Sistemas Informáticos
Curso 2004-2005

Comparación de secuencias genómicas e identificación de proteínas utilizando FPGAS

Míriam Rubio Camarillo
Sergio Peris Iniesta
Javier López Fernández

Dirigido por:
Prof. Daniel Mozos Muñoz
Dpto. ACYA

Facultad de Informática
Universidad Complutense de Madrid

I Índice

II Tabla de figuras

III Palabras clave

FPGA, Array Sistólico, Smith-Waterman, Búsqueda genómica.

IV Resumen

La comparación de cadenas es una parte importante de muchos programas y aplicaciones, en particular, es creciente su uso en el terreno de la biología y la investigación científica. Miles de secuencias provenientes de enormes bases de datos de contenido genético son diariamente comparadas con este motivo. Por ello, se hace necesaria la utilización de algoritmos rápidos, y no sólo eso, sino que sus resultados sean lo más fiables posible.

Los algoritmos existentes actualmente se basan en la búsqueda exacta, es decir, en comprobar si una cadena es igual a otra dada, o en la búsqueda inexacta, consistente en hallar un coste o valoración que indicaría lo que una cadena difiere de otra. El algoritmo de Smith-Waterman pertenece a este segundo grupo y es el que hemos elegido para implementar la comparación entre secuencias de ADN, dado que es el mejor dentro de los algoritmos de búsqueda inexacta.

Utilizando Smith-Waterman quedaría resuelto el problema de la fiabilidad, pero también es muy importante la velocidad, ya que cuanto más rápido se obtenga el resultado, el trabajo de los investigadores o programas también se acelerará y por lo tanto mejorará. Una solución software del algoritmo se obtendría aproximadamente en un tiempo N *M, siendo N y M las longitudes de las cadenas a comparar. Mientras que una solución hardware aprovechando el paralelismo que aportan arquitecturas como los arrays sistólicos podría obtenerla en N + M. Con lo cual, si las cadenas son largas, como es el caso de las secuencias de ADN, la mejora es enormemente visible. Por ello, para implementar el sistema hemos elegido la opción hardware y para hacerlo utilizaremos FPGA's.

V Abstract

String comparison is an important part of many programs and applications. Its use is especially growing in biology and scientific research. For this reason, thousands of sequences coming from enormous data bases with genetic contents, are compared daily. Therefore fast algorithms with reliable results are necessary.

The currently existent algorithms are based either on exact or on inexact search. Exact search verifies if a string is equal to another given one, and inexact search consists in finding a cost or valuation which indicates the resemblance between two strings. The Smith-Waterman algorithm is based on inexact search and is the one we have chosen to implement the comparison of DNA strings given it is the best choice for inexact search.

By using Smith-Waterman the reliability problem is solved, but the speed is also very important due to the fact that the faster the result is obtained, the faster the work of researchers and programs is done and therefore improves. A software solution of the algorithm could be obtained in approximately N*M, where N and M are the lengths of the strings to compare. Meanwhile, a hardware solution could be obtained in N+M, taking advantage of the paralelism architectures, such as systolic arrays, offer. Therefore the improvement on large strings, for instance DNA sequences, is clearly visible. Because of this, to implement the system we have chosen the hardware approach using FPGA's.

VI Autorización

Los alumnos Sergio Peris Iniesta, Míriam Rubio Camarillo y Javier López Fernández 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. Fdo.

1 Descripción del problema

1.1 Base biológica [1]

Nuestro proyecto se basa en la comparación de secuencias de gran longitud en un dominio de 4 caracteres, en particular, en la aplicación de este sistema para cadenas de ADN. Por ello, en este apartado explicaremos con la mayor simplicidad posible en qué consiste la molécula de ADN para que nos podamos hacer una idea general de su estructura, así como imaginar la importancia científica y las aplicaciones en el campo de la investigación que se le pueden dar a la funcionalidad del sistema que hemos implementado.

1.1.1 ¿Qué es el ADN?

El ADN es la molécula de la vida, es portadora de la información genética y junto al ARN es uno de los dos ácidos nucleicos presentes en todas las células. El conjunto completo del ADN de un organismo se denomina genoma.

Está formado por nucleótidos, que son uniones de ácido fosfórico, una pentosa (la D-ribosa o la 2-desoxi-D-ribosa) y una base nitrogenada.

Los nucleótidos se enlazan entre ellos para formar la molécula.

Existen dos tipos de bases nitrogenadas:

Purinas: ADENINA y GUANINA.

Pirimidínicas: CITOSINA, TIMINA y URACILO (sólo ARN).

Las cuatro bases nitrogenadas del ADN se encuentran distribuidas a lo largo de la "columna vertebral" que conforman los azúcares con el ácido fosfórico en un orden particular, (la secuencia del ADN).Y se emparejan mediante puentes de hidrógeno de la siguiente manera (ver Figura 1): la adenina (A) con la timina (T) mientras que la citosina (C) lo hace con la guanina(G).

La estructura primaria del ADN está determinada por esta secuencia de bases ordenadas sobre la "columna" formada por azúcar (pentosa) y fosfato. Este orden es en realidad lo que se transmite de generación en generación.

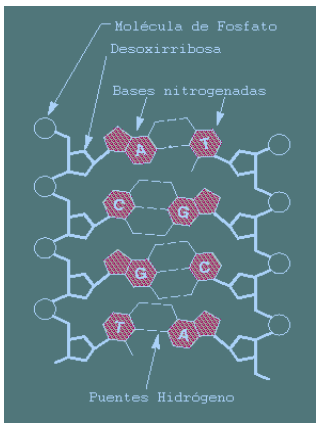


Figura 1 Emparejamiento de las cadenas de nucleótidos

La estructura en doble hélice del ADN ilustrada en la Figura 2 (dos hebras o cadenas de nucleótidos se encuentran arrolladas una alrededor de la otra formando una doble hélice), con el apareamiento de bases limitado (A-T; G-C), implica que el orden o secuencia de bases de una de las cadenas delimita automáticamente el orden de la otra, por eso se dice que las cadenas son complementarias. Una vez conocida la secuencia de las bases de una cadena, se deduce inmediatamente la secuencia de bases de la complementaria.

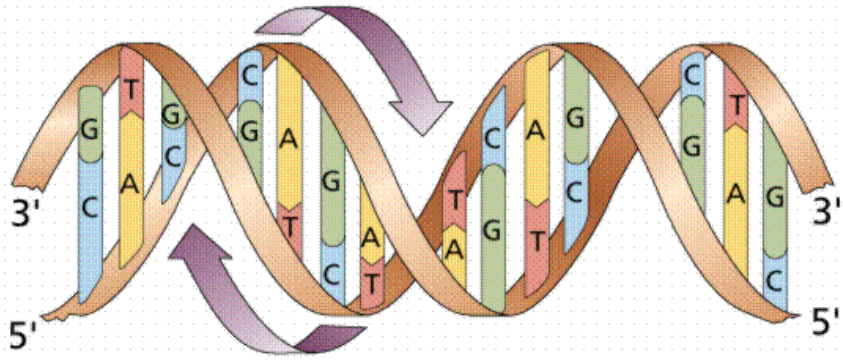


Figura 2 Modelo de doble hélice postulado por Watson y Crick

1.1.2 El problema de la comparación de secuencias

Nuestro problema consiste en determinar cuánto se parece una secuencia de ADN a otra. Pero este problema se puede aproximar de muchas maneras diferentes:

Para empezar, hay situaciones en las que nos interesa una respuesta tan sencilla que resulta fácil de obtener. Un ejemplo es buscar una correspondencia exacta (sin mutaciones, inserciones/eliminaciones, sólo coincidencia) entre una de las dos secuencias. Este es uno de esos problemas de fácil solución, ya que sólo habrá que decir si es verdadero o falso que las secuencias son idénticas.

