Sistemas Informáticos
Curso 2005-2006

Estudio y diseño con hardware reconfigurable de un sistema de análisis citológico

Iván Aldea López
María Bertrán de Lis Martín-Artajo

Dirigido por:
Prof. Daniel Mozos Muñoz
Dpto. ACYA

Facultad de Informática
Universidad Complutense de Madrid

I. Índice

I. Índice ..... - 2 -
II. Tabla de figuras ..... - 3 -
III. Palabras Clave ..... - 4 -
IV. Resumen ..... - 5 -
V. Abstract ..... - 5 -
VI. Autorización ..... - 6 -
1. Descripción del problema ..... - 7 -
1.1. Base biológica ..... - 7 -
1.2. Problema de la comparación de parámetros ..... - 8 -
2. Programación ..... - 10 -
2.1. Aproximación con el Lenguaje Perl ..... - 10 -
2.2. Aproximación con el Lenguaje C ..... - 12 -
2.3. Comparación entre ambos programas ..... - 17 -
3. Implementación Hardware ..... - 18 -
3.1. FPGAs. Descripción general ..... - 18 -
3.2. FPGAs. Descripción específica ..... - 24 -
3.3. Sistema combinacional para la comparación de células ..... - 28 -
3.4. Memoria ..... - 30 -
3.5. Sistema Completo ..... - 35 -
4. Programas utilizados ..... - 43 -
4.1. XSTOOLS ..... - 43 -
4.2. Xilinx ISE 7.1 ..... - 45 -
4.3. VSystem ..... - 46 -
5. Posibles mejoras ..... - 51 -
6. Bibliografía ..... - 53 -

II. Tabla de figuras

Figura 1: Lista doblemente enlazada..... - 13 -
Figura 2: Tabla ejemplo de pares de células similares..... - 15 -
Figura 3: Tabla ejemplo de células con sus clusters correspondientes.. - 16 -
Figura 4: Estructura básica del CLB ..... - 19 -
Figura 5: CLB de Xilinx con 2 LUT (Spartan II) ..... - 21 -
Figura 6: Estructura básica de un GRM..... - 22 -
Figura 7: Dos unidades lógicas de ALTERA comunicadas por Fast-Tracking..... - 23 -
Figura 8: Bloque de entrada/salida (Spartan II)..... - 24 -
Figura 9: CLB de la familia Spartan II..... - 25 -
Figura 10: Diagrama de bloques de una FPGA de la familia Spartan II - 26 -
Figura 11: Placa XSA 100..... - 27 -
Figura 12: Comparador de dos células..... - 28 -
Figura 13: Comparador de dos parámetros ..... - 29 -
Figura 14: Interfaz del controlador de SDRAM de la placa XSA ..... - 31 -
Figura 15: Operación de lectura en modo ráfaga de un controlador SDRAM ..... - 33 -
Figura 16: Operación de escritura en modo ráfaga de un controlador SDRAM..... - 34 -
Figura 17: Esquema de la distribución de la memoria de datos ..... - 36 -
Figura 18: Diagrama de la primera parte de la máquina de estados ..... - 38 -
Figura 19: Diagrama de la segunda parte de la máquina de estados ..... - 40 -
Figura 20: Diagrama de la tercera parte de la máquina de estados..... - 42 -
Figura 21: gxtest..... - 43 -
Figura 22: gxssetclk ..... - 44 -
Figura 23: gxsload..... - 44 -
Figura 24: Cambiar directorio de trabajo de V-System ..... - 47 -

Figura 25: Crear una librería Vhdl en V-System ..... - 47 -
Figura 26: Crear un nuevo proyecto en V-System..... - 48 -
Figura 27: Simulación de un diseño Vhdl con V-System..... - 48 -
Figura 28: Entorno de simulación de V-System ..... - 49 -

III. Palabras Clave

FPGA
Célula
Xilinx
Vhdl
Comparaciones
Paralelo
Complejidad
SDRAM
Cluster

IV. Resumen

El objetivo fundamental de este proyecto es el estudio y diseño en hardware reconfigurable de un sistema completo de análisis de datos citológicos, que se basa en comparar ciertos parámetros característicos de las células sanguíneas para un gran conjunto de ellas. Por medio de estos agrupamientos se pueden detectar clusters de células con parámetros similares, lo que permite al especialista detectar anomalías en el paciente.

Este proceso se ha realizado tradicionalmente de un modo manual, aunque ayudado por un computador. La automatización mediante software no presenta problemas difíciles de resolver, pero el tiempo requerido para obtener los resultados es demasiado grande. Por ello una solución hardware del algoritmo aprovecha el paralelismo que presenta el problema, y construyendo hardware específico para su resolución, obtendremos los resultados de forma más eficiente.

La plataforma de trabajo sobre la que se realizará el proyecto será un PC estándar al que se le conectará una placa de prototipado hardware basada en FPGAs.

V. Abstract

The main aim of this project is to study and design reconfigurable hardware of a complete system for the analysis of cytological data, which is based on a comparison of certain cellular parameters for a large number of cells. Through these groups it can be detected clusters of cells with similar parameters, which let the specialist detect anomalies in the patient.

This process has traditionally been manual although computers have helped to make it easier. Despite the fact that software automation does not lead to problems too difficult to resolve, it takes too much time to obtain the results. Due to this, a hardware solution of the algorithm, which can take advantage of parallelism, and a specific hardware design to resolve the problem, will allow us to get the results much more efficiently.

The working platform used to make this project will be a standard PC that will be connected to a hardware prototyping board based on FPGAs.

VI. Autorización

Los alumnos Iván Aldea López y María Bertrán de Lis Martín-Artajo, autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos no comerciales, y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo. Fdo.

1. Descripción del problema.

Base biológica

En la medicina actual el análisis de células juega un papel fundamental a la hora de diagnosticar si un paciente padece alguna enfermedad y determinar ésta. También estos análisis son utilizados en el campo de la investigación para estudiar la relación entre las características celulares y las enfermedades diagnosticadas.

Para ello se analizan y comparan los distintos parámetros de la muestra de sangre extraída del paciente. En torno a los resultados obtenidos tras la comparación, podemos agrupar las células formando clusters. Los miembros de estos grupos diferirán escasamente en los valores de los parámetros, siendo importante para detectar anomalías en el paciente, el número de células que forman cada cluster así como la media de sus parámetros característicos.

El estudio de los parámetros es relevante para la citometría de flujo. Esta técnica está destinada a la cuantificación de componentes o características estructurales de las células, fundamentalmente mediante métodos ópticos. A pesar de que las mediciones son realizadas sobre células individuales y por unidad de tiempo, permite procesar miles de células en pocos segundos. La citometría de flujo puede utilizarse para el análisis de los diferentes tipos celulares en una mezcla o suspensión, en base a la particularidad de que los diferentes tipos celulares pueden distinguirse mediante cuantificación de las características estructurales.

El principal uso diagnóstico se ha centrado en el estudio del ciclo celular y de la ploidía en cáncer, en el estudio de linfomas y leucemias, analizando la expresión de marcadores de superficie importantes para el diagnóstico, así como su valor pronóstico asociado. Aunque actualmente existen alternativas menos costosas a la citometría de flujo, sigue siendo el método empleado en el recuento de los niveles de células CD4 y CD8 en la sangre de pacientes con Síndrome de Inmuno Deficiencia Adquirida (SIDA).

Los parámetros analizados son:

- FSC-H: utilizado para medir el tamaño celular.
- SSC-H: utilizado para medir la complejidad estructural de las células.

- FL1-H, FL2-H, FL3-H, FL4-H2: detectores Calibur relativos a la proliferación celular, al ciclo celular en células vivas y células muertas.

1.2. Problema de la comparación de parámetros

Los algoritmos de comparación de parámetros son un tema de continuo estudio en el ámbito informático debido a la complejidad tan alta que representan.

En todos los programas informáticos se tiene muy en cuenta la eficiencia de ejecución y se tienden a optimizar lo máximo posible los algoritmos. Sin embargo los algoritmos de comparación en este problema son un obstáculo para llevar a cabo esta tarea, por depender de muchos factores a la vez.

A diferencia de los algoritmos de ordenación (como Quicksort, Mergesort…) que han demostrado reducir considerablemente el tiempo de ejecución al ordenar listas de datos, los algoritmos de comparación dependientes de más de un parámetro, como es nuestro caso, no se pueden beneficiar de tales mejoras.

Las células de nuestro estudio no se pueden ordenar mediante ningún criterio a priori que favorezca disminuir la complejidad de los algoritmos en tiempo de ejecución.

Para comparar todas las células desde el punto de vista de la programación imperativa, necesitamos al menos dos bucles para comparar cada una de ellas con el resto, por lo que estamos hablando de una complejidad cuadrática ($O(n^2)$).

Cuando hablamos de muestras reales, es decir, de millones de células frente a las mil o dos mil que empleamos nosotros como ejemplo, el tiempo de ejecución empleado es totalmente inaceptable. Por ello, es necesario un enfoque desde el punto de vista hardware donde se puede aprovechar el paralelismo reduciendo el tiempo de ejecución considerablemente.

A diferencia de la solución software obtenida con los lenguajes de programación imperativos, que no permiten realizar más de una comparación a la vez, la implementación hardware (y en nuestro caso concreto las FPGAs) consigue hacer varias comparaciones entre distintas células en un sólo ciclo. Por lo tanto, la velocidad con la que podemos

beneficiarnos del paralelismo, depende directamente de la velocidad de transferencia de datos de la memoria SDRAM de la FPGA a los registros de nuestro sistema.

2. Programación

A continuación explicaremos las soluciones software propuestas para crear de la manera más eficaz posible los clusters a partir de un fichero de células dado. Se analizarán los programas realizados en los lenguajes imperativos Perl y C, mostrando más tarde las ventajas e inconvenientes de cada uno de ellos.

2.1. Aproximación con el Lenguaje Perl

Comenzaremos señalando que el programa en Perl está formado por tres archivos .pl. En cada uno de ellos se realiza una parte del problema dividiéndolo en la comparación de células, reunión en clusters y medias de los parámetros celulares de los clusters.

ParesCélulas.pl

El primer archivo a estudiar es el ParesCelulas.pl. En este programa se lee un fichero de texto llamado datos.txt. En la primera línea de éste se encuentra el nombre de los seis parámetros de cada una de las células que vamos a comparar, a continuación en las líneas sucesivas encontraremos los valores de éstas mismas células. También tenemos como dato de entrada un valor entero llamado cutoff encargado de determinar si el resultado de la comparación de parámetros es afirmativo o negativo. Por ejemplo si el valor de cutoff es 40 la diferencia en valor absoluto de todos los parámetros de dos células debe de ser inferior a él para poder unirlas.

El programa va leyendo una por una las líneas del archivo datos.txt y las va comparando con cada una de las siguientes, por lo que constará de dos bucles anidados (dos whiles). Uno irá de i=1 al numero de células y el otro interior a éste de i+1 al número de células también. Hay que tener en cuenta que dentro del while más interno tenemos un for desde i=1 hasta 6 para comparar cada uno de los parámetros y determinar si se deben agrupar ese par de células o no. Para finalizar si la comparación sale positiva, se introduce en otro fichero de texto llamado pares.txt, que en este caso será abierto para escritura, el par de células que cumplen las condiciones para juntarse en un cluster. Cabe destacar que nombramos a cada célula con el número que ocupa como posición en el archivo datos.txt.

Teniendo todo esto en cuenta el coste del algoritmo con los dos bucles anidados es cuadrático respecto al número de células, lo que implica un

elevado tiempo de ejecución si el fichero de células es muy grande. Considerando un fichero de únicamente 2000 células el tiempo de ejecución de todo el programa es de 29 segundos aproximadamente.

En resumen la función de este programa es leer un archivo de texto con unos parámetros de un conjunto de células y devolver en otro archivo de texto los pares de células que difieran en todos sus parámetros, en un valor menor a un parámetro introducido por teclado llamado cutoff.

Clusters.pl

El segundo archivo a estudiar es el Clusters.pl, en este programa se lee el fichero de texto pares.txt creado por el programa anterior. En las líneas de éste se encuentran los pares de células que se podían unir porque sus parámetros eran similares.

El programa va leyendo una por una las líneas del archivo pares.txt y las va colocando en los cluster correspondientes por lo que constará de un bucle for con cada par de células que se pueden unir. El funcionamiento del bucle es el siguiente; leemos un par de células, si la primera célula ya tiene un cluster asignado y el segundo no, a ésta segunda célula se le asigna el cluster de la primera célula, y viceversa si tiene el cluster asignado la segunda y la primera no. En el caso de que ninguna tenga asignado el cluster, se le asigna el siguiente cluster que esté libre a ambas, y por último si los dos tienen el cluster asignado previamente, todas las células correspondientes al primer cluster se asignan al segundo cluster, lo que conlleva un nuevo recorrido de todas las células (proceso de renombramiento).

Para acabar, cada célula con su cluster asignado en el transcurso del programa, se introduce en el fichero de salida llamado clusterRes.txt, siendo cada línea el número de célula con su cluster correspondiente. Cualquier célula que no cumpliera las condiciones exigidas para formar un cluster con ninguna del resto, no aparecerá en dicho archivo.

Teniendo todo esto en cuenta el coste del algoritmo vuelve a ser cuadrático, ya que el proceso de renombrar los clusters en caso de que ambas células tuvieran uno asignado previamente implica recorrer todas las células del archivo de nuevo y esta situación es muy frecuente.

En nuestro caso, el tiempo de ejecución de este segundo archivo Clusters.pl para las 2000 células es de 22 segundos aproximadamente.

En resumen la función de este archivo es leer un archivo de texto con los pares de células que se podían comparar, asignarle clusters a cada una de esas células mostrando en el archivo de salida clusterRes.txt las células y su cluster correspondiente.

MediasClusters.pl

El tercer archivo a estudiar es el MediasClusters.pl, en este programa se reciben dos ficheros de texto datos.txt y clusterRes.txt, recibidos de los archivos anteriores, el primero de ellos contiene todos los parámetros de todas las células y el segundo está formado por pares de números, el primero representa una célula y el segundo su cluster asignado.

El programa va leyendo del fichero de texto clusteRes.txt cada par célula-cluster, por lo que constará de un primer bucle for de tamaño igual al número de líneas clústeRes.txt. En este bucle se lee cada par de célula-cluster y los parámetros de la célula obtenidos del fichero datos.txt se van acumulando en el cluster que le corresponde.

