para la transmisión y recepción de la información, tanto en el canal serie como en el administrador de memoria el esquema de la figura A.2 puede dividirse en submódulos que traten por separado la transmisión y la recepción. Tendremos entonces en la etapa de conversión dos módulos, un conversor serie paralelo (CSP) y un conversor paralelo serie (CPS). En la etapa de transferencia de imagen habrá un receptor de imágenes (RIM) y un transmisor de imagen (TIM). Finalmente, un módulo almacenara la imagen en memoria (S2M) mientras que otro módulo leerá desde la memoria la imagen resultado (M2S).

A.1.1. Conversor

Como se mencionó anteriormente, los parametros de comunicación del módulo serie se deben establecen antes de efectuar la transmisión. En este caso, han sido fijados de la siguiente manera: la tasa de transferencia es de 115200 baudios, se transmiten palabras de 8 bits y no se utilizan bits de paridad. Así un bit deberá ser transmitido en el canal serie durante un tiempo $t = \frac{1}{115200}$. La placa SB3 tiene incorporado un cristal de 50MHz que se utiliza como reloj principal. Si expresamos el tiempo t en tic (flancos ascendentes) del reloj nos queda $t = \frac{50000000}{115200} \cong 434$ tics.

CSP: Conversor serie paralelo

Este módulo debe tomar los bits que se transmiten por el canal serie *(RX)*, y devolver una palabra de 8 bits a la salida *(dout)* cuando esta se haya terminado de transmitir *(drout)* o informar si se produjo un error de sincronismo durante la recepción de la palabra *(error)*.

Se define entonces la interfaz del módulo CSP de la siguiente manera:

Señal		E/S	Bus
clkin	Reloj principal del sistema	entrada	
reset	Inicializa el módulo	entrada	
RX	Señal de entrada serie	entrada	
dout	Salida paralela de datos	salida	8 bits
drout	Indica información disponible a la salida	salida	
error	Indica que hubo un error de sincronismo	salida	

Cuando el módulo de control detecta el flanco descendente en la señal *RX*, activa el timer *entimer* que habilita el módulo del timer, el cual generara solamente 10 tics a la frecuencia indicada de transmisión serie. Con cada tic del timer, el shift Register desplazara un bit su contenido y cargara el dato de entrada. Nótese que el shift register se activa en flancos ascendente del

timer y este comienza su secuencia en el estado bajo. El muestreo de cada bit serie se realiza justo en la mitad de su tiempo de transmisión, de esta forma se puede presumir que la señal se encuentra estabilizada disminuyendo la posibilidad tomar muestras erróneas.(Fig. A.4)

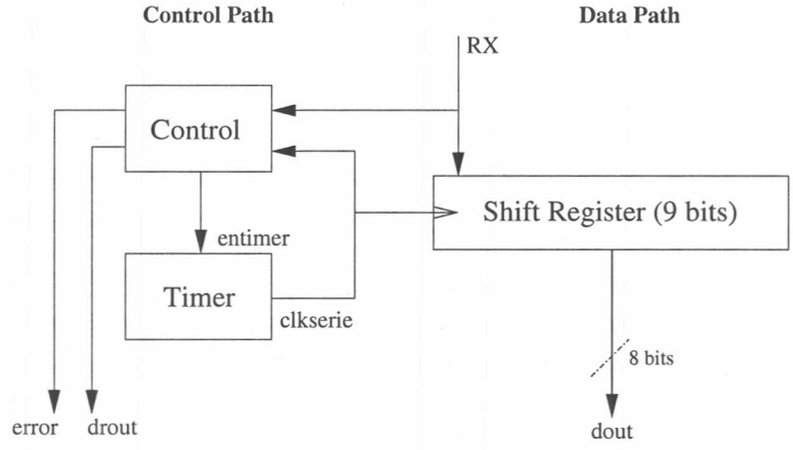


Figura A.3: Arquitectura del módulo CSP

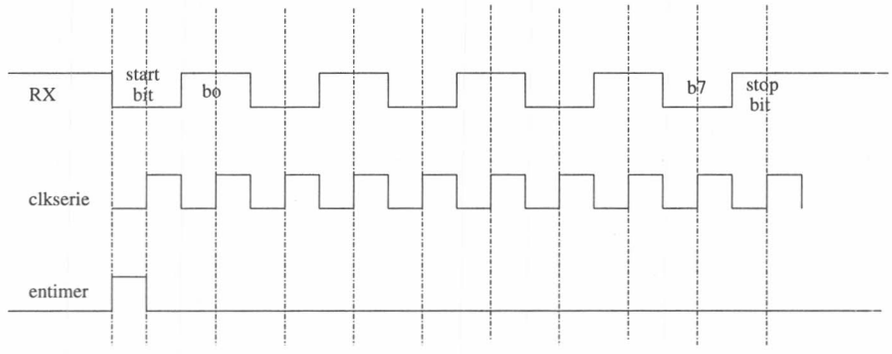


Figura A.4: Diagrama de Tiempo del módulo CSP

La figura A.5 muestra la máquina de estados que corresponde al funcionamiento del módulo. El canal serie indica que no se están enviando datos transmitiendo un 1, este estado es representado por **Idle**. Al no haberse transmitido datos, las señales de *drout* y *error* están en 0 indicando respectivamente que no hay datos disponibles ni se ha producido un error. La transmisión comienza con un flanco descendente en *RX* y se produce una transición de estados de **Idle** a **Start Bit** donde se habilita al timer *entimer*. La máquina permanecerá en el estado **Start Bit** hasta que se produzca el primer tic del timer *clkserie* para luego transitar al estado **bn**. En cada

tic del timer se ira cargando los bits de entrada en el Shift Register a la vez que la variable interna n se incrementa. Una vez se hayan producido 9 tics del reloj, el desplazador de registro estará totalmente cargado y se transitara el estado **Stop bit** donde se aguardara al ultimo tic del reloj y se analizara la señal RX, si es un 0, entonces no se esta muestreando un stop bit y en consecuencia hubo un error de sincronización en la transmisión, se transita al estado **Error** y la señal *error* se pone a 1. Si $RX = 1$, entonces se asume que se recibió la palabra completa sin errores y se transita al estado **OK** y se indica que hay un dato disponible a la salida mediante la señal *drout* en 1. Finalente se transita al estado *Idle* nuevamente a la espera de un nueva transmisión.