Una segunda clase de problemas puede ser formalizado con facilidad: como buscar la similitud permitiendo errores. Podemos recurrir a métodos bien conocidos ([1], [2], [3], [4]) y eficientes que llevarán algo más de tiempo pero hallarán igualmente una respuesta correcta, incluso nos ordenarán los resultados de forma útil.

Otro tipo de problemas pueden requerir analizar muchas posibilidades y resultar en largos tiempos de cálculo. Un ejemplo es buscar permitiendo inserciones/eliminaciones en la secuencia. Hay métodos que encontrarán las similitudes y las clasificarán en orden de calidad acomodando ambigüedad y huecos en la comparación, pero resultan habitualmente demasiado lentos e imprácticos a menos que se usen sobre conjuntos reducidos de secuencias o se disponga de un computador especializado. En estos casos a menudo resulta más interesante usar soluciones aproximativas, que aunque no garanticen encontrar el mejor encaje funcionan suficientemente bien en la mayoría de los casos y son mucho más rápidas permitiendo trabajar eficientemente.

Finalmente, hay problemas que son imposibles de analizar usando la tecnología existente. Un ejemplo es buscar permitiendo traslocaciones (cambio de ubicación de fragmentos de secuencias), inversiones y otros eventos evolutivos complejos. Estos llevan a una explosión exponencial de posibilidades y no pueden analizarse prácticamente ni siquiera para dos secuencias, mucho menos para un número elevado de secuencias. No hay forma de que hoy día podamos considerar estas posibilidades, pero aún así, en muchos casos podremos arreglarnos con una solución aproximada.

1.1.2.1 Algoritmos sencillos de baja complejidad de cálculo

Estos algoritmos son métodos aproximativos, que también permiten coincidencias exactas, errores y huecos, pero que no garantizan encontrar las mejores comparaciones y pueden obviar algunas similitudes significativas. Sus resultados requieren una interpretación más detenida antes de aceptarlos, pero son mucho más rápidos y a menudo proporcionan resultados satisfactorios. Los más populares son FASTA y BLAST ([3], [4]).

Ya que a veces comparar dos cadenas de ADN requiere un alto número de comparaciones y habitualmente los investigadores por problemas económicos o por el desconocimiento no tienen acceso a maquinaria especializada como un computador masivamente paralelo, una FPGA (el coste de una FPGA es ínfimo con respecto a las otras alternativas de hardware aquí propuesto), o un Biocelerador sólo les queda una alternativa: usar los métodos rápidos, aproximativos de FASTA y/o BLAST.

BLAST es en realidad una familia de programas que realizan varios tipos de búsqueda de secuencias. Para comparar las secuencias, BLAST toma varios residuos a la vez (esto se denomina una palabra). BLAST compara cada palabra de ambas secuencias y asigna una puntuación a esa coincidencia puntual. Esta puntuación se calcula considerando el grado de parecido entre los residuos en cada posición. Dicho de otro modo, BLAST permite errores y ambigüedad en las comparaciones.

A continuación BLAST intenta unir palabras y hallar el máximo segmento contiguo de palabras similares. Este se denominará *segmento máximo de secuencias apareadas* y representa una región de similitud sin huecos. Las puntuaciones de cada palabra se suman y se calcula el total para el segmento. BLAST trabajará con cada una de estas regiones por separado, es decir, BLAST no permite la presencia de huecos en las regiones similares. Esto no debería ser un problema, ya que todas las regiones se analizan, aunque se listen luego por separado en el informe final.

A medida que encuentra semejanzas, BLAST toma decisiones sobre cómo alinearlas y manejarlas en base a un análisis estadístico de las secuencias, descartando lo que considera que puede deberse a similitudes al azar o carentes de significado. Esto acelera el proceso pero pueden perderse semejanzas entre secuencias cortas o muy frecuentes.

FASTA también hace referencia a un conjunto de programas, cada uno pensado para una combinación específica de un tipo de secuencia y base de datos. Se basa en un método desarrollado por Pearson y Lipman y trabaja sobre premisas distintas a las de

BLAST, por lo que puede producir resultados diferentes. Habrá ocasiones en que FASTA funcione mejor o halle semejanzas que pasaron desapercibidas para BLAST y situaciones en que ocurrirá al revés. Para ir más deprisa, FASTA compara varios residuos de golpe. Busca coincidencias exactas de este número reducido de residuos (palabra). Esta es una diferencia con BLAST: FASTA no toma en consideración posibles ambigüedades o coincidencias aproximadas en la comparación. Incluso si la secuencia contiene códigos de ambigüedad, los convierte a lo que considera la secuencia más probable antes de realizar la comparación.

Una vez comparadas todas las palabras, FASTA intenta unir aquellas que coinciden en regiones contiguas y las reevalúa, esta vez considerando cambios conservativos y coincidencias menores de una palabra. De este modo localiza todas las regiones de semejanza, al igual que BLAST.

A continuación FASTA introduce una novedad importante: para cada secuencia selecciona las 10 regiones con mejor puntuación e intenta unirlas en una mayor, incluso aunque estén separadas: FASTA selecciona la región similar permitiendo huecos (indels), y calcula una puntuación global para la región con huecos.

Finalmente, FASTA ordena las secuencias por su región mayor de semejanza (tras unir secciones con huecos) y genera un alineamiento de mejor calidad usando el algoritmo de Smith-Waterman para hallar una nueva puntuación más fidedigna. Si ésta excede un umbral predeterminado que depende de su longitud, la secuencia se considera como una semejanza aceptable.

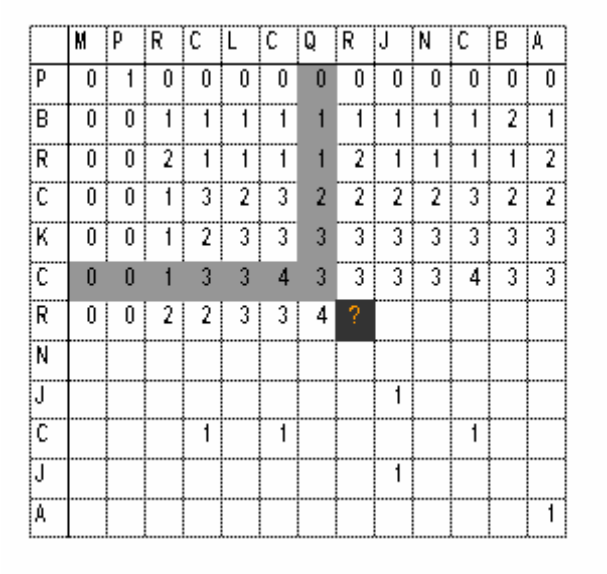
1.1.2.2 Algoritmos más complejos

Estos algoritmos se basan en métodos formales que permiten realizar búsquedas complejas permitiendo coincidencias exactas, ambigüedades e inserciones/eliminaciones. Se basan normalmente en dos algoritmos: Needleman-Wunsch o Smith-Waterman, como por ejemplo, los programas MPSRCH o BLITZ. Son más fiables, pero llevan más tiempo para terminar.