A continuación tendremos otro bucle for de tamaño igual al número de cluster. Para cada uno de los cluster formados por células vamos calculando la media aritmética de los parámetros de las células que forman dicho cluster. Para terminar, cada cluster con las medias de los parámetros se introduce en otro archivo llamado medias.txt que será el resultado final del análisis.

Teniendo todo esto en cuenta el coste del algoritmo con sus bucles for es lineal. Para el ejemplo de las 2000 células, sólo se tarda 2 segundos en calcular las medias de los clusters.

En resumen la función de este archivo es leer el archivo de entrada clusteRes.txt, ir acumulando en cada cluster la suma de los parámetros de cada una de las células que lo componen, realizar la media de esos parámetros para cada uno de los cluster y almacenarlos en el archivo de salida medias.txt.

2.2. Aproximación con el Lenguaje C

Veamos ahora la implementación en el lenguaje imperativo C. En este caso el programa consta de un único archivo llamado Proyecto.c en el cual se comparan todas las células, se agrupan en clusters y para cada uno de ellos se calcula la media de los parámetros de sus células.

El objetivo de solucionar el problema en este lenguaje es mejorar la eficiencia y optimizar al máximo los algoritmos de comparación para obtener los mejores tiempos de ejecución posibles.

Hemos utilizado una estructura para representar cada célula, que consta de un array para guardar los valores de cada parámetro, un identificador para saber de qué número de célula se trata y el número de cluster que se le asignará.

También se ha construido una estructura para representar cada cluster teniendo como campos, un contador de las células que lo forman y un array donde se llevará la suma de los parámetros de dichas células.

Era necesario construir un tipo de datos que nos permitiera acceder lo más pronto posible a los datos siendo fácil de mantener. Es decir, proporcionando las operaciones típicas de inserción, búsqueda, eliminación y mostrar de la forma más simple posible. Por ello decidimos construir listas doblemente enlazadas (Véase figura 1).

Cada nodo de estas listas apuntará al nodo anterior y al nodo posterior, excepto el primer nodo y el último que tendrán el puntero al anterior y al posterior a Null respectivamente. Además la lista de los clusters tendrá un campo más para identificar el número del cluster en ese nodo. Estas listas no se mantendrán ordenadas bajo ningún criterio durante la ejecución del programa, pues como ya se ha explicado anteriormente, el orden de las células cuando su comparación depende de varios parámetros, no facilita la optimización del algoritmo. Sin embargo, la lista de clusters identificará a cada uno de ellos según su orden en esta lista.

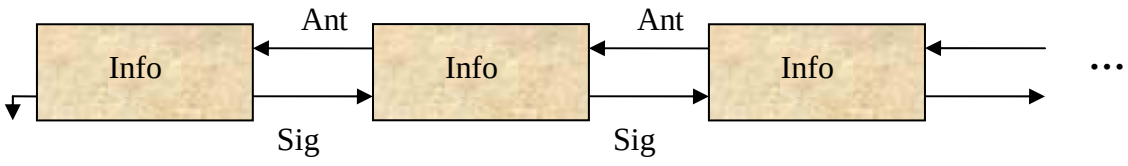

Figura 1: Lista doblemente enlazada

Para insertar un nodo en una lista emplearemos una función que recibe como parámetros la nueva célula o cluster y un puntero al nodo actual de la lista en la cual queremos añadirlo. Creamos el nuevo nodo que se insertará justo detrás del nodo actual pasando a tomar éste ese papel y actualizando

todos los enlaces debidamente (recordemos que es necesario distinguir si el nodo es la cabecera de la lista o por el contrario es el último, pues estos son casos especiales). Gracias a que la lista no está ordenada, el coste de esta función es mínimo.

De la misma manera, la función eliminar recibe por parámetro un puntero al nodo actual de la lista de la cual vamos a borrar. El nodo que se eliminará será dicho nodo, y se procederá a restablecer los enlaces de su nodo anterior y posterior correctamente. El nodo posterior al borrado pasará a ser el nodo actual de la lista.

Para las listas de clusters fue necesario implementar la función de búsqueda de un determinado cluster. Ésta recibe como parámetros el identificador de cluster y el puntero al nodo actual de la lista, recorre la lista para delante o para detrás según el identificador del nodo actual y deja el puntero a dicho nodo apuntando al nodo buscado.

Se realizaron también las funciones para mostrar las listas para la depuración del programa aunque no son estrictamente necesarias para realizar el análisis. Éstas sólo recorren la lista doblemente enlazada y muestran los datos por pantalla.

Además existen otras funciones para que el programa quedase organizado y fuese fácil de comprender y mantener. Una de ellas es la función que compara dos células pasadas por parámetro según el cutoff pasado también como tal. Si ambas células se deben juntar en un cluster devuelve 1, en caso contrario 0.

La función llamada cambiarCluster recibe la lista de células y dos identificadores de cluster, el nuevo y el antiguo. Se va recorriendo toda la lista de células y se cambia el cluster de aquellas células que tuvieran asignado el identificador antiguo por el nuevo. Este proceso de renombramiento es igual al realizado en Perl y por desgracia no puede optimizarse o evitarse en ninguno de los casos.

En primer lugar, nuestro programa abre el archivo datos.txt donde se encuentran las células y las guarda en la lista doblemente enlazada en el mismo orden en el que aparecen en dicho archivo. Una vez terminamos de leer los datos procedemos a la comparación de células.

Cada célula de la lista se compara con el resto de células mediante dos bucles. El bucle externo recorre toda la lista de células una vez y para cada una de ellas, el bucle interno recorre la lista desde la célula actual del bucle

externo a la última célula de la lista. Por tanto estamos hablando de una complejidad cuadrática.

Es en este punto, cuando se llama a la función CompararCélulas(), que determina si los 6 parámetros de las dos células pasadas como parámetros se diferencian en un valor menor que el cutoff. Si el resultado es positivo, asignamos el nuevo cluster a la célula con la que comparamos. Si ésta ya tenía un cluster previamente asignado, se llama al método cambiarCluster() que recorrerá toda la lista de células cambiando el cluster antiguo por el nuevo.

Este proceso de renombramiento tiene un coste lineal, es decir, O(n) siendo n el número de células de la muestra. Por esto, el algoritmo de comparación tiene un coste muy alto, porque además de los dos bucles anteriormente explicados en muchos casos tendremos que renombrar clusters recorriendo la lista nuevamente.

Para comprender mejor esta parte del programa, veamos un ejemplo. Supongamos que las parejas de células cuya comparación saldrá positiva son:

Célula	Célula similar
1	3
1	7
2	5
2	7
4	8

Figura 2: Tabla ejemplo de pares de células similares

Al ir recorriendo la lista de células en el bucle exterior, asignaremos el cluster número 1 a la célula 1. Como es similar a la 3 y la 7 y éstas células no tienen ningún cluster asignado previamente, se les asignará el cluster 1 también. Una vez acaba el bucle interno, pasamos a comparar la célula número 2 con el resto. Como no tiene ningún cluster asignado se le pone el siguiente libre, es decir, el cluster numero 2. Vamos comparándola con las células siguientes a ella hasta llegar a la célula 5, que como tampoco tiene cluster asignado y es similar, toma el mismo valor de cluster.

El problema viene al comparara la célula 2 con la célula 7. Éstas deben pertenecer al mismo cluster, pero la primera tiene asignado el cluster 2 mientras la segunda tenía el cluster 1. Es aquí donde interviene el proceso

de renombramiento. Se debe volver a recorrer la lista de células entera (pues puede haber células con clusters asignado en posiciones posteriores a las células comparadas), y cambiar todas aquellas que tengan asignado el cluster 1 por el cluster 2. Después de este proceso, las asignaciones de clusters para cada célula quedarán del siguiente modo:

Número de célula	Número de cluster asignado
1	2
2	2
3	2
4	3
5	2
6	---
7	2
8	3

Figura 3: Tabla ejemplo de células con sus clusters correspondientes

Nótese que la célula 6 no es similar a ninguna otra, por lo tanto, no será utilizada para formar los clusters y los valores de sus parámetros serán ignorados el resto del programa. También se puede observar cómo el cluster numero 1 deja de existir. Esto ocurrirá frecuentemente con varios clusters después del renombramiento, pero al crear la lista de clusters dinámica no perderemos espacio y habrá nodos (como el cluster 1) que no estarán en la lista.

Después de comparar todas las células, formamos la lista de clusters. Para ello, se recorre una vez la lista de células y para cada célula se busca el cluster correspondiente en la lista de clusters, se actualiza su número de células y se suman los parámetros de la célula para luego poder hacer la media aritmética. Debido a que no sabemos a priori cuántos clusters habrá, no podemos usar almacenamiento estático porque desaprovecharíamos mucha memoria o podríamos necesitar más. Esto hace que la lista de clusters tenga que recorrerse varias veces, por lo que el coste en el peor caso de formar esta lista será O(n*m), siendo n el número de células y m el número de clusters.

Una vez la lista de clusters está finalizada, se llamará a la función recorrerYcrearMedias para realizar la media aritmética de los parámetros de cada cluster y escribir los resultados en el fichero llamado medias.txt. Después se procede a liberar toda la memoria utilizada durante la ejecución del programa.

La optimización llevada a cabo respecto al programa en Perl hace que el tiempo de ejecución se reduzca mucho. Para nuestro ejemplo de 2000 células, el programa tarda menos de un segundo en ejecutarse, aunque en una muestra real este tiempo aumentaría exponencialmente. Es decir, por tratarse de un coste cuadrático, si el número de células se duplica, el tiempo de ejecución aumenta cuatro veces. Por ello es interesante plantear este problema desde el punto de vista de la programación en paralelo que se llevará a cabo mediante implementación Hardware.

2.3. Comparación entre ambos programas

Tanto el programa escrito en Perl como el programa en el lenguaje C obtienen los mismos resultados como era de esperar. Para su correcta comprobación, se realizó el programa en C llamado Comprobación.c que comprueba que los dos archivos resultantes al calcular las medias de cada cluster coincidan exactamente, tanto en número de clusters como en los valores que los determinan.

Para ello, el programa abre los dos archivos con los resultados y lee cada línea creando una lista de clusters para cada archivo. Una vez leídos ambos archivos, se comprueba que el número de clusters coincide.

Para ello, se ejecuta un bucle que recorre cada cluster resultante del programa Perl y busca su igual en la lista de clusters del programa en C. Para comprobar que todos y cada uno de los clusters de una lista coincide con otro de la otra lista, el sistema va mostrando un mensaje de éxito. En caso de encontrar alguna diferencia el sistema mostrará un mensaje de error indicando el número de clúster que no coincide con ningún otro.

Es obvio que con la implementación en C se obtienen los resultados esperados mucho antes que con Perl. Esto es debido a la gran optimización llevada a cabo en el primero junto con las grandes ventajas que nos ofrece este lenguaje para mejorar el rendimiento. Así por ejemplo, el uso de estructuras, listas doblemente enlazadas y punteros han facilitado enormemente este proceso.

3. Implementación Hardware

3.1. FPGAs. Descripción general

Una vez comprendido el problema que se desea solucionar, ante la necesidad de desarrollar un sistema que sea capaz de aportar una mejora significativa en el tiempo de cálculo de unos algoritmos que por su elevado tiempo de ejecución apenas se emplean en el campo de la biología, la opción más adecuada parece ser la implementación hardware de un sistema específico que ejecute dichos algoritmos.

La tecnología más potente disponible en el ámbito universitario para la construcción de circuitos propios es el hardware reconfigurable. Las FPGAs (Field-Programmable Gate Array) pueden considerarse como una evolución conceptual de los antiguos PLDs (Programmable Logic Devices). La naturaleza de esta evolución estriba en el hecho de que, mientras los PLDs son circuitos con una construcción y una topología fijas en los que el usuario se limita a programar las conexiones internas, en las FPGAs se otorga la posibilidad de diseñar (o configurar, de ahí el concepto de hardware reconfigurable) el circuito en sí. Se trata, además, de chips muy versátiles ya que, por un lado es posible cambiar el circuito tantas veces como se quiera, y por otro lado están diseñados de modo que son capaces de conectarse a una gran variedad de sistemas externos, como por ejemplo un sistema de memoria RAM, indispensable para la realización de este proyecto.

Las FPGAs están revolucionando la manera en que los diseñadores de sistemas implementan lógica digital. Éstas reducen radicalmente los costos y el tiempo de desarrollo para implementar diseños de miles de compuertas lógicas.

El área de aplicación principal de las FPGAs (aparte del académico), debido al bajo coste de adquisición a pequeña escala, junto a la posibilidad de reconfiguración, es el desarrollo de prototipos de circuitos específicos de los que se realizan hard-copies denominadas ASIC (Application-Specific Integrated Circuit), con menor coste de fabricación a gran escala y menor consumo, aunque no son modificables y pueden requerir varias semanas para ser implementados. No obstante, existen otras áreas de aplicación que emplean hardware reconfigurable como producto final, ya que aprovechan la posibilidad de reconfiguración dinámica de los circuitos. Esta

metodología permite cambiar en tiempo de ejecución la totalidad o incluso una parte del circuito, con lo que se podrían concentrar las tareas de varios procesadores que conformasen un sistema en un solo chip, con la evidente reducción de coste de producción. Ciertos sectores industriales, en los que se realizan tareas muy específicas y que están sujetos a frecuentes cambios de estándares son el ejemplo más claro de este tipo de desarrolladores.

El sistema que compara parámetros de células también es susceptible del aprovechamiento de esta técnica, ya que para realizar la similitud de parámetros se realizan varias comparaciones pudiendo realizar varias de éstas a la vez.

La arquitectura interna de una FPGA está compuesta principalmente por tres tipos de unidades, que se encuentran uniformemente distribuidas por el área de integración del chip: en primer lugar, existe una serie de unidades lógicas, o CLBs (Configurable Logic Block), encargadas de procesar las señales lógicas. Estas unidades están interconectadas a través de cables que son conmutados por el segundo tipo de unidades internas, los bloques de conmutación, programables al igual que los CLBs. Por último, el chip está provisto de una serie de unidades de E/S**,** encargadas de la comunicación con el exterior del circuito, conectadas igualmente a los bloques de conmutación.