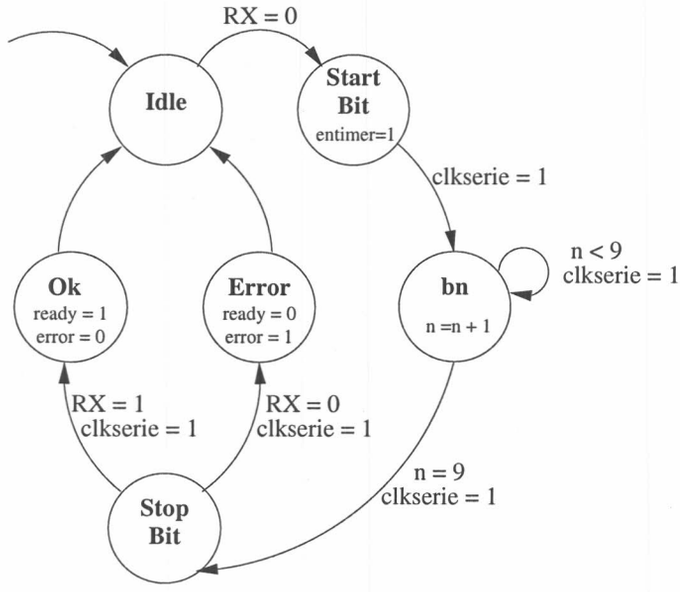


Figura A.5: Máquina de estados correspondiente al módulo de control

CPS: Conversor paralelo serie

Cuando este módulo es habilitado *(drin)* debe tomar la palabra de 8 bits de la entrada *(din)* y transmitirla por el canal serie *(TX)* bit a bit. Cuando la palabra completa de 8 bits fue transmitida, el módulo indica que esta libre para poder enviar otra palabra *(cts)*. Se define entonces la interfaz de este módulo de la siguiente manera:

Señal		E/S	Bus
clkin	Reloj principal del sistema	entrada	
reset	Inicializa el módulo	entrada	
TX	Señal de salida serie	salida	
din	Carga paralela de datos	entrada	8 bits
cts	Indica que esta libre para transmitir	salida	
drin	Comienza la transmisión	entrada	

De manera similar al módulo CSP, se utilizara un timer para sincronizar la transmisión del bit requerido al canal serie, esta vez, se cargara en el shift register una palabra de 10 bits compuesta por los 8 bits de datos mas los bits de comienzo y fin (Start y Stop bit). La señal interna *load* habilita la carga paralela del desplazador de registro, que irá desplazando los bits en cada flanco ascendente de la señal *clkserie*

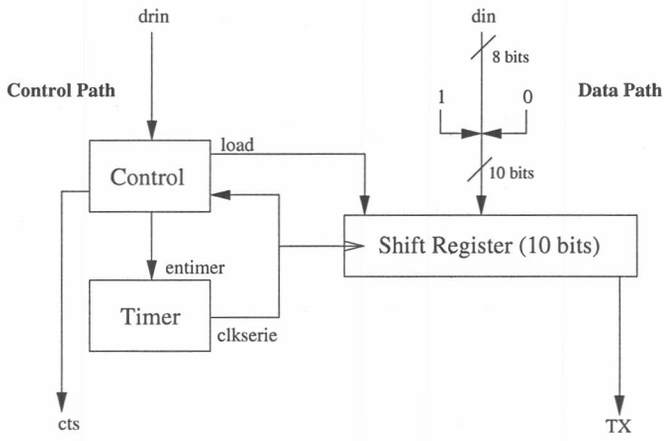


Figura A.6: Arquitectura del módulo CPS

La figura A.7 muestra la máquina de estados que corresponde al funcionamiento del módulo de control. Mientras no se estén transmitiendo datos, la señal *TX* se mantiene en 1, indicando que el canal serie esta inactivo y la señal *cts* también en 1 indica la disponibilidad para enviar datos. La transmisión comienza con un flanco ascendente en *drin* y se produce una transición de estados de **Idle** a **Carga**, la señal *cts* se pone en 0 indicando que se esta transmitiendo información y la señal *load* se pone a 1 para cargar los datos de entrada *din* en el shift register y se habilita el timer *entimer*. La máquina permanecerá en el estado **Carga** hasta que se produzca un tic del timer y transitara al estado **Trans** permaneciendo en este durante 10 tics del timer mientras el desplazador de registro transmite los bits al canal serie *TX*. Una vez enviados todos los 10 bits, la transmisión finaliza y se transita al estado **Idle** en el onceavo tic del timer.

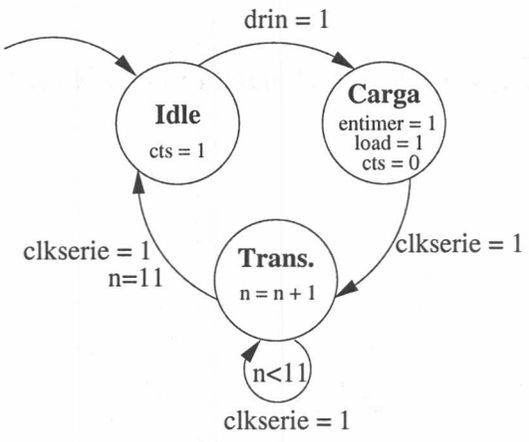


Figura A.7: Máquina de estados del módulo de control

A.1.2. Transmisión de imágenes

Este módulo se encarga de la transferencia de imágenes. Para esto, se implementó un protocolo muy simple que marca el comienzo y fin de cada imagen transferida. De esta manera, todo lo que este encerrado entre las marcas es considerada una imagen y el único control de error es que entre ambas marcas se hayan transferido la cantidad correcta de bytes. Las marcas utilizadas para indicar el comienzo y fin de la imagen son bytes que por la naturaleza de la imagen pueden también pertenecer a la misma, por lo tanto se incorporó otra marca que indica que el próximo byte transmitido es un dato y no un carácter de control. El cuadro A.1 muestra los caracteres utilizados.