1.1.2.2.1 El algoritmo de Needleman-Wunsch

La idea principal del algoritmo es la de calcular el número más grande de bases nitrogenadas de una secuencia que se pueden emparejar con otra permitiéndose cualquier eliminación de elementos.

Antes de nada, es necesario establecer valores de similitud entre las diferentes bases nitrogenadas, el más simple es 1 si ambas son iguales, 0 en otro caso. Además de esto, es necesario decidir cuál va a ser la penalización de una eliminación o una inserción en cualquiera de las dos cadenas a comparar. La forma más eficiente de implementar el algoritmo es mediante programación dinámica y la definición del problema recursivamente es la siguiente:



	M	P	R	C	L	C	Q	R	J	N	C	B	A
P	0	1	0	0	0	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1	1	1	1	2	1
R	0	0	2	1	1	1	1	2	1	1	1	1	2
C	0	0	1	3	2	3	2	2	2	2	3	2	2
K	0	0	1	2	3	3	3	3	3	3	3	3	3
C	0	0	1	3	3	4	3	3	3	3	4	3	3
R	0	0	2	2	3	3	4	?					
N													
J									1				
C				1		1					1		
J									1				
A													1

Figura 3 Ejemplo de la matriz Needleman-Wunch

H_{ij}=max: 0<=m<j :max{max: 0<=k<i: H_k, j-1 – W_k + s(a_i,b_j), $H_{i-1,m}$ – W_m +s(a_i,b_j)}

- H_{ij} representa el máximo alineamiento entre la cadena a_0..a_i y la cadena b_0..b_j
- s(a_i,b_j) la similitud entre las bases nitrogenadas a_i y b_j
- W_k la penalización para que esa subcadena se convierta en la otra, el coste de los gaps.

El coste de este algoritmo en memoria es cuadrático mientras que en tiempo es cúbico porque para cada celda tiene que mirar en todos los elementos de la fila anterior y en todos los elementos de la columna anterior, como las cadenas de ADN son enormes su implementación sobre un ordenador comercial supone mucho tiempo de espera, por eso son más utilizados los algoritmos mencionados anteriormente.

1.1.3 Algoritmo de Smith-Waterman

Antes de explicar el funcionamiento del algoritmo hablaremos de qué hace que dos secuencias de ADN sean distintas: las mutaciones.

Las diferencias entre secuencias de ADN provienen del cruce entre los genomas de los antecesores y de las mutaciones aleatorias ocurridas durante el proceso de copia del ADN en el fenómeno de la división celular.

Hay tres tipos de mutaciones:

Tipo	Secuencia Original	Secuencia Mutada
Sustitución	AC**G**TA	AC**T**TA
Inserción	ACGTA	AC**A**GTA
Eliminación	ACGTA	ACTA

Cuanto mayor número de mutaciones haya entre dos secuencias evidentemente mayor será la diferencia entre ellas, y obtener este tipo de información como resultado de un algoritmo es importante, porque no lo es tanto averiguar si dos cadenas son iguales o no, si no cuánto de iguales, cuál sería el coste mínimo para transformar una cadena en la otra, este resultado es mucho más útil para los investigadores para por ejemplo poder hallar antecesores, evolución entre especies, y muchas otras aplicaciones. Y esto es lo que precisamente hace el algoritmo de Smith- Waterman.

El algoritmo de Smith-Waterman [2] es el mejor para la comparación inexacta de cadenas en el campo de la bioinformática.

Dadas dos secuencias Source ($S = S_0...S_{n-1}$) y Target ($T = T_0 \ldots T_{m-1}$), de longitudes N y M respectivamente, obtiene como resultado una valoración de lo que costaría transformar una cadena en la otra o lo que es lo mismo, lo que costaría hacer que ambas cadenas fueran iguales.

Actúa comparando todos los pares posibles formados con un elemento de S y otro de T, como indica la Figura 4. En base a la similitud de estos y a resultados de cálculos anteriores(*a*, *b*, *c*), se obtiene el resultado intermedio, *d*, para los siguientes pares a comparar. La comparación de los últimos pares, S_n y T_m, da lugar al resultado final.

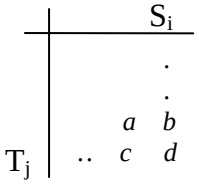


Figura 4 Comparación de pares de nucleótidos

Los resultados intermedios en el algoritmo se calculan en base a la fórmula dada en la Figura 5. Si los dos elementos a comparar son el mismo, el valor de *a* es el que se usa para obtener el resultado intermedio *d*, si no son iguales, se usa el valor de *a* más un coste asociado a la mutación del tipo sustitución. El valor del resultado *d* es el mínimo entre este valor, el valor de *b* más el coste de la inserción y el valor de *c* más el coste de la eliminación.

$$d = \min \begin{cases} \begin{cases} a & \text{si } S_i = T_j \\ a + \text{coste sustitución} & \text{si } S_i <> T_j \end{cases} \\ b + \text{coste inserción} \\ c + \text{coste eliminación} \end{cases}$$

Figura 5 Algoritmo de Smith-Waterman

El coste de las mutaciones dependerá de los requerimientos del sistema de comparación. Comúnmente se toman como valores para la inserción y eliminación 1 y para la sustitución 2.

Veamos paso a paso, la ejecución del algoritmo y cómo este hace uso de la programación dinámica.

Inicialización:

En este primer paso se crea una matriz de N +1 columnas y M +1 filas, donde M y N son las longitudes de las secuencias, y se inicializan la primera fila con los valores 0 a N y la primera columna con los valores 0 a M.

Rellenado de la matriz:

A partir de los valores iniciales y aplicando la ecuación de la Figura 5, vamos rellenando la matriz. Al final del proceso, el valor de la esquina inferior derecha es el resultado final (Figura 6).

		A	C	G
	0	**1**	**2**	3
A	**1**	**¿?**	**¿?**	¿?
T	**2**	**¿?**	**¿?**	¿?
C	3	¿?	¿?	¿?

		A	C	G
	0	**1**	**2**	3
A	**1**	**0**	**1**	2
T	**2**	**1**	**2**	3
C	3	2	1	2

Figura 6 Inicialización y rellenado

La complejidad del algoritmo es O(N * M), ya que se comparan todos lo pares posibles de las dos cadenas.