A pesar de que todas las FPGAs encajan en este esquema arquitectónico, cada fabricante sigue un estilo propio en el diseño de las unidades funcionales, sobre todo de los CLBs. La estructura básica de esta unidad es la siguiente:

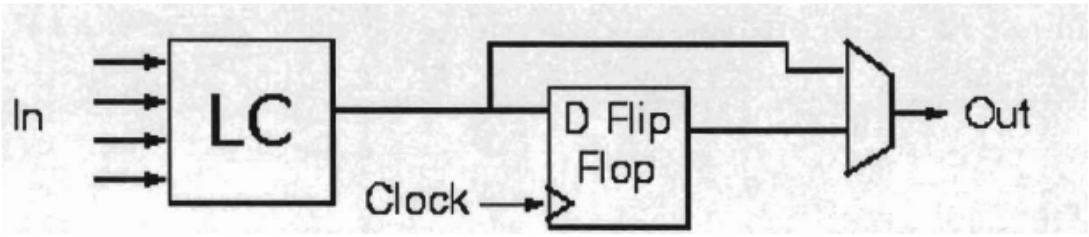

Figura 4: Estructura básica del CLB

El bloque de la izquierda constituye la lógica combinacional del circuito. Tiene un número determinado de entradas y es capaz de implementar cualquier función lógica de las mismas. La salida de este bloque se conecta a un biestable disparado por flanco (suele haber cierto número de CLBs que contiene latches disparados por nivel) que

conformará la lógica secuencial del circuito. Finalmente, el multiplexor de la derecha se configura de modo que el CLB implemente lógica secuencial (seleccionando la salida del biestable) o combinacional (seleccionando la otra entrada).

Xilinx es el mayor proveedor de FPGAs mundial [2]**.** El diseño de sus CLBs (ver Figura 5) está basado en tecnología SRAM**,** de modo que las configuraciones no se conservan al cortar el suministro de energía, aunque para solucionar esto las placas suelen estar dotadas de una memoria Flash adjunta que almacene los diseños. El bloque combinacional está implementado en forma de tabla de look-up (LUT) mediante celdas locales distribuidas de SRAM. Esto permite un proceso de configuración rápido y admite un amplio rango de funciones sin necesidad de una red de puertas lógicas de varios niveles. En contrapartida, este enfoque penaliza circuitos con un número elevado de entradas, ya que el tamaño de la LUT aumenta exponencialmente.

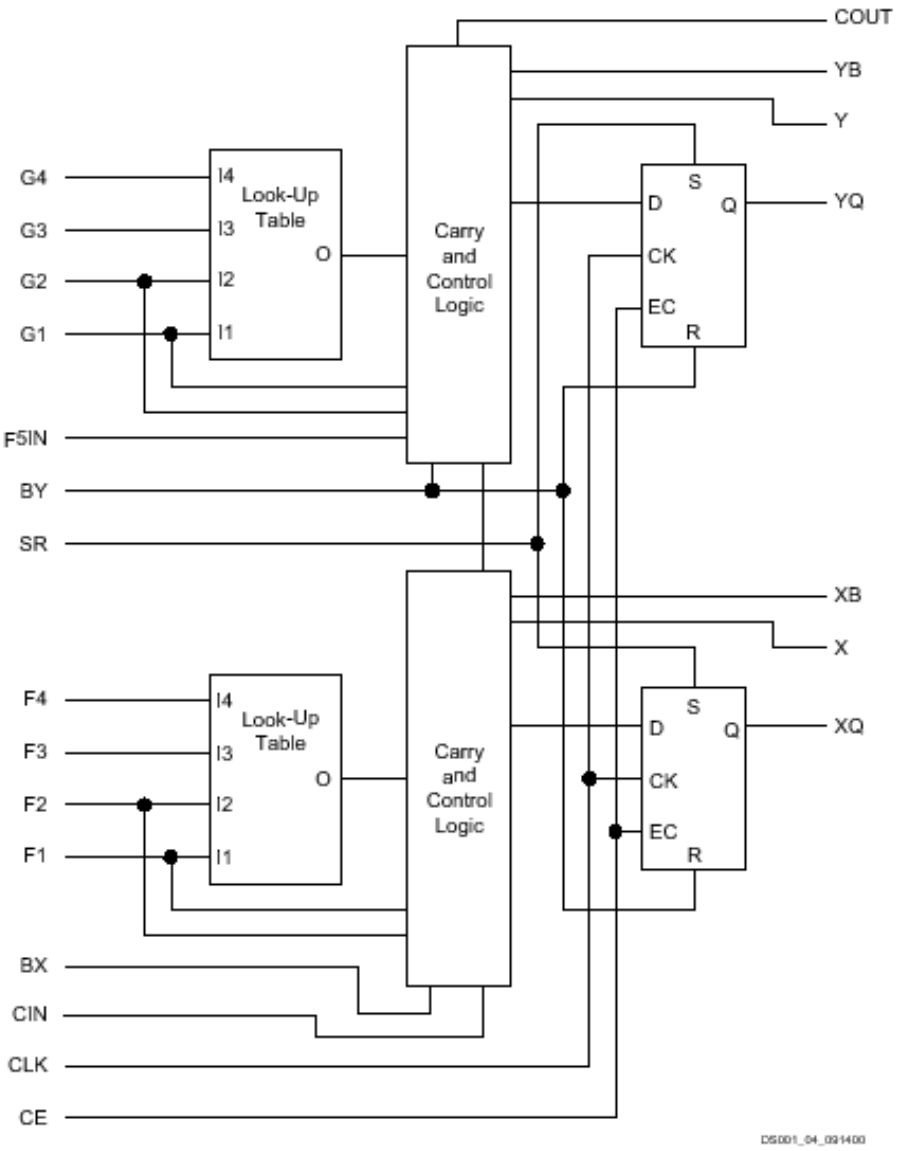

Figura 5: CLB de Xilinx con 2 LUT (Spartan II)

La segunda alternativa es Altera [3]. El diseño de sus CLBs está basado en memorias EPROM y EEPROM, por lo que las configuraciones no son volátiles, a diferencia de las de Xilinx. Este diseño (o al menos los iniciales) está mucho más próximo a la idea de los antiguos PLD, ya que el bloque combinacional principal del CLB está compuesto por dos niveles de puertas AND-OR con interconexiones programables. En contraposición al enfoque de Xilinx, los CLBs de Altera penalizan circuitos con pocas entradas, ya que desaprovechan mucha área de silicio, pero funcionan mejor para muchas entradas. No obstante, el mayor problema que presentan es la existencia de un consumo estático inexistente en los CLBs de Xilinx.

Como se ha dicho arriba, las interconexiones entre distintos CLBs del chip han de ser programables por el usuario. Las FPGAs de Xilinx

disponen de una serie de GRMs (General Routing Matrix) programables (ver Figura 6), adyacentes a cada CLB, que se encargan de realizar una función de conmutación convencional para confeccionar el enrutamiento del circuito.

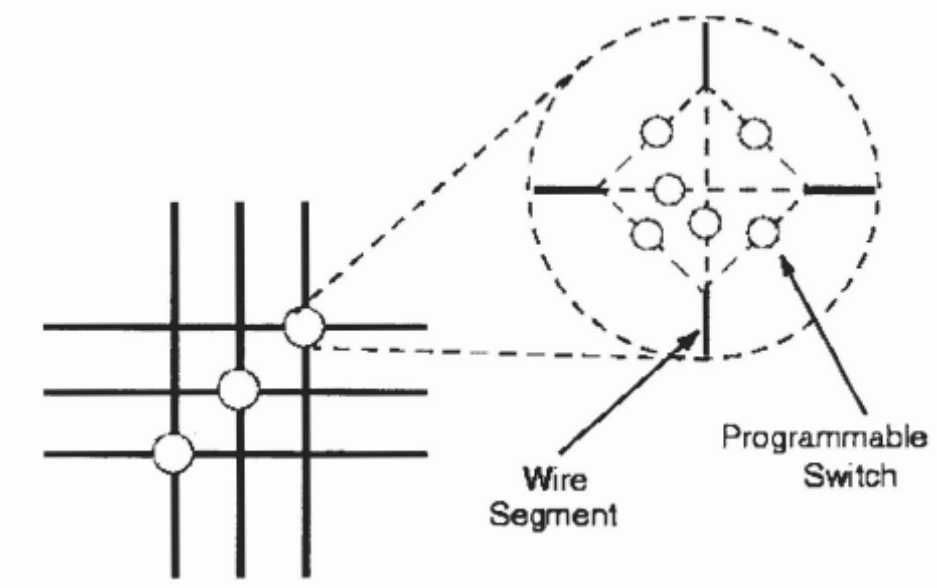

Figura 6: Estructura básica de un GRM

La función de enrutamiento de Altera, en cambio, no emplea unidades específicas, sino que se sirve de una técnica más compleja denominada Fast-tracking, que se puede observar en la Figura 7.

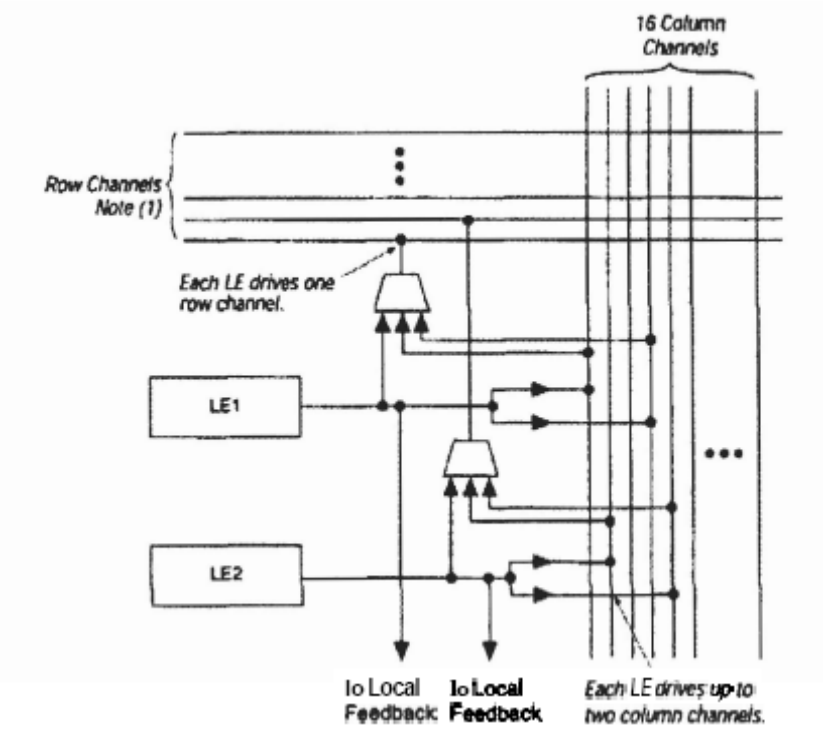

Figura 7: Dos unidades lógicas de ALTERA comunicadas por Fast-Tracking

El tercer elemento principal en la arquitectura de las FPGAs lo constituyen los bloques de entrada/salida. Como ya se ha dicho, la versatilidad de las FPGAs reside en gran medida en la gran variedad de elementos a los que pueden estar conectadas. Esto impone varias restricciones que deben ser tenidas en cuenta. En primer lugar, distintos estándares conllevan distintos voltajes de referencia, que en muchos casos es necesario que aporte el propio chip de la FPGA. Además, la comunicación con sistemas externos siempre hace necesario algún sistema de sincronización. Este tipo de problemas son los que se encargan de solucionar los bloques de entrada/salida, que disponen de mecanismos analógicos (diodos, etc.) para permitir trabajar con los distintos voltajes propios de los distintos estándares existentes, por ejemplo los 3.3V de TTL, AGP o PCI, o los 2.5V de CMOS. En este caso, la FPGA se comunica con una SDRAM que funciona a 3.3V. En cuanto a la sincronización con sistemas externos, los puertos de entrada/salida disponen de líneas de retardo programables que ajustan posibles errores de sincronismo. Asimismo, cada bloque de E/S dispone de biestables que pueden funcionar como flip-flops disparados por flanco, para registrar las salidas y entradas, o bien como latches disparados por nivel.

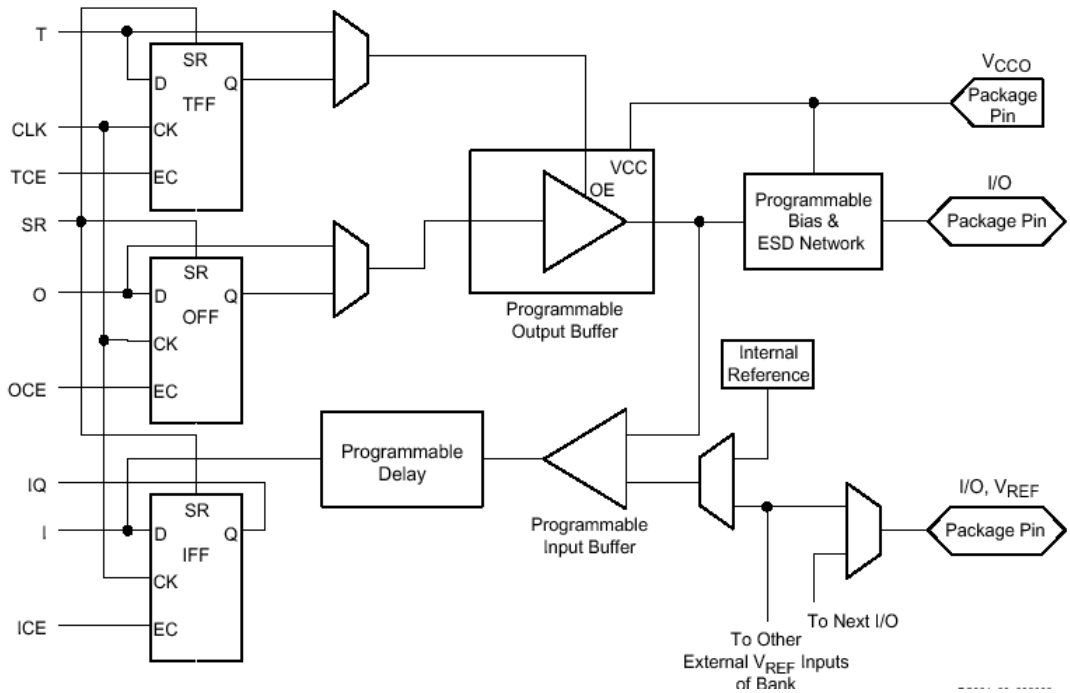

Figura 8: Bloque de entrada/salida (Spartan II)

3.2. FPGAs. Descripción específica

La FPGA empleada para realizar este proyecto (Xilinx XC2S100) es la más comúnmente utilizada en nuestra universidad en la fecha de su realización por los alumnos. Pertenece a la familia Spartan II, contiene alrededor de 100 000 puertas lógicas y la tecnología de fabricación es de 0.1 8 micras. Dispone de 600 CLBs, cada uno con cuatro celdas lógicas (ver Figura 9). Dichas celdas están provistas de una LUT de 4 entradas que está implementada con tecnología SRAM, lo cual permite la posibilidad de ser empleada como memoria RAM distribuida, en lugar de almacenar las funciones combinacionales necesarias.