Hex	Char	Funcion
02	SOF	Comienzo de Transmisión
03	EOF	Fin de Transmisión
10	DLE	Data Link Escape

Cuadro A.1: Caracteres de Control

RIM: Recepción de imágenes

Este módulo recibe del módulo CSP los bytes *(din)* uno a uno a medida que están disponibles *(drin)*. El primer byte transmitido deberá ser un SOF, de ser así, la señal *busy* se encenderá indicando que se comenzó con la transferencia de una imagen. Cada byte que corresponda a un pixel de la imagen será propagado a la salida *dout* y los bytes que correspondan a caracteres

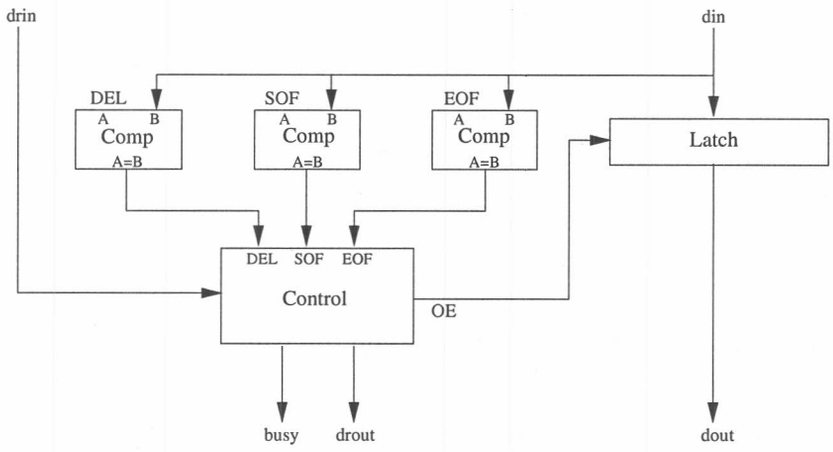


Figura A.8: Arquitectura del módulo RIM

de control serán descartados. La transferencia termina cuando se recibe un EOF.

Se define entonces la interfaz de este módulo de la siguiente manera:

Señal		**E/S**	**Bus**
clkin	Reloj principal del sistema	entrada	
reset	Inicializa el módulo	entrada	
drin	Datos de entrada disponible	entrada	
din	Bus de datos de entrada	entrada	8 bits
drout	Datos de salida disponibles	salida	
dout	Bus de datos de salida	salida	8 bits
busy	Indica que se esta recibiendo una imagen	salida	

La figura A.9 muestra la máquina de estados del módulo de control. Esta comienza con un flanco ascendente de *drin*, indicando que hay datos disponibles, y transita al estado **Inicio** si el dato corresponde a la marca de comienzo de transmisión SOF. Una vez que *drin* se haya apagado se pasa al estado **Análisis**. La señal *drin* sincroniza este módulo con el CSP y es la que determina el cambio de estados. En el estado de **Análisis** se verifica si el byte recibido es un pixel de la imagen o un carácter de control. Si es un pixel, se propaga directamente a la salida mediante el estado **Enviando**. Si es un caracter de control (DLE), entonces se espera al próximo byte transitando al estado **Escape**. Cuando el nuevo byte este disponible, se propagará directamente a la salida **OE=1** y se volverá al estado de **Análisis** a la espera del próximo byte. Cuando el byte recibido sea un EOF se transita al estado **Idle** y se finaliza la recepción.

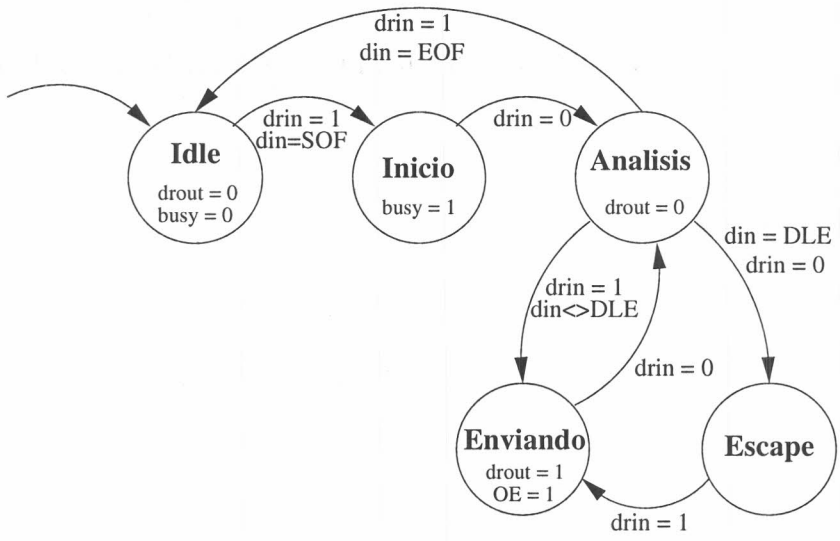


Figura A.9: Máquina de estados del módulo RIM

TIM: Transmisión de imágenes

Este módulo se encarga de transferir la imagen resultado hacia la PC. Cuando es habilitado *(en)* transmite el caracter de control de comienzo de transmisión (SOF), y comunica que esta disponible para recibir información de la memoria *(cts)*. Cuando la memoria tiene un byte disponible *(drin)* comienza a transmitir codificando, según corresponda, los pixels anteponiendo un DLE. Esta máquina es un poco más compleja que la anterior puesto que el destino donde se transmite la información (puerto serie) es más lento que el origen (memoria), en consecuencia, debe contener la llegada de datos desde la memoria para compensar velocidades *(cts,rts)*.