1.1.3.1 Simplificación del algoritmo

En esta sección presentamos una variante del algoritmo de Smith-Waterman propuesta en 1985 por Lipton y Lopresti [7]. Tomando como costes para las mutaciones los valores comunes citados anteriormente, *b* y *c* sólo pueden ser *a* ±1, con lo cual la ecuación de la Figura 5 queda simplificada a:

$$d = \min \begin{cases} a & \text{si } ((b \text{ ó } c = a\text{-}1) \text{ ó } (S_i = T_j) \\ a + 2 & \text{si } ((b \text{ ó } c = a\text{+}1) \text{ y } (S_i <> T_j) \end{cases}$$

Figura 7 Algoritmo simplificado

De esta manera *d* siempre será *a* ó *a* +2, pero además *b* y *c* sólo podrán tomar dos valores cada uno, *a* +1 ó *a*-1. Este hecho es interesante sobre todo a la hora de

pensar en implementación del algoritmo ya que podemos definir *b*, *c* y *d* a partir de *a* con 1 bit.

Así, para *b* y *c*, un cero significará que su valor es *a* – 1 y un uno que su valor es *a* +1. De la misma forma para *d*, un cero significará que su valor es *a* y un uno que su valor es *a* +2.

Para que este algoritmo quede más claro lo demostraremos:

Es fácil comprobar que si se cumple que *b* y *c* son *a* ±1 entonces *d* será *a* ó *a* +2. Las cuatro situaciones que se pueden dar partiendo del algoritmo de Smith-Waterman, suponiendo que el coste para la eliminación y la inserción es 1 y el de la sustitución es 2, son éstas:

a	*a+1(b)*	*a*	*a-1(b)*	*a*	*a-1(b)*	*a*	*a+1(b)*
a+1 (c)	*d*	*a-1 (c)*	*d*	*a+1 (c)*	*d*	*a-1 (c)*	*d*

Figura 8 Situaciones posibles en el algoritmo de Smith-Waterman

Si tomamos como cierto que *b* y *c* sólo pueden tomar los valores *a* ±1 tenemos:

Para la primera situación:
Si las bases a comparar son iguales:
d = min(a, a +1+1, a +1+1)=a.
Si son diferentes:
d = min(a +2, a +1+1, a +1+1)= a +2.

Para la segunda situación:
Si las bases a comparar son iguales:
d = min(a, a-1+1, a-1+1)=a.
Si son diferentes
d = min(a +2, a-1+1, a-1+1)=a.

Para la tercera situación:
Si las bases a comparar son iguales:
d = min(a, a +1+1, a-1+1)=a.
Si son diferentes:
d = min(a +2, a +1+1,a-1+1)=a.

Para la cuarta situación:
Si las bases a comparar son iguales:
d = min(a, a-1+1, a +1+1)=a.
Si son diferentes:
d = min(a +2, a-1+1, a +1+1)=a.

Ahora nos queda demostrar que si el coste de la inserción y la eliminación es uno y el de la sustitución dos, *b* y *c* sólo pueden ser *a* +1 ó *a*-1:

Lo haremos por reducción al absurdo, supongamos sin pérdida de generalidad, la siguiente situación:

	S_0	…	S_i	S_{i+1}
T_0				
.				
.				
T_j			*a*	***b***
T_{j+1}			*c*	*d*

Premisas:

1. *a* representa el mínimo coste de transformar $T_0..T_j$ en $S_0..S_i$.
2. *b* el mínimo coste de transformar $T_0..T_j$ en $S_0..S_{i+1.}$
3. c el mínimo coste de transformar $T_0..T_{j+1}$ en $S_0..S_{i.}$

Supongamos que *b > a +1* (estrictamente), si transformamos $T_0..T_j$ en $S_0..S_i$, su coste sería el de *a* .Si a esto le añadimos S_{i+1} , obtenemos $S_0..S_{i+1}$ y esto nos costaría *a* +1. Hemos transformado $T_0..T_j$ en $S_0..S_{i+1}$ con coste *a* +1, pero hemos dicho que *b > a+1* entrando en contradicción con la premisa 2.

Si por el contrario *b < a-1* (estrictamente), si transformamos $T_0..T_j$ en $S_0..S_{i+1}$ y eliminamos S_{i+1} , hemos transformado $T_0..T_j$ en $S_0..S_i$ con un coste *b +1* por la eliminación, pero como *b < a-1* entonces *b +1 < a* y entramos en contradicción con la premisa 1.

Si *b = a*, eso supondría que existe una cadena $T_0..T_j$ que se puede transformar en $S_0..S_i$, costándonos lo mismo que si la transformáramos en $S_0..S_{i+1}$, o lo que es lo mismo, se podría transformar una cadena de tamaño fijo en dos de diferentes tamaños con el mismo número de operaciones. La sustitución la podemos ver como una inserción y luego una eliminación. Y es imposible obtener dos cadenas a partir de una usando el mismo número de operaciones (inserciones, eliminaciones) que sean de diferentes tamaños. Sería lo mismo que intentar aplicar a un entero, un número fijo de operaciones (sumas y restas de uno) y que existieran dos caminos diferentes en los que obtuviéramos dos números *x, y* tal que *x = y-1*. Si siempre sumamos, en cuanto se reste nos diferenciamos del número en el que siempre hemos sumado en un valor de dos, porque restamos y porque dejamos de hacer una suma, si volvemos a restar, nos diferenciaremos en 4 y así sucesivamente.

Luego, si *b* no puede ser mayor que *a +1* ni menor que *a-1* ni *a*, *b* sólo puede ser *a-1* ó *a +1*.

Para *c* el razonamiento es el mismo. Y así queda demostrada la simplificación del algoritmo de Smith-Waterman.

El algoritmo que utilizamos en nuestro sistema es esta variante simplificada, en apartados posteriores daremos los detalles sobre su implementación.

2 Implementación Hardware

2.1 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 Device*). 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.

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 requieren de 4 a 8 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 cadenas de ADN también es susceptible del aprovechamiento de esta técnica, ya que para realizar la comparación entre dos cadenas grandes se realizan de forma iterativa cálculos entre fragmentos pequeños, de modo que sería posible trabajar con circuitos en cuya lógica estuviesen encajados los caracteres de cada fragmento a comparar, en lugar de ser éstos entradas del sistema, acelerando los caminos críticos de la lógica combinacional.

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:

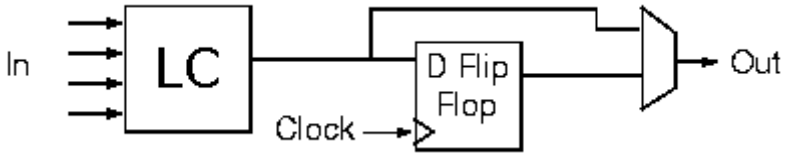


Figura 9 Estructura básica del CLB

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 [8]. El diseño de sus CLBs (ver Figura 10) está basado en tecnología SRAM, de modo que los circuitos 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.