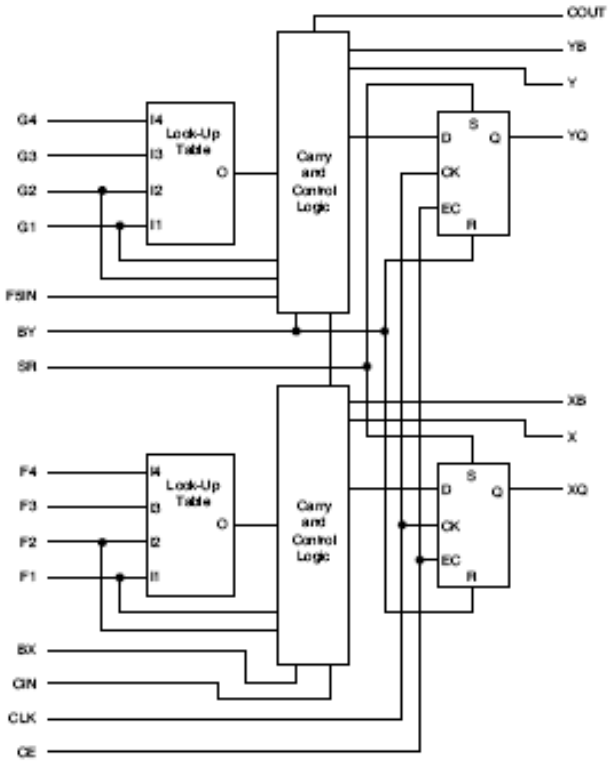

Figura 9: CLB de la familia Spartan II

Por otro lado, los CLBs están provistos de lógica de acarreo, lo que se traduce en una optimización de la comunicación entre unidades lógicas adyacentes para realizar funciones aritméticas con mayor rapidez. Concretamente, cada unidad lógica dispone de una puerta XOR para confeccionar sumadores con acarreo. Esta característica es de gran importancia en el sistema, ya que para calcular las direcciones de memoria se realizan varias sumas en cascada de vectores de bits de longitud bastante elevada (unos 14 bits), lo cual influye decisivamente en el rendimiento global, ya que forma parte del camino critico del circuito.

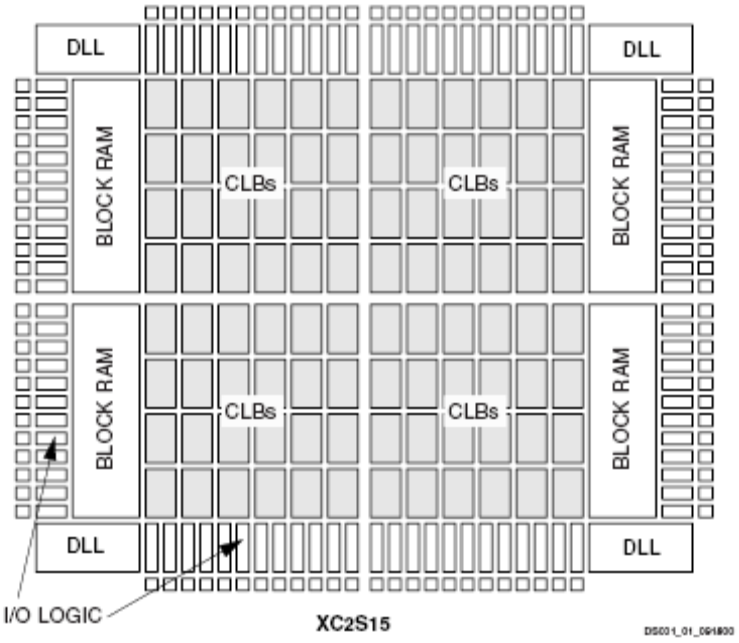

Figura 10: Diagrama de bloques de una FPGA de la familia Spartan II

Otro elemento de gran interés de este modelo de FPGA lo constituyen los bloques de RAM internos del chip. Se trata de 10 bloques de SRAM totalmente independientes que suman 5KB en total. Al estar dentro del chip, eliminan la necesidad de incluir sincronizadores, ya que las tareas de sincronización local se realizan automáticamente durante el proceso de implementación. Además, la independencia total de los bloques (tanto en relación a líneas de dirección y datos como a señal de reloj) permitiría realizar lecturas y escrituras paralelas en distintos bloques, lo que se ajusta de una manera natural al funcionamiento del algoritmo de comparación de parámetros de células.

Por ello, se ha optado por emplear memoria RAM externa, aunque la memoria interna podría ser aprovechada en posibles ampliaciones de forma complementaria.

Una vez decidido trabajar con memoria RAM externa, como se ha dicho antes, se ha de considerar la necesidad de sincronizar la ruta de datos y la memoria, sobre todo cuando se trata de memoria síncrona, como es el caso. Para ello, la FPGA dispone de cuatro DLLs (Delay-Locke Loop). Estos módulos eliminan el posible skew entre el reloj local y otra señal externa. Esencialmente, se trata de un sincronizador en lazo cerrado capaz de gestionar dos dominios de reloj distintos. Esta unidad será empleada en el interfaz de memoria para ajustar el reloj local y el reloj de la SDRAM.

La FPGA empleada está montada en la placa XSA-100, de Xess (Figura 11). La placa dispone de un módulo de memoria SDRAM de 16 MBytes que funciona a un máximo de 133MHz, aunque a pesar de que las FPGAs de la familia Spartan II son capaces de funcionar a una frecuencia de reloj de 200MHz, el oscilador de la placa funciona a 100MHz como máximo.

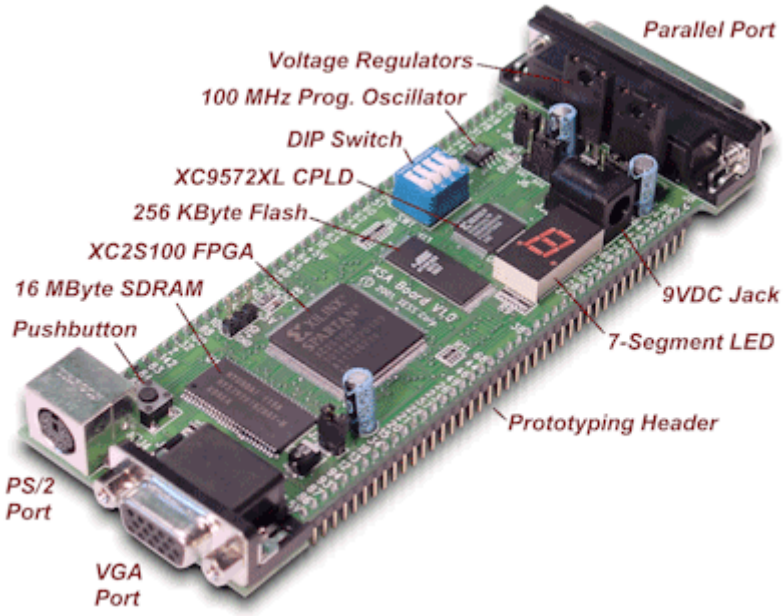

Figura 11: Placa XSA 100

Internamente, el módulo de memoria está dividido en 4 bloques, aunque debido al modo de montaje, la FPGA lo gestiona como un único banco de 8M palabras de 16 bits. Al tratarse de una SDRAM, necesita refrescarse cada cierto tiempo, y permite operaciones de lectura y escritura en ráfagas de 1,2,4 u 8 operaciones.

Hay que destacar que existe una utilidad de libre distribución creada por Xess [1] que permite almacenar valores en memoria a través del cable paralelo conectado al PC antes de configurar el circuito en la FPGA a partir de un archivo en formato hexadecimal, así como realizar el proceso inverso, es decir, descargar el contenido de la memoria al PC. Éste es el medio empleado tanto para cargar los parámetros de células en memoria como para obtener los resultados generados tras el cálculo. La aplicación se llama GXSLOAD y está integrada en el paquete XSTOOLS, disponible en la página de Xess [1].

3.3. Sistema combinacional para la comparación de células

A continuación pasamos a detallar el funcionamiento de todo el circuito combinacional en vhdl para comparar dos células, mostrando el archivo donde se encuentra implementada cada función explicada.

Todo este sistema está estructurado en distintos módulos para facilitar su comprensión y aislar el funcionamiento de cada componente. El módulo que engloba el resto de componentes será ComparadorCelula.vhd.

Nuestro circuito consta de dos bancos de registros (Registro6.vhd), uno para cada célula donde se guardarán los seis parámetros correspondientes. Cada uno de ellos está formado por seis registros de 10 bits (Registro.vhd) para poder almacenar cada parámetro celular. Esto se puede observar en la figura 12 que muestra un comparador de dos células completo (ComparadorCelula.vhd).

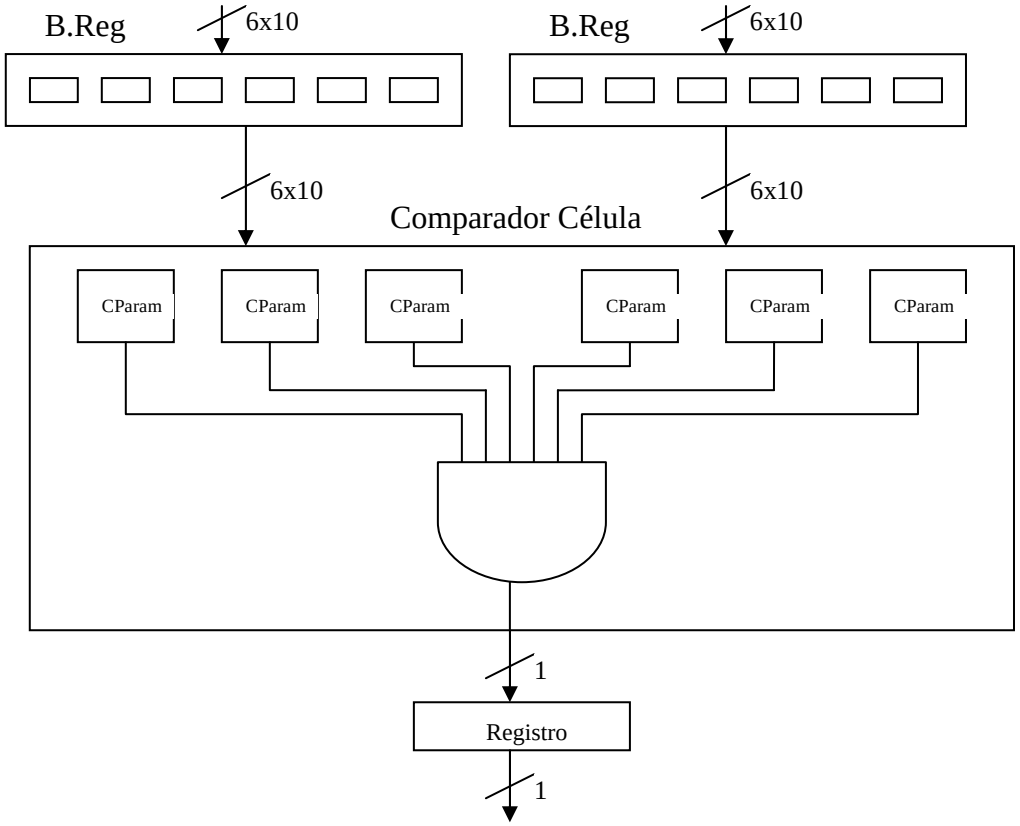

Figura 12: Comparador de dos células

El resultado de la comparación de ambas células será el bit obtenido al realizar la and de las salidas de los seis comparadores de parámetros. Este valor será cargado en un registro de 1 bit obteniéndose el resultado de la comparación en el siguiente ciclo de reloj. Si dicha salida tiene el valor 1 lógico significará que los seis parámetros de las dos células no difieren en más del valor cutoff (en nuestro caso 40) para cada uno de ellos y se procederá a guardar la pareja de células para agruparlas en el mismo cluster. En caso contrario, la salida tendrá el valor 0 lógico y pasaremos a comparar otras dos células.

El comparador de células propiamente dicho consta a su vez de seis comparadores de dos parámetros cada uno (ComparadorParametro.vhd). El circuito de cada comparador de parámetro se puede ver en la figura 13. Éste posee dos entradas de 10 bits cada una y una salida de 1 bit que irá a la entrada de la puerta and anteriormente mencionada en el comparador de células. Una vez introducidas las señales de entrada, colocaremos la entrada de mayor valor en el parámetro a, siendo el minuendo de la resta, y el de menor valor en el parámetro b, siendo el sustraendo de la misma (colocaresta.vhd).

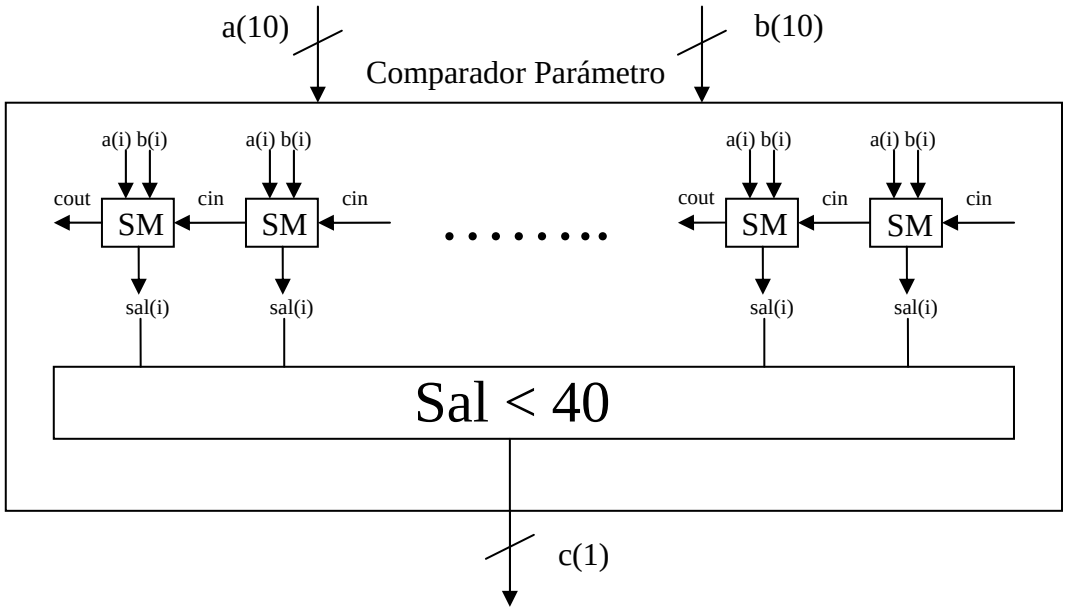

Figura 13: Comparador de dos parámetros

Como puede apreciarse en la figura, cada comparador está formado por diez semi-restadores iguales (semirestador.vhd). El primero que compara los bits 0 de los parámetros, tendrá como entrada cin=0 además de las entradas a(0) y b(0) correspondientes. Una vez restados estos valores, muestra la salida por la señal sal(0) y si hay acarreo (es decir, cuando cout=1) se pasa al siguiente semi-restador por su entrada cin poniendo esta señal a 1. El resultado de la resta se comparará con el valor de cutoff y si es menor, obtendremos en la salida el valor 1 lógico (compararlimite.vhd).