Se define entonces la interfaz de este módulo de la siguiente manera:

Señal		E/S	Bus
clkin	Reloj principal del sistema	entrada	
reset	Inicializa el módulo	entrada	
drin	Datos de entrada disponible	entrada	
din	Bus de datos de entrada	entrada	8 bits
cts	Indica que esta libre para recibir datos	salida	
drout	Datos de salida disponibles	salida	
dout	Bus de datos de salida	salida	8 bits
rts	Indica que se pueden transmitir datos	entrada	
en	Comienzo de transmisión de una imagen	entrada	

La figura A.10 muestra la arquitectura definida para este módulo, donde

los comparadores y el multiplexor conforman el camino de datos y el camino de control esta dado por el módulo de control definido por la máquina de estados de la figura A.11

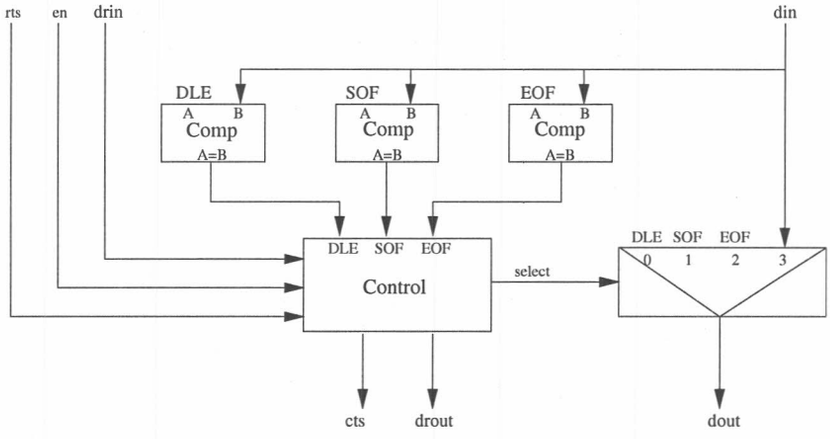


Figura A.10: Arquitectura del módulo TIM

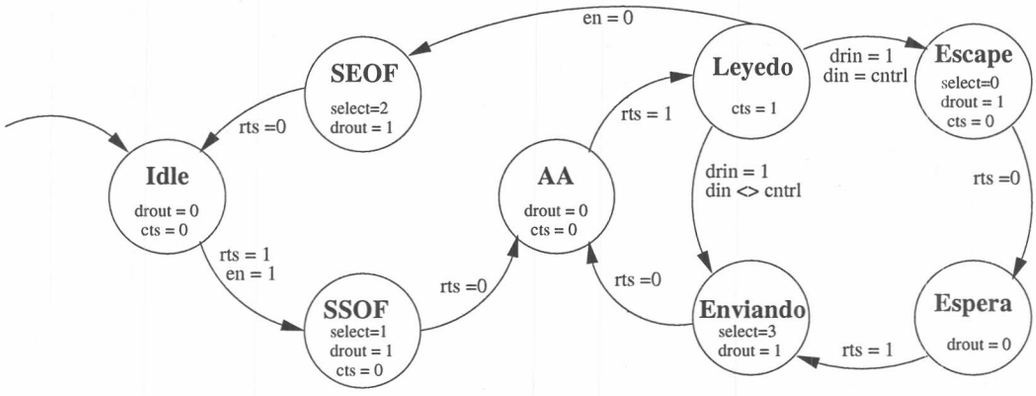


Figura A.11: Máquina de estados del módulo TIM

A.1.3. Almacenamiento

Como se mencionó anteriormente, por el puerto serie se transmite una palabra de 8 bits, mientras que la transferencia a memoria se realiza con palabras de 32 bits. Este módulo se encarga de convertir el largo de palabra y de almacenar la imagen en una posición especifica de la memoria, como también de leer la imagen resultado desde otra ubicación en memoria.

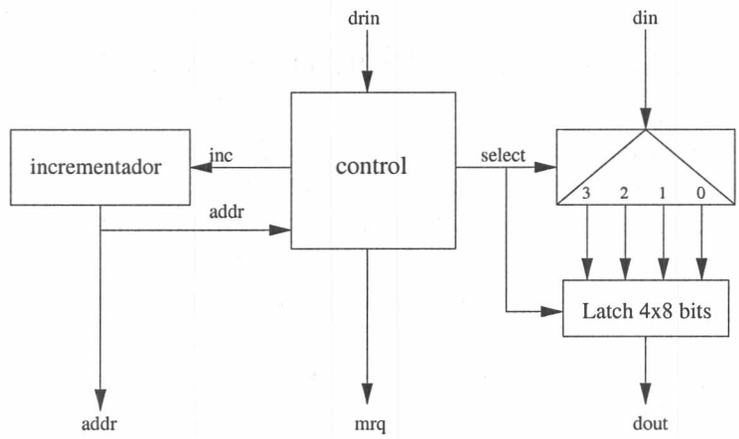


Figura A.12: Arquitectura del módulo S2M

S2M: Interfaz de escritura en memoria

A medida que en la entrada *din* se encuentran datos disponibles *drin*, se van cargando y almacenando en registros hasta obtener la palabra de 4 bytes para enviar a la memoria. Para los accesos a memoria necesitamos una señal para indicar que se desea comenzar un ciclo de memoria *mrq*, el bus de datos *dout* y el bus de direcciones *addr*. La interfaz queda definida entonces como:

Señal		E/S	Bus
clkin	Reloj principal del sistema	entrada	
reset	Inicializa el módulo	entrada	
drin	Datos de entrada disponible	entrada	
din	Bus datos de entrada	entrada	8 bits
addr	Bus de direcciones de memoria	salida	18 bits
dout	Bus datos de salida	salida	32 bits
mrq	Requerimiento de acceso a memoria	salida	
en	Habilitación del módulo	entrada	