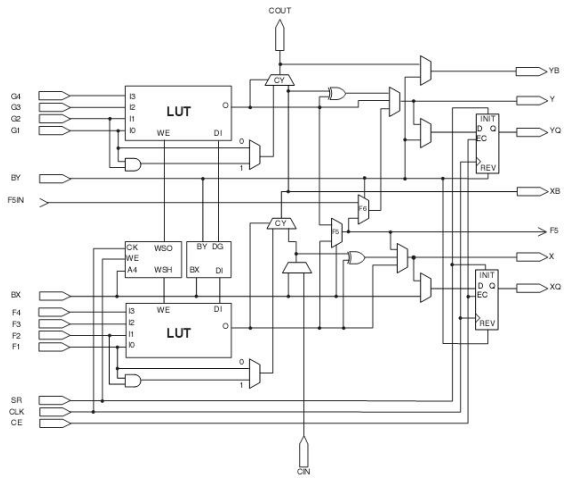


Figura 10 CLB de Xilinx con 2 LUT y 2 FF (Familia Virtex)

La segunda alternativa es Altera [9]. 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 11), adyacentes a cada CLB, que se encargan de realizar una función de conmutación convencional para confeccionar el enrutamiento del circuito.

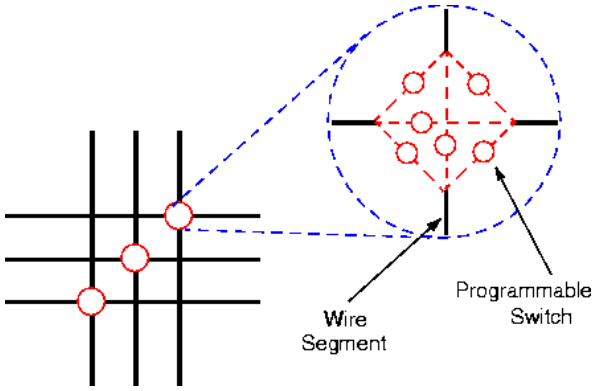


Figura 11 Estructura básica de un GRM

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 12.

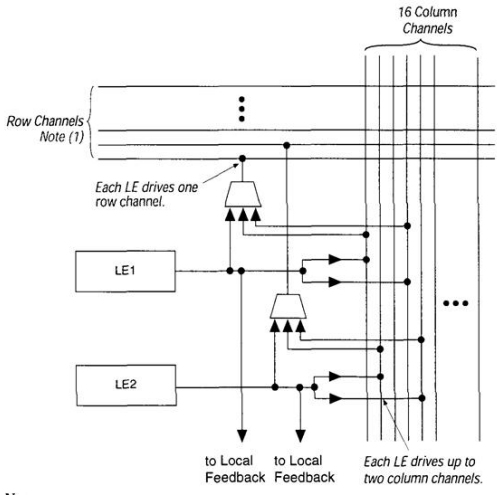


Figura 12 Dos unidades lógicas de ALTERA comunicadas por Fast-Tracking

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.

2.2 FPGAs. Descripción específica.

La FPGA empleada para realizar este proyecto (Xilinx XC2S100) es la más comúnmente utilizada en la 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 0.18 micras. Dispone de 600 CLBs, cada uno con cuatro celdas lógicas (ver Figura 13). 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. Aunque se podría aprovechar esta característica para almacenar parte de las secuencias de ADN, la escasa capacidad que se conseguiría en relación con el tamaño que tienen las cadenas reales que se compararán hace que se descarte la idea.

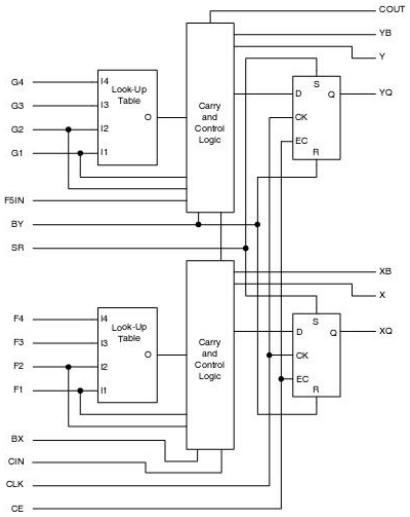


Figura 13 CLB de la familia Spartan II

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 crítico del circuito.

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 cadenas. No obstante, dicha independencia aumenta la complejidad de diseño del interfaz con la memoria, y tan sólo sería posible comparar cadenas de longitud relativamente pequeña (poco más de 5000 nucleótidos). 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 14). 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.

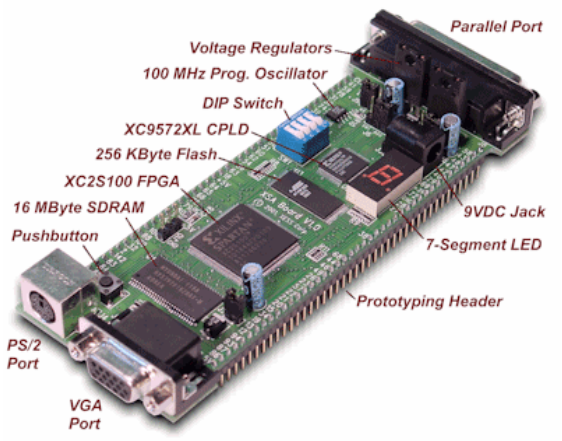


Figura 14 Placa XSA100

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 [10] 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 las secuencias de nucleótidos 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 [10].

2.3 Implementación matricial

La primera aproximación al algoritmo de Smith-Waterman es implementar directamente una matriz de n por m elementos, siendo n el número de bases nitrogenadas de la cadena Target y m el de la cadena Source. La matriz estaría compuesta por celdas que calculasen el valor de transformar la cadena q0..qi en la cadena d0..dj para el elemento i,j de la matriz.

Es decir recordemos el algoritmo, en el siguiente ejemplo:

		Qi	(Source)
		.	
		.	
	a	b	
Dj …	c	d	
(Target)			

$$d = \min \begin{cases} \begin{cases} a & \text{Si } Qi = Dj \\ a + \text{coste de una sustitución} & Qi <> Dj \end{cases} \\ b + \text{coste de una inserción} \\ c + \text{coste de una eliminación} \end{cases}$$

La implementación de una celda básica quedaría del siguiente modo:

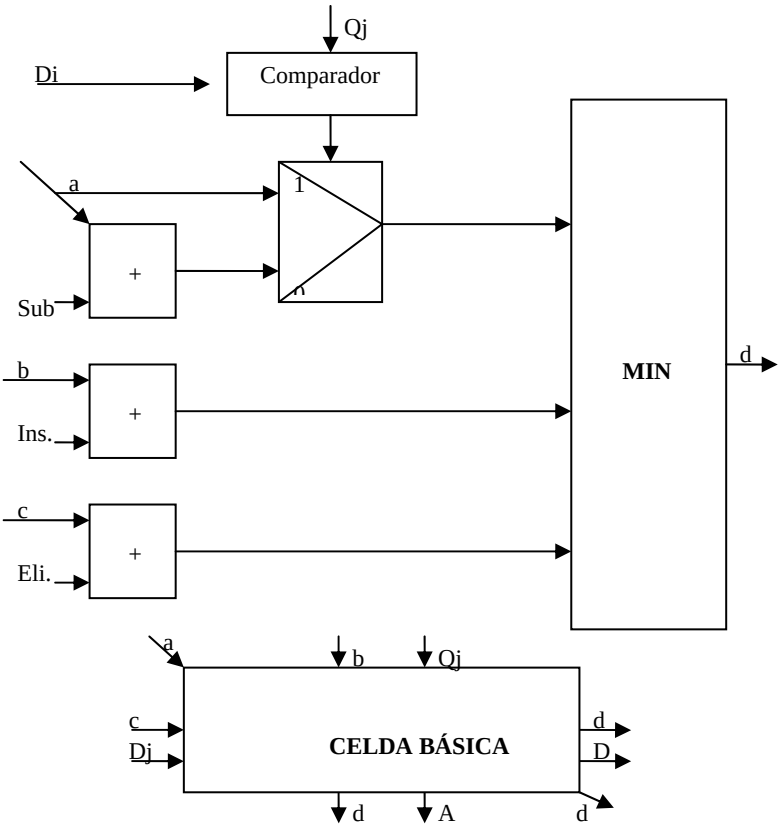


Figura 15 Celda básica

Sub: Coste de sustituir una base nitrogenada por otra.
Ins: Coste de insertar una base nitrogenada a una cadena.
Eli: Coste de eliminar una base nitrogenada de una cadena.

Por ejemplo, para calcular la similitud entre dos cadenas de 4 bases nitrogenadas cada una, tendríamos que implementar la siguiente matriz:

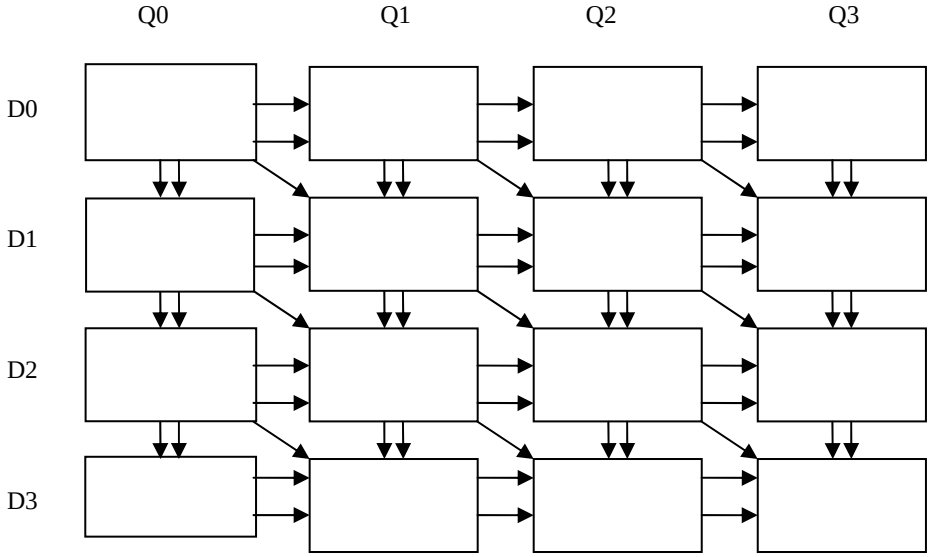


Figura 16 Matriz de celdas básicas

El resultado lo obtendríamos en la celda 3,3.

El mayor problema que existe al usar esta solución es que es dependiente del número de bases nitrogenadas de las cadenas a comparar ya que para resolver el problema debemos construir una matriz de N*M elementos (N número de elementos del Target, M número de elementos del Source) y obviamente si empezamos a comparar cadenas de cada vez mayor longitud llegará un punto en el que no tendremos hardware a nuestra disposición para implementar el anterior array. Una posible solución a este problema es implementar una matriz de un tamaño fijo por ejemplo 8x8, llamémosle celda, y diseñar una máquina de estados que vaya recorriendo por filas de a 8 y columnas de a 8 la matriz de tamaño N*M con nuestra celda con lo que en principio podríamos calcular cualquier matriz de cualquier tamaño (ya que si los elementos del Source y los del Target no son múltiplos de ocho se pueden rellenar con adeninas hasta el múltiplo de 8 más cercano a cada uno de ellos).

En realidad el problema lo hemos camuflado porque siendo verdad que no necesitamos una cantidad ilimitada de recursos hardware no sabemos cómo de grande van a ser los resultados parciales, es decir, no podemos establecer a priori cuántas entradas deben tener los sumadores o las d's que vamos calculando. Podríamos pensar que el resultado no va a ser más grande que por ejemplo 2^{17} y por tanto que las d's tendrían 17 salidas. Esto provoca que la mayor parte del tiempo estemos desaprovechando recursos hardware, pero eso es un problema menor.

Por otro lado, también hay que tener en cuenta que debemos guardar en algún lugar los resultados intermedios obtenidos por nuestro array, lo lógico es ir guardándolo en memoria, pero al igual que antes desaprovechamos la mayor parte de las veces la memoria ya que guardamos 17 bits cuando a lo mejor lo que estamos calculando en ese momento cabe en 4, supongamos que sólo guardamos las d's de la última columna de nuestra celda entonces necesitaríamos guardar en memoria 17*tamaño del Target bits lo que supone 3 palabras para palabras de 8 bits, en el caso de palabras de 16 bits la situación es peor (por tanto 3* tamaño del Target).

El desperdicio de la memoria se puede solventar utilizando una propiedad que se explica en el apartado Array Sistólico pero que viene a decir que si establecemos que el coste de una inserción, eliminación es 1 y el de una sustitución dos el algoritmo escrito en el inicio del apartado nos queda:

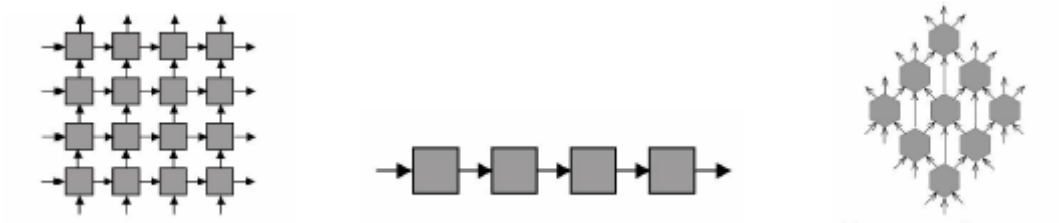
$$d = \min \begin{cases} a & \text{si } ((b \text{ ó } c = a\text{-}1) \text{ ó } (S_i = T_j) \\ a + 2 & \text{si } ((b \text{ ó } c = a\text{+}1) \text{ y } (S_i <> T_j) \end{cases}$$

Figura 17 Simplificación de Lipton y Lopresti

Pero al final lo que hacemos es construir un sistema que hace lo mismo que el array sistólico pero con muchos más recursos hardware.