3.4. Memoria

Ahora pasaremos a detallar el manejo de la memoria SDRAM de la FPGA. Necesitamos encapsular el funcionamiento genérico de una SDRAM para adaptarlo a la placa XSA. El archivo asociado que desempeñará esta función será xsasdramctl.vhd.

El controlador de la memoria DRAM síncrona (SDRAM) de la placa utilizada acepta peticiones simples de lectura y escritura, generando las señales de onda necesarias para llevar a cabo estas operaciones en la SDRAM. Si el modo ráfaga está activado (parámetro PIPE_EN), estas operaciones realizadas sobre una misma fila podrán ser ejecutadas cada ciclo de reloj.

Además, el controlador se encargará de las operaciones de refresco de la memoria que son necesarias para mantener los datos correctamente en la SDRAM dado su carácter volátil. Éste pondrá a la memoria en modo auto-refresco, de forma que aunque el controlador deje de operar, los datos permanecerán almacenados.

La SDRAM está formada por 4096 filas x 512 columnas (NROWS x NCOLS) por cada banco de memoria. Cada columna de cada fila contiene una palabra de memoria, es decir, 16 bits que viene determinado por el parámetro general DATA_WIDTH.

El interfaz de la memoria se muestra en la figura 14. Las funciones de las principales señales de entrada/salida son:

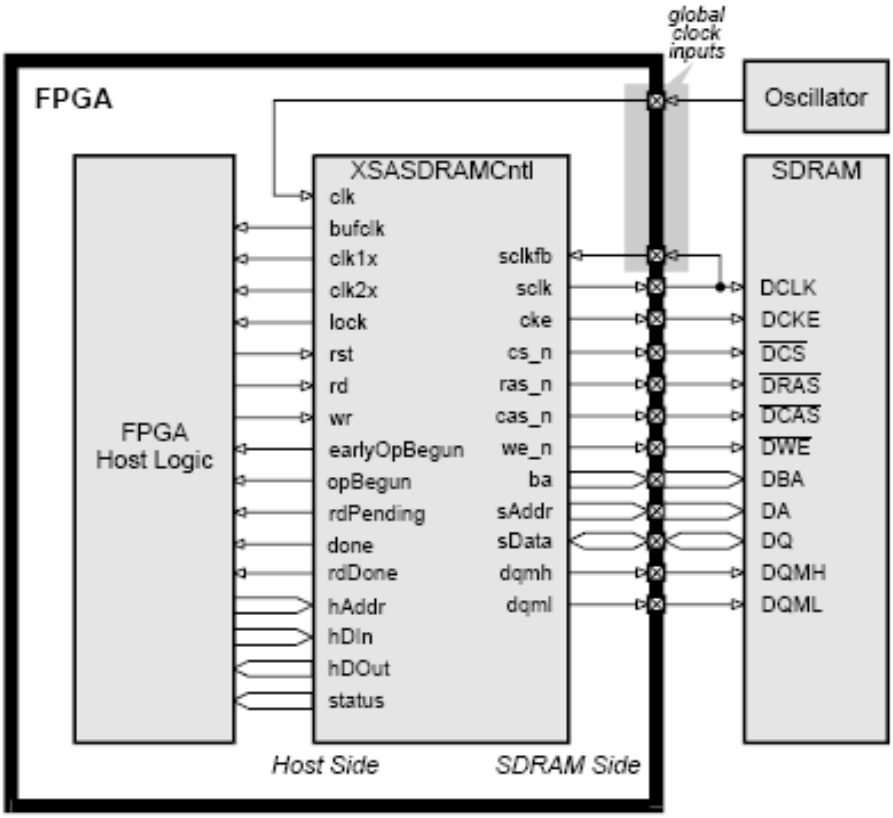

Figura 14: Interfaz del controlador de SDRAM de la placa XSA

- Clk: ésta es la entrada del reloj principal. El reloj del oscilador de la placa entra en la FPGA a través de esta entrada.
- Sclk: esta salida lleva la señal del reloj principal a la SDRAM externa.
- Rst: esta señal es asíncrona y se activa a alta. Su función es resetear la lógica interna del controlador SDRAM. También inicializa la memoria para poder ser utilizada.
- Rd: esta entrada activa a alta inicia una operación de lectura de una palabra de la memoria. Esta señal deberá mantenerse activa hasta que la señal opBegun indique que el proceso de lectura ha comenzado. Si no queremos que se realice otra operación de lectura después, esta señal se deberá desactivar antes de comenzar el siguiente ciclo.
- Wr: esta señal activa a alta inicia una operación de escritura de una palabra de memoria. Al igual que en el proceso de lectura, esta señal deberá mantenerse hasta que se active la señal opBegun y deberá desactivarse antes del siguiente ciclo si no se desea seguir escribiendo en memoria.

- opBegun: es una señal de salida síncrona que se activa a alta para indicar que se está realizando una operación de lectura o escritura en ese ciclo.
- earlyOpBegun: señal de salida asíncrona que se activa durante el ciclo de reloj que precede a la activación de la señal opBegun.
- rdPending: señal de salida síncrona para indicar que hay datos procedentes de una operación de lectura que todavía no han sido sacados por la salida hDOut.
- Done: señal síncrona de salida que se activa a alta indicando que la operación de lectura o escritura ha finalizado correctamente. Se mantiene a alta durante todo el ciclo de reloj.
- rdDone: señal de salida activa a alta que indica la finalización de una operación de lectura. Se mantiene a alta durante todo el ciclo de reloj.
- hDin: bus de entrada donde serán introducidos los datos que queramos escribir en la memoria SDRAM. Los valores de los datos deberán mantenerse estables hasta que se active la señal opBegun.
- hDout: los datos procedentes de las operaciones de lectura saldrán por este bus. Los datos aparecerán en la subida del ciclo de reloj después de activarse la señal done o rdDone.
- hAddr: esta señal de entrada es para introducir la dirección de la memoria donde queremos leer o escribir. Al igual que la señal hDin, necesita mantenerse estable hasta que opBegun se active. Los dos bits más significativos de esta dirección corresponderán al banco de memoria, los $\log_2$(NROWS) bits se referirán a la fila de ese banco a la que hay que acceder y los $\log_2$(NCOLS) bits menos significativos corresponderán a la dirección de la columna en esa misma fila. Para la placa XSA 100 las direcciones serán así:

Banco	**Fila**	**Columna**
00	RRRRRRRRRRR	CCCCCCCCC

- Cke: señal de salida para habilitar la señal de reloj de la SDRAM.
- We_n: señal que habilita la SDRAM para una operación de escritura.
- Ba: bus de dos bits para seleccionar uno de los cuatro bancos de memoria de la SDRAM.

- sAddr: señal de salida hacia la SDRAM que contiene la dirección de la fila y columna a la que queremos acceder.
- sData: bus de doble dirección que llevará los datos de tamaño una palabra de memoria (16 bits) que se escribirán en la SDRAM o que se leen desde ésta.

Veamos un diagrama con las principales señales durante el proceso de lectura en modo ráfaga (figura 15). En este ejemplo se asume que las tres operaciones de lectura son sobre el mismo banco y fila activadas. Si fuesen en distintos bancos o filas, el controlador de memoria SDRAM completará todas las lecturas en proceso y activará el nuevo banco y la nueva fila antes de activar las señales opBegun y earlyOpBegun.

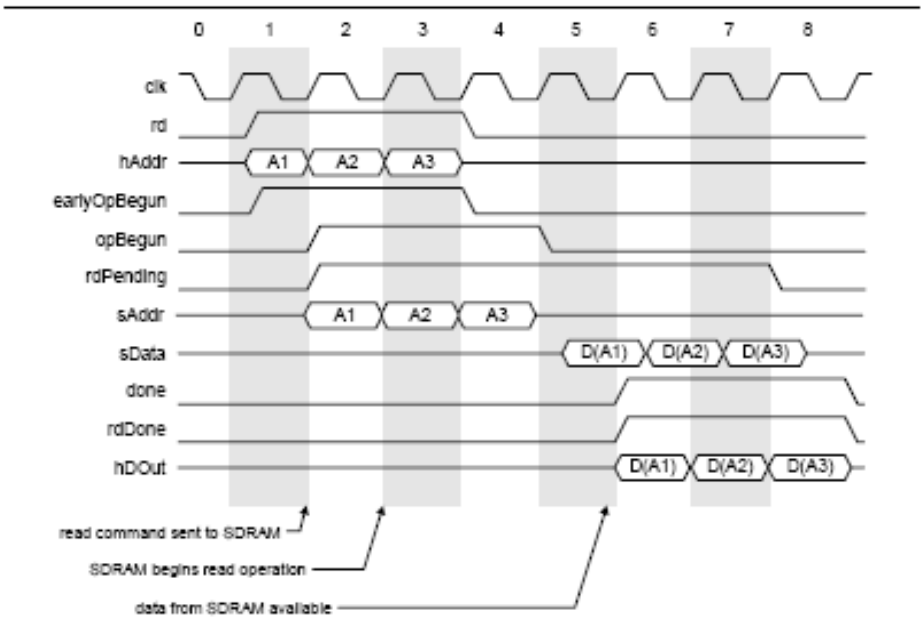

Figura 15: Operación de lectura en modo ráfaga de un controlador SDRAM

Mientras mantengamos la señal rd a alta, se irán mandando las operaciones de lectura a la memoria. Necesitaremos ir introduciendo la nueva dirección de memoria a leer cada vez que la señal opBegun se active indicando que la lectura previa se ha iniciado correctamente.

Podemos ver en la figura como se van activando y desactivando las distintas señales. Por ejemplo, hasta el inicio del ciclo 3 (flanco de subida del reloj) no se inicia la operación de lectura ya que la señal opBegun se mantenía a baja y es en el ciclo 2 cuando se manda por primera vez la operación de lectura. También se puede ver como hasta el ciclo 6 no se

activa la señal done que indicará que los datos se pueden coger del bus hDout con seguridad.

Veamos ahora un diagrama con las principales señales durante el proceso de escritura en modo ráfaga (figura 16). En este ejemplo se asume que las tres operaciones de escritura son sobre el mismo banco y fila activadas. Si fuesen en distintos bancos o filas, el controlador de memoria SDRAM completará todas las escrituras en proceso y activará el nuevo banco y la nueva fila antes de activar las señales opBegun y earlyOpBegun.

Mientras mantengamos la señal wr a alta, se irán mandando las operaciones de escritura a la memoria. Necesitaremos ir introduciendo la nueva dirección de memoria donde escribir cada vez que la señal opBegun se active indicando que la escritura previa se ha iniciado correctamente.

En este ejemplo vemos que necesitamos menos ciclos de reloj para mandar una operación de escritura a la SDRAM. Esto es debido a que no necesitamos esperar a que nos devuelva ningún dato. En el comienzo del ciclo 2 se mandará la operación de escritura y es en el ciclo 3 cuando la SDRAM empezará la escritura propiamente dicha.

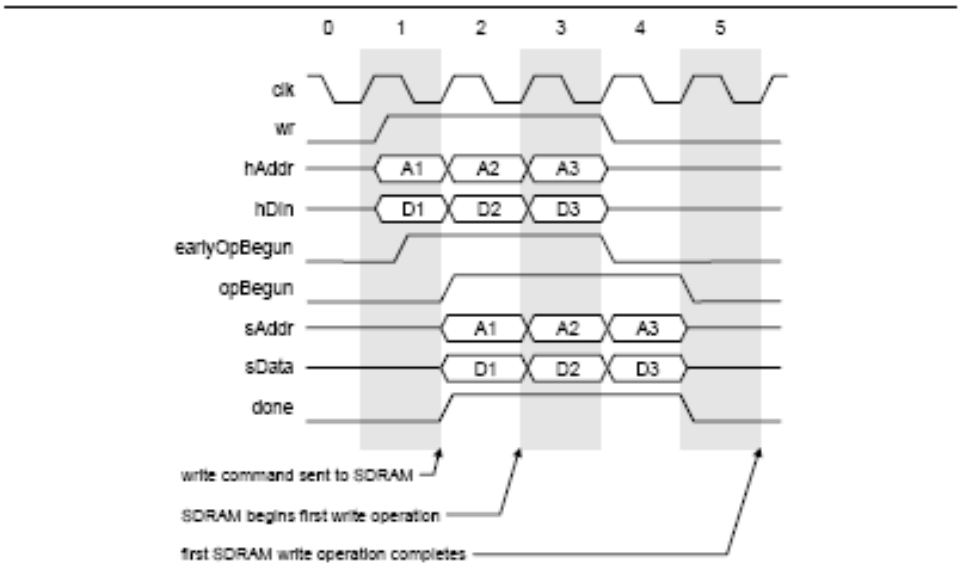

Figura 16: Operación de escritura en modo ráfaga de un controlador SDRAM

En ambos casos, si se está haciendo el refresco de una fila, el controlador de la memoria termina el refresco, vuelve a activar el banco y la fila de la dirección pasada por el bus y procede a realizar la operación requerida. Al retrasar la ejecución de la operación, la señal opBegun y

earlyOpBegun se mantendrán a baja hasta activarse más tarde cuando progrese la operación normalmente.

3.5. Sistema Completo

A continuación explicaremos mediante máquinas de estados el funcionamiento del sistema completo realizado en hardware. Dividiremos el comportamiento del problema en tres máquinas de estados, donde cada una de ellas realizará una tarea diferenciada totalmente de las otras, y explicaremos la distribución de la memoria.