La figura A.13 muestra la máquina de estado que implementa este módulo. Esta comienza cuando es habilitada *(en)* y hay un dato disponible en la entrada *(drin)* transitando del estado **Idle** al estado **Leyendo** donde se lee un dato de entrada *(din)*. Una vez que el dato fue almacenado en un registro *(dout(count)=din)*, se transita al estado **Espera** hasta la llegada de un nuevo dato. Como los datos de entrada son de 8 bits, la máquina cicla entre estos dos estados **Leyendo** y **Espera** hasta que haya recibido los 4 bytes que necesita para completar la palabra de 32 bits. Una vez recibidos todo los bytes, transita al estado **Enviar**, donde inicia el ciclo de memoria *(mrq=1)* y

vuelve al estado **leyendo**, así continua ciclando hasta que se hayan transferido todos los datos en memoria *(en=0)*. Como el ciclo de memoria es mucho menor a la velocidad de transferencia de datos del puerto serial, se asume que es imposible que se pierdan datos de entrada mientras se realiza un ciclo de memoria, es por eso que esta condición nunca se verifica en la máquina de estados.

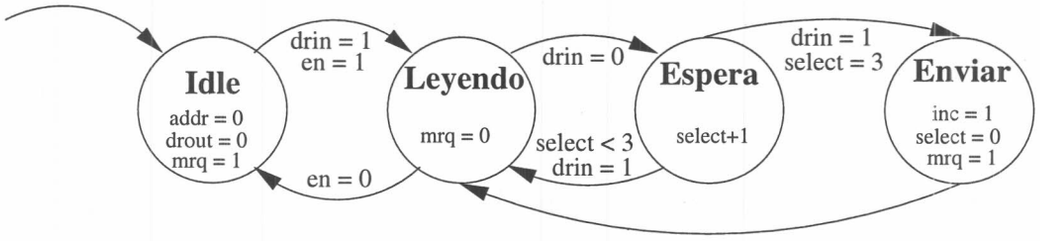


Figura A.13: Máquina de estados del módulo S2M

M2S: Interfaz de lectura en memoria

Este módulo, al igual que la anterior, debe comunicarse con la memoria *addr, din, mrq*. También debe enviar datos, esta vez, la conversión de palabra se realiza de 32 bits a 8 bits y como el dispositivo que los recibe es mas lento que la memoria, se agrega una señal de control para sincronizar el envío de datos*(rts)* Se define entonces la interfaz de este módulo de la siguiente manera:

Señal		E/S	Bus
clkin	Reloj principal del sistema	entrada	
reset	Inicializa el módulo	entrada	
din	Bus datos de entrada	entrada	32 bits
addr	Bus de direcciones de memoria	salida	18 bits
drout	Datos de salida disponibles	salida	
dout	Bus datos de salida	salida	8 bits
mrq	Requerimiento de acceso a memoria	salida	
en	Habilitación del módulo	entrada	

La figura A.15 muestra el diseño del control de este módulo. La máquina comienza cuando es habilitada *(en=1)* e inmediatamente pasa al estado **Leyendo** donde comienza un ciclo de memoria para luego transitar al estado **Enviar**. De la memoria se han leído 31 bits (4 bytes) que deberán ser entregados byte a byte al próximo módulo, así en el estado **Enviar** se activa la señal *(drout)* avisando que se tiene un byte disponible y se transita al estado

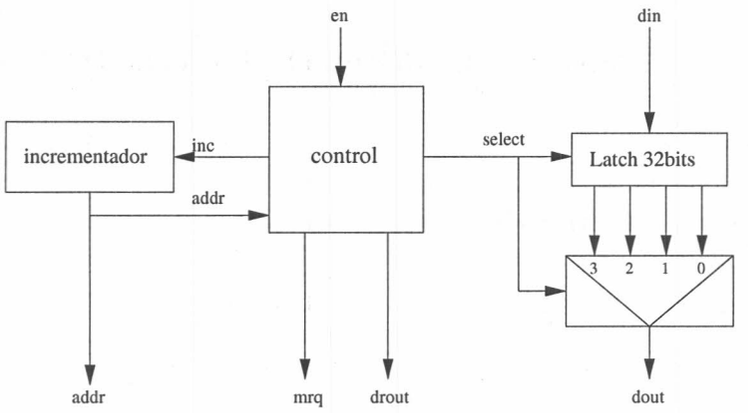


Figura A.14: Arquitectura del módulo M2S

Espera. Cuando la señal *rts* se activa (indicando que se puede transmitir otro byte) la máquina pasa al estado **Enviar** nuevamente para transmitir un nuevo byte. Se continúa ciclando entre **Enviar** y **Espera** hasta transmitir los 4 bytes (*select*=4) recibidos de la memoria antes de volver al estado **Leyendo** y comenzar otro ciclo de memoria. La máquina termina cuando fue transmitida toda la imagen resultado *addr=65536*

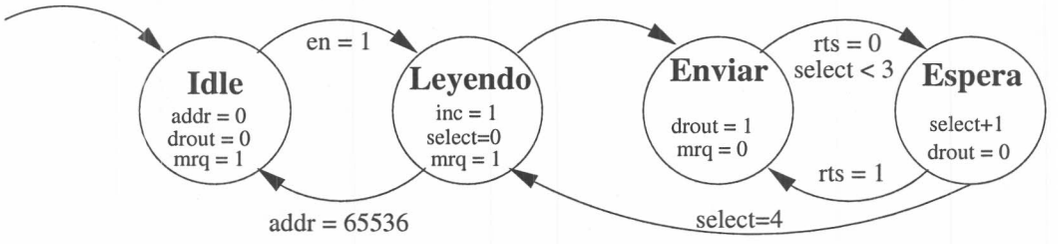


Figura A.15: Máquina de estados del módulo M2S

ES: Modulo de Entrada Salida

Finalmente el módulo ES queda formado por la interconexión de todos lo módulos precedentes. Así, su interfaz queda definida por las señales del puerto serial y por la interfaz a memoria como se muestra a continuación:

Señal		E/S	Bus
TX	Linea de transmisión del canal serie	salida	
RX	Linea de recepción del canal serie	entrada	
clkin	Reloj del sistema	entrada	
reset	Reinicializa el módulo	entrada	
waddr	Bus de direcciones de escritura en memoria	salida	18 bits
wdata	Bus de datos de escritura en memoria	salida	32 bits
wmrq	Requerimiento de escritura en memoria	salida	
raddr	Bus de direcciones de lectura en memoria	salida	18 bits
rdata	Bus de datos de lectura en memoria	entrada	32 bits
rmrq	Requerimiento de lectura en memoria	salida	
loading	Comienzo de recepción de imagen	salida	
sending	Comienzo de transmisión de imagen	entrada	
error	Error de sincronismo en la recepción serie	salida	

La figura A.16 muestra la interconexión de cada uno de los submódulos que conforman el módulo de entrada y salida.(ES).

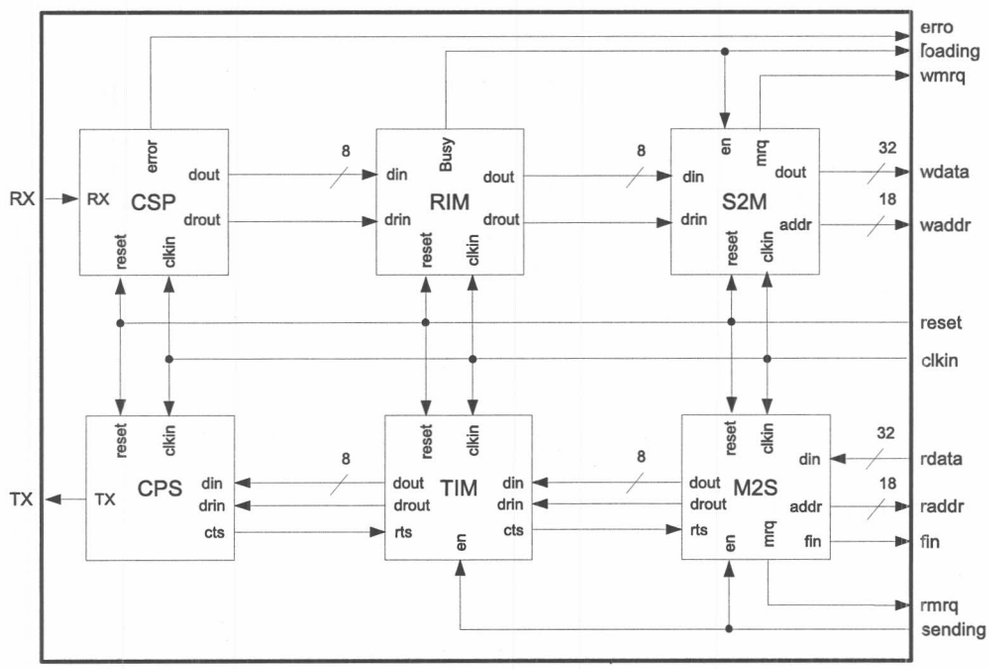


Figura A.16: Arquitectura del módulo de entrada y salida

Apéndice B

Codigo

Listing B.1: Modulo Administrado de Memoria

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity memmgr is
    Port ( clkin : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           addrW : in  STD_LOGIC_VECTOR (17 downto 0);
           addrR : in  STD_LOGIC_VECTOR (17 downto 0);
           dataW : in  STD_LOGIC_VECTOR (31 downto 0);
           dataR : out  STD_LOGIC_VECTOR (31 downto 0);
           rqR : in  STD_LOGIC;
           rqW : in  STD_LOGIC;
           addrBus : out  STD_LOGIC_VECTOR (17 downto 0);
           dataBus : inout  STD_LOGIC_VECTOR (31 downto 0);
           we : out  STD_LOGIC;
           oe : out  STD_LOGIC;
           lb0 : out  STD_LOGIC;
           lb1 : out  STD_LOGIC;
           ub0 : out  STD_LOGIC;
           ub1 : out  STD_LOGIC;
           ce0 : out  STD_LOGIC;
           ce1 : out  STD_LOGIC);
end memmgr;

architecture Behavioral of memmgr is

type estados is (mread,mwrite,iniW,wsW,endW,IniR,wsR,endR);
signal estado,prox_estado:estados;
signal reg_dataread,prox_dataread,datawrite:std_logic_vector(31 downto 0);
signal addressW,addressR: std_logic_vector(17 downto 0);
signal rw,rrqR,rrqW,prox_rrqR,prox_rrqW : std_logic;

begin
ce0<='0';
ce1<='0';
lb0<='0';
lb1<='0';
ub0<='0';
```

```
ub1<='0';


dataR   <= prox_dataread;
databus   <= datawrite when rw='0' else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
addrbus   <= addressW when rw='0' else addressR;

process(rqR)
begin
  if rqR='1' and rqR'event then
    addressR<=addrR;
  end if;
end process;

process(rqW)
begin
  if rqW='1' and rqW'event then
    addressW<=addrW;
    datawrite<=dataW;
  end if;
end process;


--
-- Registro de Estado
--

process(clkin)
begin
  if reset='1' then
    estado<=mread;
    rrqR<='0';
    rrqW<='0';
  elsif clkin='1' and clkin'event then
    reg_dataread<=prox_dataread;
    estado<=prox_estado;
    rrqR<=prox_rrqR;
    rrqW<=prox_rrqW;
  end if;
end process;