2.4 Arrays sistólicos

El término de arquitectura sistólica fue utilizado por primera vez por Kung y Leiserson en 1978 [11], y se basa en la descomposición de un problema en simples elementos de procesamiento (PE) idénticos entre ellos y localmente conectados sólo con sus vecinos más cercanos. Los PE normalmente funcionan síncronamente aunque también existe la posibilidad de que funcionen asíncronamente, pero ya no serían arrays sistólicos propiamente dichos si no *wavefront arrays*. Los PEs se disponen en mallas en muchos estilos de tipologías, en la Figura 18 podemos observar algunos de ellos.



a) Matriz b) Lineal c) Hexagonal

Figura 18 Algunas topologías de arrays sistólicos

Una de las ventajas más importantes de los arrays sistólicos es el paralelismo que son capaces de aportar a un sistema, por ello nosotros aplicamos este tipo de arquitectura ya que mejora en gran medida la complejidad del algoritmo.

2.4.1 El algoritmo de Smith-Waterman utilizando arrays sistólicos

Comenzaremos recordando que para hallar el resultado de la comparación de cada par de elementos o bases de la matriz se necesitaban valores calculados anteriormente. Realizamos un dibujo (Figura 19) de la matriz de comparación con las operaciones que podrían realizarse en paralelo (líneas gruesas).

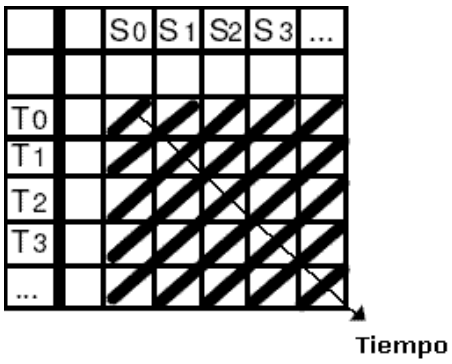
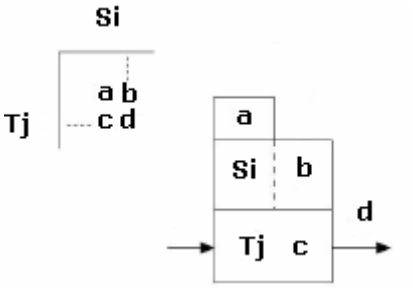


Figura 19 Cálculo de la matriz por antidiagonales

A la vista de la Figura 19, podemos observar que cada antidiagonal de la matriz del algoritmo podría realizarse en un ciclo de reloj, de aquí la extraordinaria mejora en velocidad del sistema, ya que un algoritmo O(N *M) podríamos hacerlo en O(N +M).

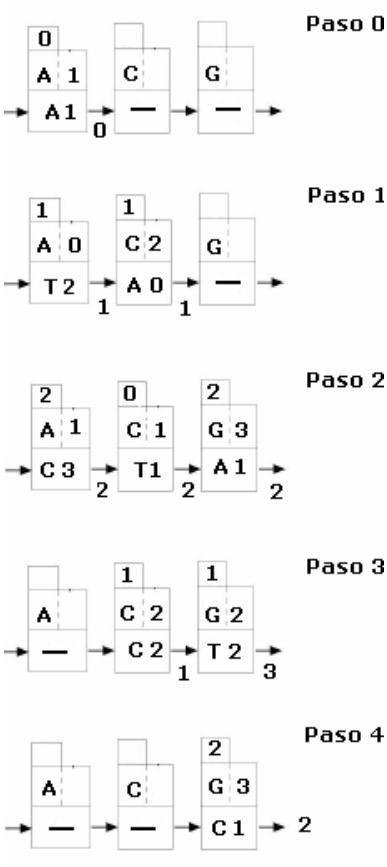
El sistema que implemente el algoritmo debe tener tantos PEs como bases tenga la cadena Source y deben estar dispuestos en una tipología lineal. Cada uno de estos PE tendrá fija su correspondiente base del Source desde la 0 hasta la N-1. En cada ciclo de reloj, una nueva base del Target es introducida en nuestro array por la primera celda e irá desplazando hacia la siguiente la base del Target que se introdujo en el ciclo anterior. De esta forma y durante N + M ciclos se van componiendo las antidiagonales.

Para ilustrarlo mejor haremos paso a paso un ejemplo, compararemos la secuencia Source = ACG con la secuencia Target = ATC. Representaremos a modo explicativo cada PE como se indica en la Figura 20, cada PE constará de su base fija, S_i, el elemento del Target a comparar, T_j, los valores anteriores *a*, *b* y *c* y tendrá como salida *d*.



		A	C	G
	0	**1**	**2**	3
A	**1**	**0**	**1**	2
T	**2**	**1**	**2**	3
C	3	2	1	2

Figura 20 Representación de cada PE y matriz de ejemplo



Para construir las antidiagonales, cada celda ejecuta la columna de su respectivo elemento del Source. En los sucesivos pasos o ciclos de reloj, va realizando las comparaciones con el elemento del Target de cada fila. Veámoslo.

Inicialmente tenemos el array sistólico vacío, formado por 3 PEs (tantos como número de elementos del Source) enlazado cada uno con su vecino, con su elemento del Source correspondiente ya fijo en cada celda.

En el primer ciclo de reloj (Paso 0), se inyecta en nuestro array el primer elemento del Target, la adenina (A), los valores de *a*, *b*, *c* los tenemos, ya que son valores iniciales predeterminados, por lo tanto, podemos calcular la *d*, que se pasa a la celda vecina.

En el Paso 1, se vuelve a introducir otra base del Target por la entrada, esto produce el desplazamiento a la celda vecina de la base introducida con anterioridad. El primer PE funciona utilizando los valores iniciales de *a* y *c*, y para *b*, el valor almacenado de su *d* obtenido en

el ciclo anterior. El segundo PE utilizará para sus cálculos datos iniciales para *a* y *b*, y para *c* el valor guardado del resultado, *d*, de su celda vecina en el ciclo anterior.

El paso 2, funciona de igual forma que los anteriores, hay que observar que en este paso se está construyendo ya la antidiagonal principal, y que ya no quedan más elementos del Target por inyectar.

En los siguientes ciclos se va vaciando el array, el Paso 4 es la última comparación de la matriz, la de más abajo a la derecha, la *d* obtenida de esta celda, es el resultado final del algoritmo. En el ciclo siguiente, el quinto, el array sistólico queda vacío por completo.

Por lo tanto el algoritmo se ejecuta en 6 ciclos, N + M.

Hemos aprovechado el paralelismo de los arrays sistólicos y su aplicación concreta para el algoritmo de Smith-Waterman para implementar una submatriz de ocho PEs o celdas básicas, que compararía pequeñas cadenas de 8 x 8 elementos, que utilizamos como componente en nuestro proyecto. Antes de explicar el diseño completo de nuestro sistema, comentaremos el funcionamiento y la ruta de datos de cada PE.

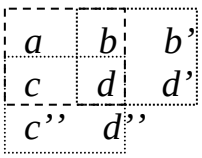
Existen diversas lecturas con diferentes proposiciones sobre la implementación de las celdas básicas, escogimos como artículo base un trabajo desarrollado en la Universidad de Honk Kong, *"A Smith-Waterman Systolic Cell"* [12], cuyo diseño y funcionamiento se detallan a continuación.