Comenzaremos con la distribución de la memoria (figura 17), que estará dividida en cuatro zonas, donde almacenaremos los datos necesarios del problema. En la primera de ellas encontraremos, empezando en la dirección cero de la memoria, los parámetros de cada una de las células, es decir, para la primera célula necesitaremos espacio para sus seis parámetros específicos (cada parámetro necesitará dos bytes para almacenarse).

Seguidamente colocaremos en la segunda zona de la memoria la dirección de cada par de células que tengan afinidad de parámetros. Esto ocupará cuatro bytes por pareja (dos para la dirección de cada célula).

En la tercera zona de la memoria encontraremos para cada célula el número de cluster al que esta asociada. Esta zona de pares, la situaremos en la primera dirección de memoria libre cuyos dos bytes menos significativos coincidan con los dos bytes menos significativos donde están guardados los parámetros de las células. De esta forma identificaremos el cluster asociado a cada célula, ya que coinciden los dos bytes menos significativos de estas dos partes de la memoria. Con esto evitamos tener que guardar la dirección de la célula junto con el cluster asociado aunque desaprovechamos algo de memoria.

Para finalizar en la cuarta y última zona de la memoria guardaremos la suma de los parámetros acumulados en cada cluster junto con el número de células que lo forman. Esta parte será direccionada a partir de una dirección base (la primera dirección libre que encontremos después de la tercera parte de la memoria) sumándole el cluster asignado a cada célula multiplicado por el número de bytes necesarios para almacenar los datos del cluster.

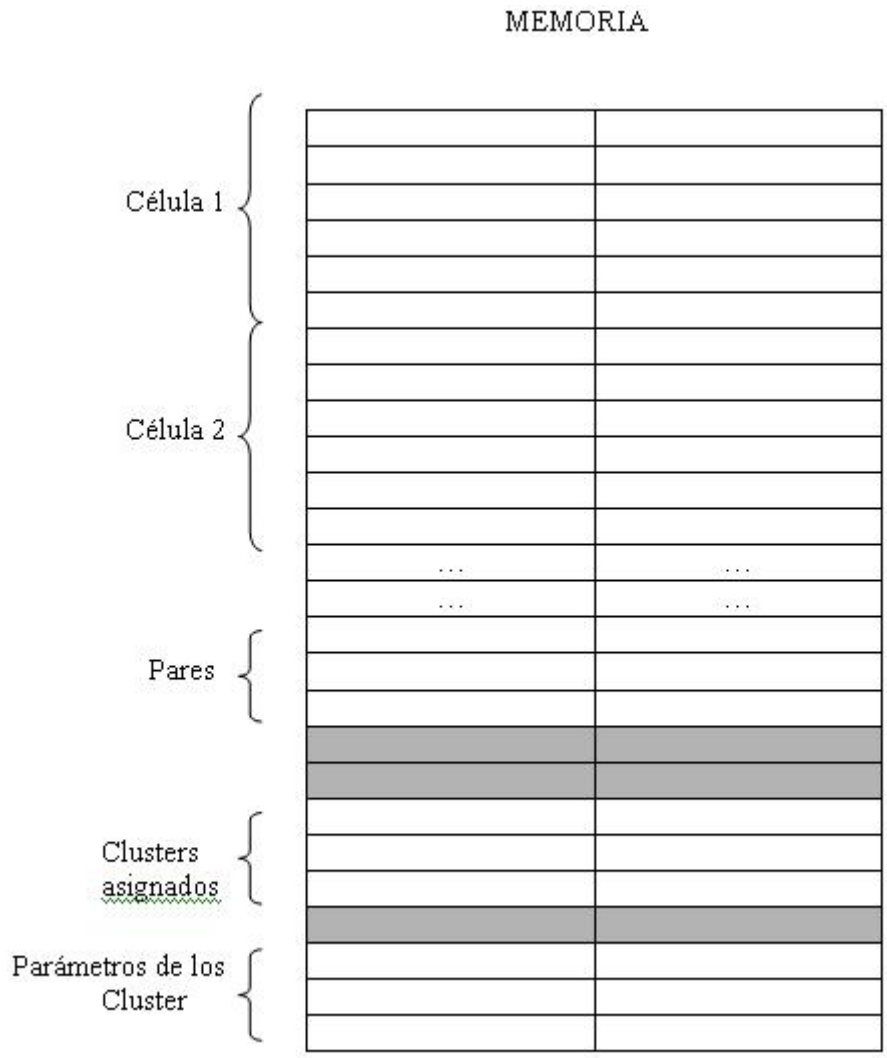

Figura 17: Esquema de la distribución de la memoria de datos

Suponemos cargados los datos de las células en memoria a partir de la dirección cero de la memoria.

La primera parte de la máquina de estados (figura 18) es la encargada de ir leyendo los parámetros de las células y escribir aquellos pares de células similares en la segunda parte de la memoria. Veamos los diferentes estados:

- S0 → Partimos del estado inicial donde se prepara la memoria para trabajar con ella, se inicializa la dirección de la primera célula a comparar con el resto (dirA), la dirección a partir de la cual escribiremos las direcciones de los pares de células (dirPar = nºcelulas*2bytes*6 parámetros). Pasamos al estado S1.
- S1 → Procederemos a la lectura de los parámetros de la primera célula. Esto nos llevará seis ciclos donde tendremos que mantener la

señal de lectura rd = 1 y la dirección dirA a la entrada del bus hAddr. Pasamos al estado S2.

- S2 → Inicializamos una segunda dirección (dirB) que contendrá la dirección de la siguiente célula apuntada por dirA. Pasamos al estado S3.
- S3 → Procederemos a la lectura de los parámetros de la segunda célula. Esto nos llevará seis ciclos donde tendremos que mantener la señal de lectura rd = 1 y la dirección dirB a la entrada del bus hAddr. Pasamos al estado S4.
- S4 → Una vez hayamos cargado los dos grupos de registros con los parámetros de las células se realiza la comparación entre ellas obteniendo en la salida un 1 en el caso de que sean similares y un 0 en caso contrario.

  Si el resultado de la comparación es cero y hemos terminado de recorrer todas las células con las que comparar la primera (es decir dirB = nºcélulas * 2 bytes * 6 parámetros) saltamos al estado S1.

  Si el resultado de la comparación es cero y no hemos terminado de recorrer todas las células con las que comparar la primera (es decir dirB ≠ nºcélulas * 2 bytes * 6 parámetros) saltamos al estado S3.

  Si el resultado de la comparación es cero y hemos terminado de recorrer todas las células con las que comparar y las comparadas, hemos terminado esta fase del proceso y pasaríamos a la máquina de estados de asignar los clusters a las células.

  Si el resultado de la comparación es uno vamos al estado S5.
- S5 → En este estado pasaremos a escribir en memoria las direcciones de memoria dirA y dirB de las células en la zona de pares. Para ello mantendremos la señal rw = 1 durante 2 ciclos (para escribir 4 bytes) y en la entrada hAddr = dirPar que posteriormente será incrementada para apuntar a la siguiente dirección libre.

  Si hemos terminado de recorrer todas las células con las que comparar y las comparadas, hemos terminado esta fase del proceso y pasaríamos a la máquina de estados de asignar los clusters a las células.

  Si no hemos terminado de recorrer todas las células con la que comparar pasamos al estado S3.

  Si hemos terminado de recorrer las células con las que comparar pero no las comparadas, vamos al estado S1.

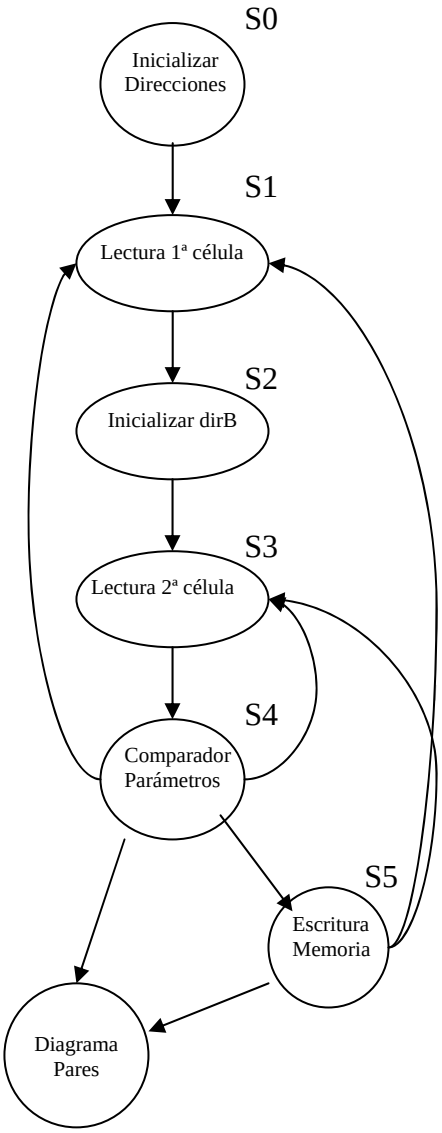

Figura 18: Diagrama de la primera parte de la máquina de estados

La segunda parte de la máquina de estados (figura 19) es la encargada de ir leyendo los pares de direcciones de célula anteriormente escritos en memoria e ir asignando a cada célula el cluster correspondiente. Veamos los diferentes estados:

- S6 $\rightarrow$ Inicializamos el contador de clusters contClus = 0 y la dirección de lectura (dirA) a la dirección donde se encuentre el primer par. Además inicializamos la dirección de inicio de asignación de los clusters (dirClus). Como se ha dicho anteriormente, ésta será la

primera dirección libre cuyos dos bytes menos significativos coincidan con las direcciones de las células. Pasamos al estado S7.

- S7 → Leemos la dirección de la primera célula del par manteniendo rd = 1 y hAddr = dirA. Se pasa al estado S8.
- S8 → Leemos la parte de la memoria donde se encuentra la asignación de los clusters de la célula actual (la dirección dirClus + dirA) guardando el dato en la variable datoClus.
  Si el dato es igual a 0 significa que no tiene ningún cluster asignado y pasaremos al estado S9.
  Si el dato es distinto de cero significa que tiene un cluster asignado y pasaremos al estado S10.
- S9 → Escribimos en la dirección de memoria dirClus + dirA el contador de cluster (contClus). Para ello podemos la señal wr = 1, metemos la dirección en el bus de datos hAddr y ponemos el contador de clusters en el bus hDin. Pasamos al estado S10.
- S10 → Leemos la dirección de la segunda célula del par manteniendo rd = 1 y hAddr = dirA que después de leer la dirección de la primera célula del par se ha incrementado. Se pasa al estado S11.
- S11 → Leemos la parte de la memoria donde se encuentra la asignación de los clusters de la célula actual (la dirección dirClus + dirA). Esto se guardará en la variable datoClusAnt.
  Si el dato es igual a 0 significa que no tiene ningún cluster asignado y pasaremos al estado S12.
  Si el dato es distinto de cero significa que tiene un cluster asignado pasaremos al estado S13 donde renombraremos los clusters.
- S12 → Escribimos en la dirección de memoria dirClus + dirA el datoClus (cluster asignado a su pareja) de su pareja. Para ello ponemos la señal wr = 1, metemos la dirección en el bus de datos hAddr y ponemos el datoClus en el bus hDin.
  Si hemos terminado de leer los pares se termina esta fase de asignación pasando a la última parte de la máquina de estados.
  Si no, pasamos al estado S7 incrementando el valor de dirA para que apunte al siguiente par a analizar.
- S13 → En este punto habría que recorrer la lista de clusters asignados y donde aparezca datoClusAnt lo sustituiremos por datoClus (renombramiento). Este proceso requerirá lecturas y escrituras por lo que consumirá muchos ciclos de reloj.
  Si hemos terminado de leer los pares se termina esta fase de asignación pasando a la última parte de la máquina de estados.
  Si no, pasamos al estado S7 incrementando el valor de dirA para que apunte al siguiente par a analizar.

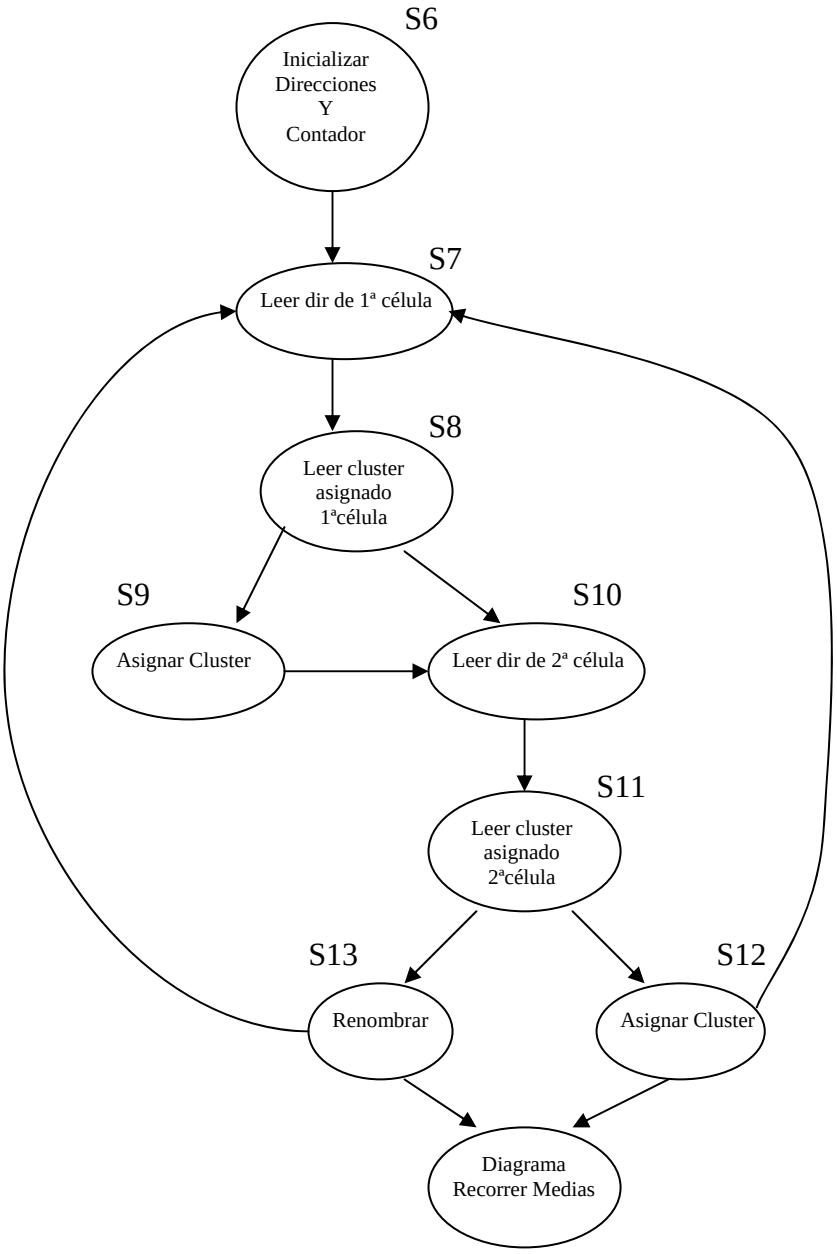

Figura 19: Diagrama de la segunda parte de la máquina de estados

La tercera parte de la máquina de estados (figura 20) es la encargada de ir sumando los parámetros de cada cluster y de contar el número de células que pertenecen a cada uno, así como de calcular sus medias. Veamos los diferentes estados:

- S14 → Inicializamos una dirección base (dirBase) con la primera dirección libre después de la asignación de clusters. Además necesitamos una dirección de lectura para ir recorriendo la asignación de clusters de cada célula (dirLec) que se inicializará a dirClus. Pasamos al estado S15.
- S15 → Leemos dirLec y guardamos el dato en dirDesp. Pasamos al estado S16.
- S16 → Accedemos a la dirección de la célula actual (principio de la memoria donde los bytes menos significativos coinciden con dirLec), para leer los parámetros de la célula y lo guardamos en datosCel. Pasamos al estado S17.
- S17 → Leemos la dirección de memoria dirBase + dirDesp donde se encontrarán los parámetros y el número de células del cluster de la célula considerada. En este punto se deberán sumar los nuevos parámetros e incrementar en uno el contador de células del cluster. Esto conllevará varias lecturas y escrituras que consumirán muchos ciclos de reloj. Pasamos al estado S18.
- S18 → Incrementamos dirLec. Si ya hemos terminado de recorrer todas las células haremos la media aritmética de cada cluster pasando al estado S19.
  Si no pasamos al estado S15.
- S19 → Leemos los parámetros de cada cluster y los dividimos entre el número de células que tiene cada uno. Así obtenemos las medias que es el resultado que queríamos obtener del análisis. Otra vez esto conllevará muchas lecturas y escrituras. Una vez se recorren todos los clusters habremos finalizado el análisis y pasaremos al estado final de la máquina de estados.

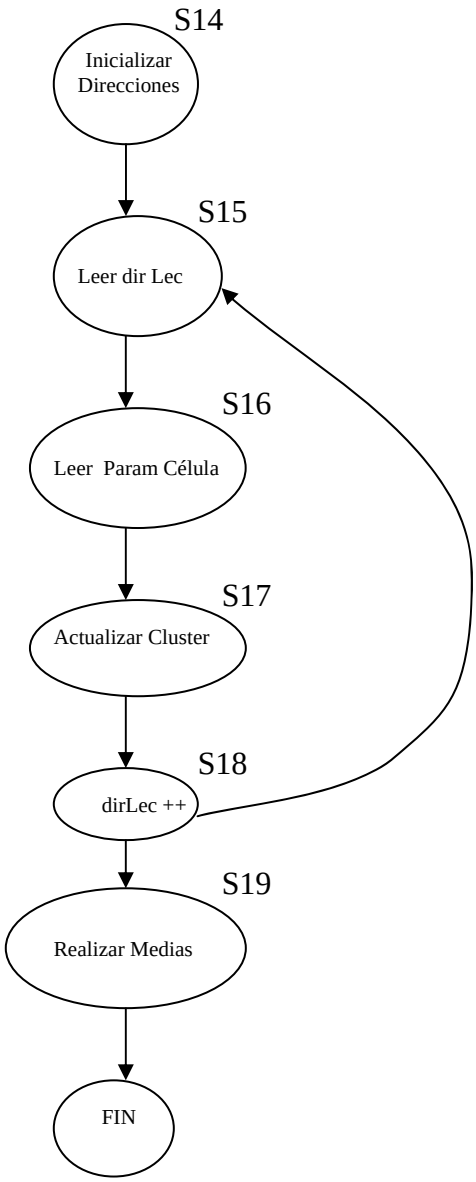

Figura 20: Diagrama de la tercera parte de la máquina de estados

Como se puede apreciar la máquina de estados es muy compleja debido a los múltiples accesos a memoria tanto para escribir como para leer. Además, estos accesos no están siempre ordenados lo que supone un mayor tiempo de acceso.

También se ha de tener en cuenta que el tiempo que dura un ciclo de reloj es vital para la eficiencia del sistema puesto que necesitaremos muchísimos para ejecutar nuestro programa. Por ello cuanto menor sea este tiempo más rápida será la ejecución del sistema.

4. Programas utilizados

En el desarrollo de este proyecto hemos utilizado fundamentalmente para la parte hardware el programa Xilinx ISE 7.1 [2], las XSTOOLS [1] y el programa VSystem. Hablaremos brevemente de en qué consiste cada una de estas herramientas.

4.1. XSTOOLS

Son unas pequeñas aplicaciones que proporciona la Xess Corp. que ofrecen al usuario una interfaz de comunicación sencilla entre el PC y la FPGA. Entre estas utilidades se encuentran:

- GXSTEST

Esta aplicación proporciona una manera de comprobar si la FPGA funciona correctamente. Para ello, mediante la selección en la ventana de ejecución del programa del tipo de placa y puerto paralelo por el que se comunica con el PC (Figura 21) y accionando el botón TEST, se hace ejecutar un pequeño programa que adecuado para cada placa, comprueba todos sus componentes. La placa mostrará por uno de sus displays el resultado del test. Si todo funciona correctamente mostrará 0, mientras que si la prueba falla mostrará E y una ventana con los posibles errores.

Figura 21: gxtest

- GXSSETCLK

Esta aplicación nos permite dividir la frecuencia de trabajo del oscilador de la placa, que trabaja a 100Mhz, en factores de 1, 2,. . . en adelante. El manejo de esta utilidad es muy sencillo, simplemente consiste

en introducir en la ventana de programa (Figura 22) el factor por el que queremos dividir el reloj, el tipo de placa y el puerto paralelo.

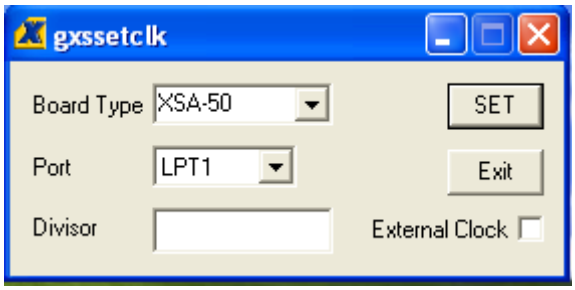

Figura 22: gxssetclk

- GXSLOAD

Esta es sin duda una de las utilidades más importantes de las XSTOOLS, ya que nos permite descargar nuestros diseños en la placa las veces que se deseen, para depurar o comprobar que funcionan. Para utilizarlo simplemente tenemos que arrastrar el archivo .BIT de nuestro circuito a la columna nombrada como FPGA/CPLD (Figura 23), seleccionar el tipo de placa, el puerto paralelo y dar a LOAD.

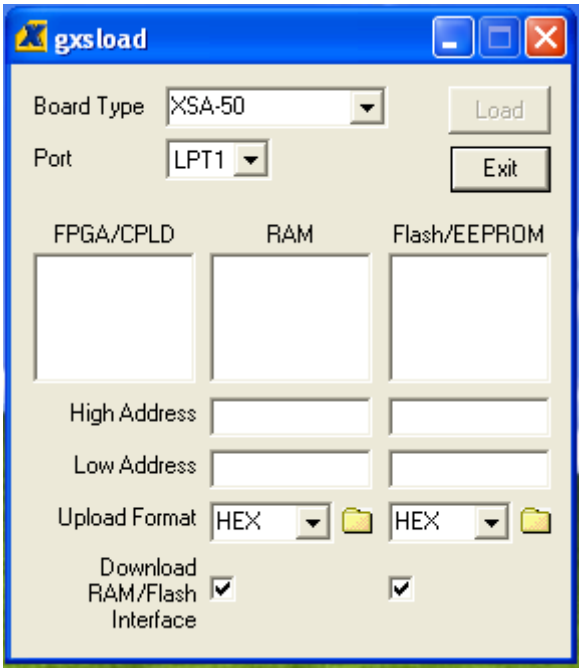

Figura 23: gxsload

Pero esta aplicación no sólo realiza esta tarea, si no que también permite cargar y descargar datos de la RAM y de la memoria Flash. Los datos se cargan a través de archivos .EXO, .MCS ó .HEX, que se sitúan en la columna correspondiente a RAM o Flash, según la que se desee cargar. Para descargar el contenido de las memorias, primero deben seleccionarse el rango de direcciones en los que guardaremos los datos, seleccionar el tipo de codificación que deseemos y arrastrar la carpeta dibujada en la ventana. El programa creará un archivo denominado RAMUPLD con la extensión elegida con el resultado de la descarga.

4.2. Xilinx ISE 7.1

El Integrated Software Environment (ISETM) es la suite de diseño software de Xilinx. Permite transformar nuestro diseño a un formato capaz de programar la FPGA para que realice la función de nuestro circuito. El navegador del proyecto maneja y procesa nuestro diseño según el flujo del diseño de ISE mediante los pasos siguientes:

- Entrada de diseño

Es el primer paso en el flujo del diseño de ISE. En este punto, se crean los archivos fuente basados en nuestros objetivos de diseño. Se pueden crear en lenguaje de alto nivel usando HDL, por ejemplo VHDL, Verilog, o ABEL, o usando un diagrama esquemático.

- Síntesis

Después del paso anterior y de una simulación opcional, se ejecuta el paso de síntesis. Durante éste, VHDL, Verilog, o diseños en varios lenguajes, se convierten en archivos de netlist que se aceptan como entrada para el paso siguiente.

- Implementación

En esta etapa se convierte el diseño lógico en un archivo físico que se puede descargar al dispositivo seleccionado. En el navegador del proyecto, se puede seleccionar si el proceso de implementación quiere realizarse en un paso o en varios. Los procesos de implementación varían dependiendo de si se selecciona una FPGA o un CPLD.

- Verificación

Se puede verificar la funcionalidad de nuestro diseño en varios puntos del flujo del diseño. Para ello puede utilizarse el software del simulador para verificar la funcionalidad y la sincronización del diseño o de una porción de él.

El simulador interpreta VHDL o el código de Verilog y muestra resultados lógicos del HDL descrito para determinar que el circuito funciona correctamente. La simulación permite crear y comprobar funciones complejas en una cantidad de tiempo relativamente pequeña.

- Configuración de dispositivo

Después de generar el archivo de programación del dispositivo, éste se configura. Durante la configuración, se generan archivos de configuración y se descarga el archivo de programación del ordenador al dispositivo de Xilinx.

4.3. VSystem

A continuación nos centraremos en otra herramienta utilizada en nuestro proyecto, que sirve para simular circuitos diseñados en el lenguaje de programación vhdl.

Estudiaremos el lenguaje VHDL, y nos familiarizaremos con el entorno de trabajo de la herramienta V-System de Model Technology.

Un circuito escrito en VHDL consta básicamente de una entidad y de una o más arquitecturas. La entidad indica el número, nombre y anchura de cada uno de los puertos de entrada y de salida del circuito. Y la arquitectura recoge el funcionamiento del circuito, es decir, cómo se propagan los valores de las entradas del sistema para generar las salidas del mismo.

Una vez realizado el diseño comprobaremos que este diseño sea correcto, para ello vamos a simularlo utilizando la herramienta V-System. En primer lugar generaremos un fichero con el código correspondiente al diseño a implementar, para ello utilizaremos cualquier editor de textos y salvaremos la práctica en el directorio c:\hlocal con el nombre diseño.vhd.

A continuación abriremos la herramienta V-System de Model

Technology, y nos aseguraremos que el directorio de trabajo es el directorio en el que hemos guardado nuestra práctica, c:\hlocal. Para ello utilizaremos el cuadro de diálogo Directory que encontraremos en el menú File de la herramienta, y en él cambiaremos el directorio de trabajo a c:\hlocal (como muestra la figura 24), sin olvidar pulsar seguidamente el botón Change To.

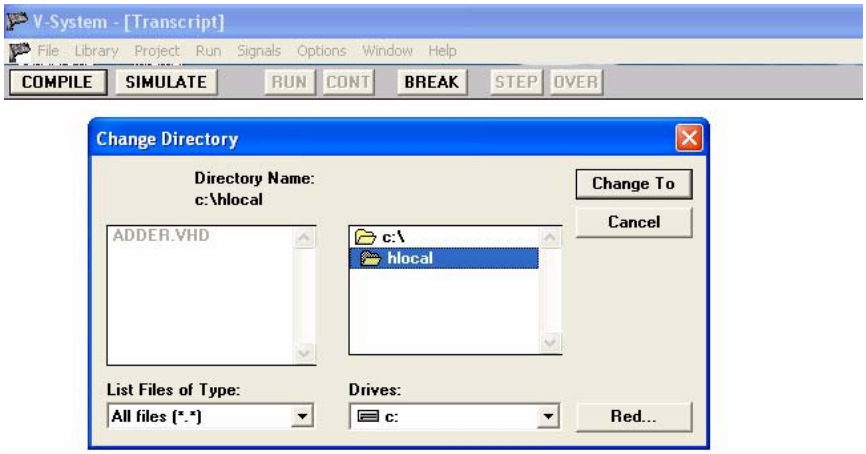

Figura 24: Cambiar directorio de trabajo de V-System

Necesitamos generar ahora una librería de diseño para almacenar los resultados de la compilación. Lo hacemos mediante el cuadro de diálogo New que encontraremos en el menú Library de la herramienta (véase figura 25). El nombre que debemos dar a la librería es work. Esto generará un nuevo directorio llamado work en c:\hlocal que no sería válido si lo hubiéramos creado con la orden crear nueva carpeta de Windows.