--
-- Funcion de cambio de estado
--

process (estado,rqW,rqR)
begin
    case estado is
      when mread  =>  prox_estado<=mread;
                if rqR='1' or rrqR='1' then
                  prox_estado<=IniR;
                elsif rqW='1' or rrqW='1'then
                  prox_estado<=IniW;
                end if;

      when mwrite =>  prox_estado<=mwrite;
                if rqW='1' or rrqW='1' then
                  prox_estado<=IniW;
                elsif rqR='1' or rrqR='1' then
                  prox_estado<=IniR;
                end if;
```

```
        when IniR   => prox_estado<=wsR;

        when wsR    => prox_estado<=endR;

        when endR =>  prox_estado<=mwrite;


        when IniW   => prox_estado<=wsW;

        when wsW    => prox_estado<=endW;

        when endW => prox_estado<=mread;

    end case;
end process;

--
--Salidas
--

process(estado,rqR,rqW)
begin
  prox_dataread<=reg_dataread;
  prox_rrqR<=rrqR;
  prox_rrqW<=rrqW;
  case estado is

    when mread  =>  oe<='1';
                    we<='1';
                    rw<='1';
                    if rqR='1' or rrqR='1' then
                      rw<='1';
                    elsif rqW='1' or rrqW='1' then
                      rw<='0';
                    end if;

    when mwrite =>  oe<='1';
                    we<='1';
                    rw<='0';
                    if rqW='1' or rrqW='1' then
                      rw<='0';
                    elsif rqR='1' or rrqR='1' then
                      rw<='1';
                    end if;

    when IniR   =>  oe<='0';
                    we<='1';
                    rw<='1';

    when wsR  => oe<='0';
                    we<='1';
                    rw<='1';
                    prox_rrqR<='0';
                    prox_rrqW<=rqW;

    when endR   => oe<='0';
                    we<='1';
                    rw<='1';
                    prox_dataread<=databus;

    when IniW   => oe<='1';
                    we<='0';
```

```
                rw<='0';

    when wsW    => oe<='1';
                we<='0';
                rw<='0';
                prox_rrqR<=rqR;
                prox_rrqW<='0';

    when endW   => oe<='1';
                we<='0';
                rw<='0';

  end case;
end process;
end Behavioral;
```

Listing B.2: Modulo Calculo del proceso de filtrado

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity FI is
    Port (
        clkin : in STD_LOGIC;
        readyin : in STD_LOGIC;
        readyout : out STD_LOGIC;
        datain : in  STD_LOGIC_VECTOR (199 downto 0);
        dataout : out  STD_LOGIC_VECTOR (7 downto 0);
        dir : out  STD_LOGIC_VECTOR (7 downto 0)
);
end FI;

architecture Behavioral of FI is

signal  v,vu,vl,
      h,hu,hl,
      pd,pdu,pdl,
      nd,ndu,ndl,
      se,de,
      max1,max2,
      min1,min2:std_logic_vector (8 downto 0);
signal rl1,rl2,rl3,Vmayor,Pdmayor : std_logic;
signal dato1,dato2,dato3,res : std_logic_vector (7 downto 0);
begin
---
--- Tabla de referencia
---
--
--1)  (199 ~ 192) 2)  (159 ~ 152) 3)  (119 ~ 112) 4)  (79 ~ 72) 5)  (39 ~ 32)
--6)  (191 ~ 184) 7)  (151 ~ 144) 8)  (111 ~ 104) 9)  (71 ~ 64) 10) (31 ~ 24)
--11) (183 ~ 176) 12) (143 ~ 136) 13) (103 ~  96) 14) (63 ~ 56) 15) (23 ~ 16)
--16) (175 ~ 168) 17) (135 ~ 128) 18) ( 95 ~  88) 19) (55 ~ 48) 20) (15 ~  8)
--21) (167 ~ 160) 22) (127 ~ 120) 23) ( 87 ~  80) 24) (47 ~ 40) 25) ( 7 ~  0)
--
-- Nivel 1
--
process (clkin)
begin
  if clkin'event and clkin='1' then
```

```
    vu <= ('0'&datain (119 downto 112))+('0'&datain (111 downto 104));
    vl <= ('0'&datain ( 95 downto  88))+('0'&datain ( 87 downto  80));
    hu <= ('0'&datain (183 downto 176))+('0'&datain (143 downto 136));
    hl <= ('0'&datain (63 downto 56))  +('0'&datain (23 downto 16));
    pdu <= ('0'&datain(199 downto 192))+('0'&datain (151 downto 144));
    pdl <= ('0'&datain (55 downto 48)) +('0'&datain ( 7 downto  0));
    ndu <= ('0'&datain (39 downto 32)) +('0'&datain (71 downto 64));
    ndl <= ('0'&datain(135 downto 128))+('0'&datain (167 downto 160));

    dato1<=datain(103 downto  96);
    rl1<=readyin;
  end if;
end process;

---
--- Nivel 2
---
process (clkin)
begin
  if clkin'event and clkin='1' then
    if vu > vl then
      v<=vu-vl;
    else
      v<=vl-vu;
    end if;

    if hu > hl then
      h<=hu-hl;
    else
      h<=hl-hu;
    end if;

    if pdu > pdl then
      pd<=pdu-pdl;
    else
      pd<=pdl-pdu;
    end if;

    if ndu > ndl then
      nd<=ndu-ndl;
    else
      nd<=ndl-ndu;
    end if;

    dato2<=dato1;
    rl2<=rl1;
  end if;
end process;

---
--- Nivel 3
---
process (clkin)
begin
  if clkin'event and clkin='1' then
  --C1
    if v < h then
      max1<=h;
      min1<=v;
      vmayor<='0';
    else
      max1<=v;
```

```
      min1<=h;
      vmayor<='1';
    end if;
  --C2
    if pd < nd then
      max2<=nd;
      min2<=pd;
      Pdmayor<='0';
    else
      max2<=pd;
      min2<=nd;
      Pdmayor<='1';
    end if;
    dato3<=dato2;
    rl3<=rl2;
  end if;
end process;

---
--- Nivel 4
---

process (clkin)
begin
  if clkin'event and clkin='1' then
    if max1 < max2 then
      dataout <=max2(8 downto 1);
    else
      dataout <=max1(8 downto 1);
    end if;

    if min1 < min2 then
      dir <= "000000"&Vmayor&'0';
    else
      dir <= "000000"&Pdmayor&'1';
    end if;
    readyout<=rl3;
  end if;
end process;
end Behavioral;
```