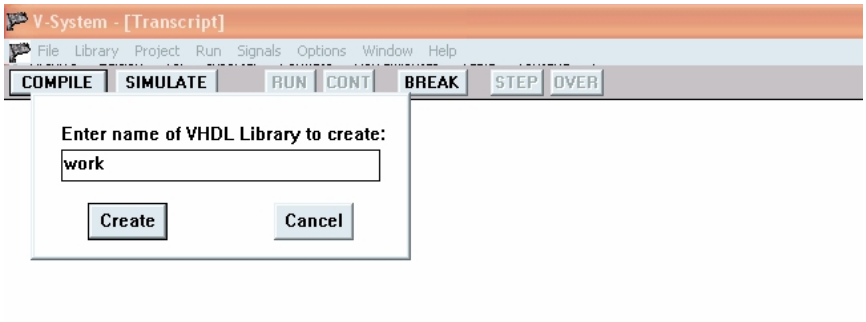

Figura 25: Crear una librería Vhdl en V-System

Seguidamente es necesario crear un Archivo de proyecto nuevo sobre el directorio de trabajo, para lo cual emplearemos el cuadro de

diálogo New del menú Project de la herramienta. Elegiremos como nombre vsystem.ini como puede verse en la siguiente figura.

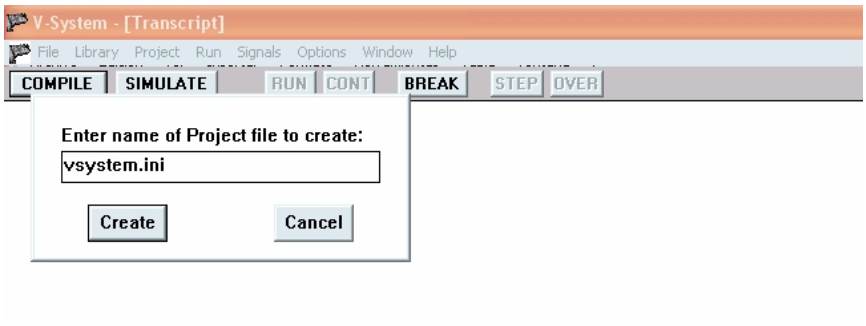

Figura 26: Crear un nuevo proyecto en V-System

Tras estos primeros ajustes de la herramienta estamos ya listos para compilar la práctica. Pulsamos para ello el botón COMPILE que encontraremos en la parte superior izquierda de la pantalla principal de la herramienta, y seleccionamos nuestro fichero, a continuación pulsamos el botón COMPILE y finalizamos pulsando DONE.

Una vez compilada la práctica y comprobado que no tienen errores podemos simular el circuito diseñado. Pulsamos el botón SIMULATE (situado junto a COMPILE) apareciéndonos una ventana donde seleccionamos la entidad y arquitectura a simular y pulsamos OK (mostrado en la figura 27).

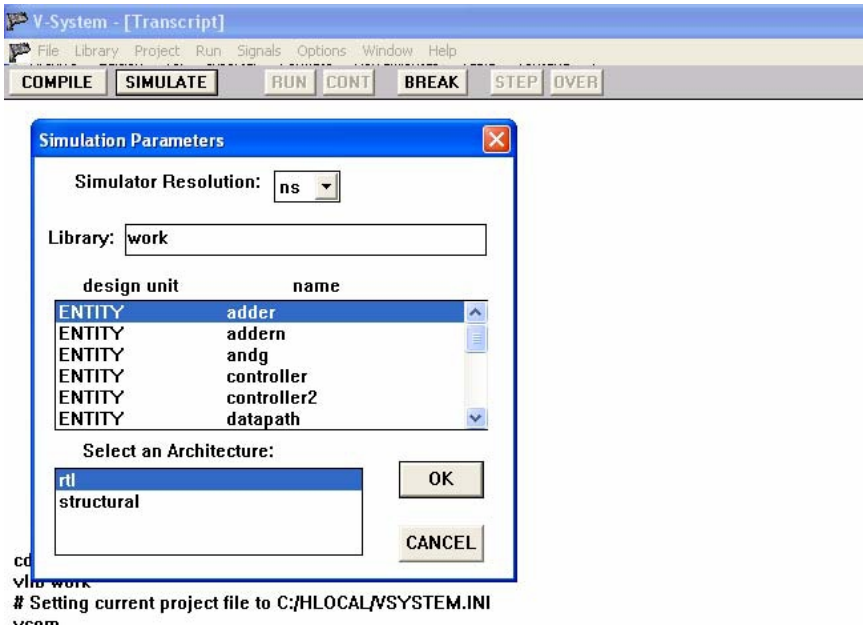

Figura 27: Simulación de un diseño Vhdl con V-System

Tras esta orden han aparecido 8 nuevas ventanas. Mediante las opciones Restore All del menú Windows y Tile Horizontally también del menú Windows (Tile Vertically también puede servir) conseguimos ver todas las ventanas simultáneamente.

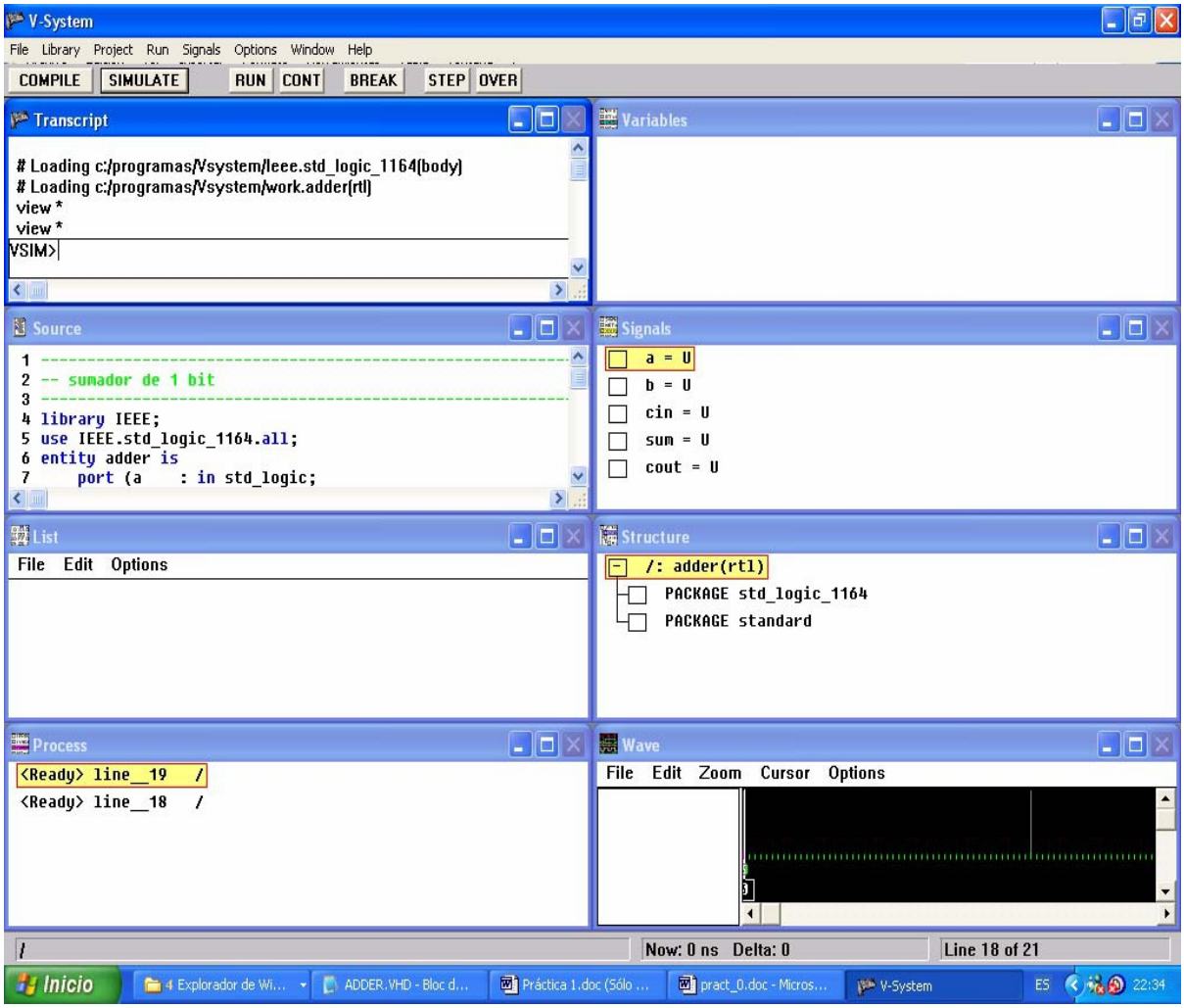

Figura 28: Entorno de simulación de V-System

Sobre la ventana Transcript se pueden dar órdenes que afectan al sistema. Para que en la ventana List aparezcan los resultados de la simulación en la pantalla Transcript escribiremos la orden: list *

Y para que en la ventana Wave aparezcan los resultados de la simulación, en forma de ondas, en la pantalla Transcript escribiremos la orden: wave *

Para poder simular nuestro diseño hay que aplicar estímulos a las

señales de entrada. Esto se puede hacer si se ha creado un banco de pruebas o mediante la orden force. Por ejemplo daremos a la señal a (que debe estar creada en la entidad de nuestro diseño) el valor 0 en el instante inicial (0 ns) y el 1 en el instante 50 ns, escribimos para ello la siguiente orden en la pantalla Transcript: force a 0 0, 1 50

Si queremos que además se repita esta orden cada 100 ns utilizaremos la orden: force a 0 0, 1 50 -repeat 100

Para dar valores a las señales de entrada podemos utilizar el cuadro de diálogo Force Signal del menú Signals de la herramienta.

A continuación ejecutamos la simulación utilizando la orden run sobre la ventana Transcript. Esta orden simula por defecto 100 ns. Si queremos una simulación más larga podemos por ejemplo especificar el número de ns escribiendo run y a continuación una cantidad, así por ejemplo run 1000 simularía 1000 ns. Otras opciones son:

run @5000 simula hasta el ns 5000

run -all simula hasta que se detenga pulsando el botón BREAK y mediante el botón CONT se reanuda la simulación.

Estas opciones se encuentran también en el menú Run de la herramienta.

Maximizando la pantalla List o Wave podemos comprobar el funcionamiento de nuestro diseño, para ello hemos de fijarnos que los valores de las señales de salida son los correctos en cada caso.

5. Posibles mejoras

A continuación realizaremos un estudio de nuestro trabajo desde el punto de vista de la optimización centrándonos en los factores a mejorar para el óptimo funcionamiento del sistema, tanto en cuestiones temporales (tiempo de ejecución) como de espacio (memoria).

Nuestro proyecto en sí, se basa en mejorar el tiempo de ejecución de un programa realizado en software, realizándolo en hardware. Utilizaremos el hardware para realizar ciertas operaciones al mismo tiempo, mientras que la programación software tiene que realizarlas de una en una por la falta de recursos.

Comenzaremos hablando del paralelismo utilizado en nuestro proyecto, que es una solución muy útil para mejorar el tiempo de ejecución de distintos programas realizados en hardware. En nuestro caso tenemos que ir realizando comparaciones de distintos parámetros celulares para determinar si podemos agruparlos o no, derivando en el diagnóstico de ciertas enfermedades. Estas comparaciones tienen un gran costo desde el punto de vista temporal, puesto que debemos ir leyendo cada uno de los parámetros y compararlos con el resto de células en bucles de grandes dimensiones, y además dentro de los mismos sólo podemos realizar una comparación a la vez (realizado en software).

Sin embargo al realizar este proyecto en hardware, más concretamente en FPGAs, podemos aprovechar el paralelismo para conseguir un ligero ahorro de tiempo a la hora de hacer las comparaciones de los parámetros siendo posible realizarlas a la vez y conseguir los resultados en un menor espacio de tiempo. Esto vendrá limitado por el número de comparadores que nos permita tener a la vez la FPGA.

Para beneficiarnos del paralelismo desde el punto de vista de la memoria la situación se complica bastante. En nuestro problema, para las comparaciones debemos de ir leyendo los parámetros celulares de una manera secuencial, ya que están colocados consecutivamente en memoria, que es una ventaja con respecto a la lectura de datos de forma aleatoria.

Nuestro programa podría estar leyendo datos continuamente y según los va recibiendo realizar las comparaciones oportunas, optimizando el tiempo de toda la parte relativa a las comparaciones. El problema vendría cuando hay que almacenar los resultados obtenidos al realizar las comparaciones. Hay que permitir hacer escrituras después de algunas

comparaciones, y puesto que no se puede leer y escribir en memoria a la vez, tendremos que parar las operaciones de lectura de células.

En esta fase de escritura hay que almacenar datos que como hemos mencionado anteriormente no están colocados de forma secuencial respecto a las lecturas que se están realizando a la vez, lo que hace la operación de escritura más costosa en cuanto a tiempo de ejecución. Es decir, la velocidad de acceso a memoria viene determinada en gran parte por la secuencialidad o no, de los datos a leer.

Para sincronizar todas las operaciones, es complicado controlar cuándo se debe leer un dato, cuándo se debe escribir otro…sin solapar el uso de recursos. Para conseguir controlar las operaciones mandadas a la memoria, se podrían utilizar un buffer de lectura y otro de escritura (éste con mayor prioridad) para resolver el problema antes mencionado de la sincronización de operaciones. Con este buffer este problema quedaría resuelto y facilitaría notablemente la sincronización de operaciones con la memoria.

6. Bibliografía

- [1] Xess: http://www.xess.com
- [2] Xilinx: http://www.xilinx.com
- [3] Altera: http://www.altera.com
- [4] Lluís Teres, Yago Torroja, Serafín Olcoz, Eugenio Villar: VHDL Lenguaje Estándar de diseño electrónico. Mc Graw Hill (1998)
- [5] José Jaime Ruiz Ortiz: VHDL de la tecnología a la arquitectura de computadores. Editorial Síntesis.
- [6] Ricardo Peña Marí: Diseño de programas, formalismo y abstracción. Prentice Hall (1998).
- [7] Narciso Martí Oliet, José Alberto Verdejo López, Yolanda Ortega Mallén: Estructuras de datos y métodos algorítmicos (2003)
- [8] Perl: http://www.perl.com
- [9] Perl: www.fdi.ucm.es/profesor/mozos/BIO/master_2005b.ppt
- [9] FPGA: http://www.dacya.ucm.es/mendias
- [10] Datos biológicos: http://www.germanstrias.org/cast/instal.htm
- [11] Datos biológicos: http://www.cbm.uam.es/mkfactory.esdomain/webs/cbmso/plt_Servicio_Pagina.aspx?IdServicio=5&IdObjeto=200
- [12] FPGA: http://www.cs.unc.edu/~lastra/comp190/Notes/13-Xilinx-FPGAs.ppt