Apéndice C

Archivo de Restricciones

Listing C.1: Archivo tesis.ucf

```
NET "clkin" TNM_NET = "clkin";
TIMESPEC "TS_clkin" = PERIOD "clkin" 20 ns HIGH 50 %;
NET "addrbus<0>"  LOC = "L5"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<10>"  LOC = "G5"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<11>"  LOC = "H3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<12>"  LOC = "H4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<13>"  LOC = "J4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<14>"  LOC = "J3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<15>"  LOC = "K3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<16>"  LOC = "K5"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<17>"  LOC = "L3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<1>"  LOC = "N3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<2>"  LOC = "M4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<3>"  LOC = "M3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<4>"  LOC = "L4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<5>"  LOC = "G4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<6>"  LOC = "F3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<7>"  LOC = "F4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<8>"  LOC = "E3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "addrbus<9>"  LOC = "E4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "ce0"  LOC = "p7"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "ce1"  LOC = "n5"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "clkin"  LOC = "T9"  ;
NET "conf<0>"  LOC = "f12"  ;
NET "conf<1>"  LOC = "g12"  ;
NET "conf<2>"  LOC = "h14"  ;
NET "conf<3>"  LOC = "h13"  ;
NET "conf<4>"  LOC = "j14"  ;
NET "conf<5>"  LOC = "j13"  ;
NET "conf<6>"  LOC = "k14"  ;
NET "conf<7>"  LOC = "k13"  ;
NET "databus<0>"  LOC = "N7"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<10>"  LOC = "F2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<11>"  LOC = "H1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<12>"  LOC = "J2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<13>"  LOC = "L2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<14>"  LOC = "P1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<15>"  LOC = "R1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<16>"  LOC = "P2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<17>"  LOC = "N2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
```

```
NET "databus<18>"  LOC = "M2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<19>"  LOC = "K1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<1>"  LOC = "T8"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<20>"  LOC = "J1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<21>"  LOC = "G2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<22>"  LOC = "E1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<23>"  LOC = "D1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<24>"  LOC = "D2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<25>"  LOC = "E2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<26>"  LOC = "G1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<27>"  LOC = "F5"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<28>"  LOC = "C3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<29>"  LOC = "K2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<2>"  LOC = "R6"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<30>"  LOC = "M1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<31>"  LOC = "N1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<3>"  LOC = "T5"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<4>"  LOC = "R5"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<5>"  LOC = "C2"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<6>"  LOC = "C1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<7>"  LOC = "B1"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<8>"  LOC = "D3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "databus<9>"  LOC = "P8"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "error"  LOC = "p12"  ;
NET "filtering"  LOC = "p13"  ;
NET "lb0"  LOC = "p6"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "lb1"  LOC = "p5"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "loading"  LOC = "p11"  ;
NET "oe"  LOC = "k4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "reset"  LOC = "l14"  ;
NET "RX"  LOC = "t13"  ;
NET "sending"  LOC = "n12"  ;
NET "TX"  LOC = "r13"  ;
NET "ub0"  LOC = "t4"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "we"  LOC = "g3"  | SLEW = FAST  | IOSTANDARD = LVCMOS33;
NET "control<0>"  LOC = "d14"  ;
NET "control<1>"  LOC = "g14"  ;
NET "control<2>"  LOC = "f14"  ;
NET "control<3>"  LOC = "e13"  ;
NET "dsplines<0>"  LOC = "p16"  ;
NET "dsplines<1>"  LOC = "n16"  ;
NET "dsplines<2>"  LOC = "f13"  ;
NET "dsplines<3>"  LOC = "r16"  ;
NET "dsplines<4>"  LOC = "p15"  ;
NET "dsplines<5>"  LOC = "n15"  ;
NET "dsplines<6>"  LOC = "g13"  ;
NET "dsplines<7>"  LOC = "e14"  ;
```

Bibliografía

[1] Janick Bergeron. *Writing Testbenches using SystemVerilog.* Springer Science+Business Media, Inc., 2006.

[2] Pong P. Chu. *FPGA Prtotyping by VHDL Examples.* Wiley-Interscience, 2008.

[3] Tom Chen Fahad M. Alzahrani. A real-time edge detector: Algorithm and vlsi architecture. In *Real Time Imaging 3*, pages 363–378, 1997.

[4] Asher Hazanchuk Hong ShanÑeoh. Adaptive edge detection for real-time video processing using fpgas. Technical report, Altera Corporation.

[5] Integrated Silicon Solution, Inc (ISSI). *IS61LV25616AL*, 2006.

[6] Clive Maxfield. *The Design Warrior's Guide to FPGAs.* ELSEVIER, 2004.

[7] Patricia Borensztejn Maximiliano A. Sacco. Implementacion parcial del algoritmo adm para deteccion de bordes. In *IV Southern Conference on Programmable Logic*, 2008.

[8] App. Notes. Using digital clock managers (dcms) in spartan-3 fpgas. Technical report, Xilinx, 2006.

[9] Daniel Borrajo Pedro Isasi, Paloma Martínez. *Lenguajes, Gramaticas y Automatas.* Addison-Wesley, 1997.

[10] Jim Turley. Embedded processors by the numbers. *Embedded System Programming*, 1999.

[11] Xilinx. Spartan-3 starter kit board user guide.

[12] Xilinx. Spartan-6 family overview.

[13] Xilinx. University program.

[14] Xilinx. Xst user guide.

[15] Xilinx. Using block ram in spartan-3 generation fpgas, Marzo 2005.