2.4.2 Smith-Waterman Systolic Cell

La implementación de esta celda utiliza la simplificación del algoritmo ya expuesta y definida por ecuaciones en la Figura 17. Recordemos que de esta simplificación se obtenía la idea de expresar *b, c* y *d* en términos de *a* con 1 sólo bit.

$$b, c = \begin{cases} 0 & \text{si } a - 1 \\ 1 & \text{si } a + 1 \end{cases} \qquad d = \begin{cases} 0 & \text{si } a \\ 1 & \text{si } a + 2 \end{cases}$$

Y recordemos también que la *d* calculada en una celda, es su b para el ciclo siguiente y la *c* para su vecina; y que la *b* de una celda es la *a*, para la vecina. Por lo tanto, esta representación nos vale mientras seamos coherentes con ella. Supongamos la siguiente situación:



Para calcular *d'* necesitamos *b*, *b'*, *d*, pero, representados a partir de *b*. Así que el valor resultado *d* que pasa la primera celda debe estar transformado en función de su propia *b*, esto es:

$$d = \begin{cases} 0 & \text{si es igual a } b - 1 \\ 1 & \text{si es igual a } b + 1 \end{cases}$$

Pero no hay que olvidar que *d* también esta representada con un bit (0 si $d = a$, 1 si $d = a + 2$), así que para calcular el valor de *d* en función de *b*, d_b, debemos fijarnos en el algoritmo anterior. Según el mismo, si:

b = a - 1 (b = 0) -> d = a (d = 0), de aquí:

$d_b = 1$ ya que $d = b + 1$.

b = a + 1 (b = 1) ->

si *d = a (d = 0)*

$d_b = 0$ ya que *d = b-1*.

si *d = a +2 (d = 1)* (por el algoritmo*: b = c = a +1*)

$d_b = 1$ ya que *d = b +1*.

Igualmente, para calcular *d''* necesitamos *c, d, c''*; todos representados a partir de la *c*. Nos fijaremos en el valor de la *d*, también podría hacerse mirando el de la *c''*, pero esto describe la misma situación anterior.

Ahora este valor se utiliza para ser consumido un ciclo después de obtenerse por la celda que lo ha calculado, pero antes hay que transformarlo. Este nuevo valor d_c (*d* representado en base a *c*) se obtiene de la siguiente manera:

c = a -1 (c = 0) -> d = a (d = 0), de aquí:

$d_c = 1$ ya que *d = c +1*.

c = a +1 (c = 1) ->

si *d = a (d = 0)*

$d_c = 0$ ya que *d = c -1*.

si *d = a +2 (d = 1)* (por el algoritmo*: b = c = a +1*)

$d_c = 1$ ya que *d = c +1*.

Además, la utilización de esta representación introduce diferencias a la hora de inicializar la matriz, ya que antes teníamos números naturales y ahora sólo un bit.

		A	C	G
	0	**1**	**2**	3
A	**1**	**¿?**	**¿?**	¿?
T	**2**	**¿?**	**¿?**	¿?
C	3	¿?	¿?	¿?

a) Número naturales

		A	C	G
	0	**1**	**1**	1
A	**1**	**¿?**	**¿?**	¿?
T	**1**	**¿?**	**¿?**	¿?
C	1	¿?	¿?	¿?

b) Bit

Figura 21 Inicialización simplificada

Si observamos la matriz de inicialización de la Figura 21a, vemos como todas las columnas de la fila 0, son uno más que su vecina en la fila y para todas las filas de la columna 0 ocurre exactamente lo mismo. Aplicando la ecuación del algoritmo simplificado a los valores naturales se construye la matriz b de la Figura 21b.

La forma de aplicar esta inicialización al array sistólico sería introduciendo, durante tantos ciclos como elementos tuviera el Target, unos por la entrada de la primera celda del array, ya que esa celda correspondería a la primera columna de la matriz. Los elementos de la primera fila corresponden a la *b* que está almacenada en cada PE, y se inicializan internamente a uno hasta que no se saca el primer resultado de una comparación por la celda correspondiente. Más adelante explicaremos esta inicialización interna con más detalle.

Figura 22 Inicialización del array por el método simplificado

Otro cambio que introduce la nueva visión del algoritmo es la forma de calcular el resultado, mientras que en la forma general del algoritmo, el resultado es directamente el valor de la celda inferior derecha, en este caso, sólo tenemos ceros y unos. La solución está en conectar la entrada U/D (up/down) de un contador a la salida del último PE del array sistólico. El contador debe tener como valor inicial el tamaño del Source, e irá restando o sumando uno en función del valor de las *d* que va generando la última celda. Si la salida es cero (*d = b -1*) restará, si es uno (*d = b +1*) sumará. Comenzará a realizar estas operaciones a partir del ciclo en el que el último PE empieza a hacer comparaciones efectivas hasta que se vacía el array, es decir, mientras se computa la última columna de la matriz. Una vez terminado, el resultado final se encuentra almacenado en el contador.

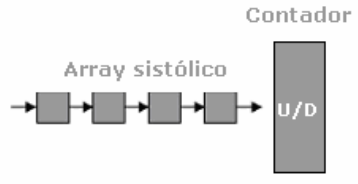


Figura 23 Solución al cálculo del resultado final

Una vez explicado todo esto, resulta más sencillo comprender el funcionamiento de la celda.

2.4.2.1 Diseño

Todos los biestables utilizados en el diseños son de tipo D y disparados por flanco de subida, con *clear* asíncrono y CE (Chip Enable).

Las cuatro bases nitrogenadas (A, G, C, T) elementos de las secuencias Source y Target de codifican con dos bits cada una.

La ruta de datos (ver Figura 24) de cada celda consta de:

- Entradas:
  - *s_in*: consta de dos bits, corresponden a S_i.
  - *t_in*: consta de dos bits que representan a T_j.
  - *data_in*: datos provenientes de la celda vecina.
  - Señales de control: *transfer, en, init_in*
  - Reloj: *clk* que se distribuye por todo el sistema.
- Salidas:
  - *s_out*: valor *s_in* propagado a través de los biestables.
  - *t_out*: valor *t_in* propagado a través de los biestables.
  - *data_out*: salida de la celda.
  - *init_out*: valor *init_in* propagado a través de los biestables.
  - *transfer*, *en*: valores propagados directamente a todas las celdas.
- Ocho biestables:
  - Dos utilizados para almacenar cada elemento del Source, S_i.
  - Dos para almacenar cada elemento del Target, T_j.
  - Un biestable para almacenar la señal de *init_in*.
  - Un biestable para guardar la variable de entrada *data_in*.
  - Un biestable para guardar la salida del multiplexor.
  - Un biestable para guardar un valor intermedio, b.
- Lógica combinacional:
  - Dos puertas XNOR y una AND.
  - Un comparador de dos bits, cuya función es comprobar si S_i y T_j son iguales.
  - Un multiplexor 2 a 1, con la entrada de datos más significativa fija a 0.

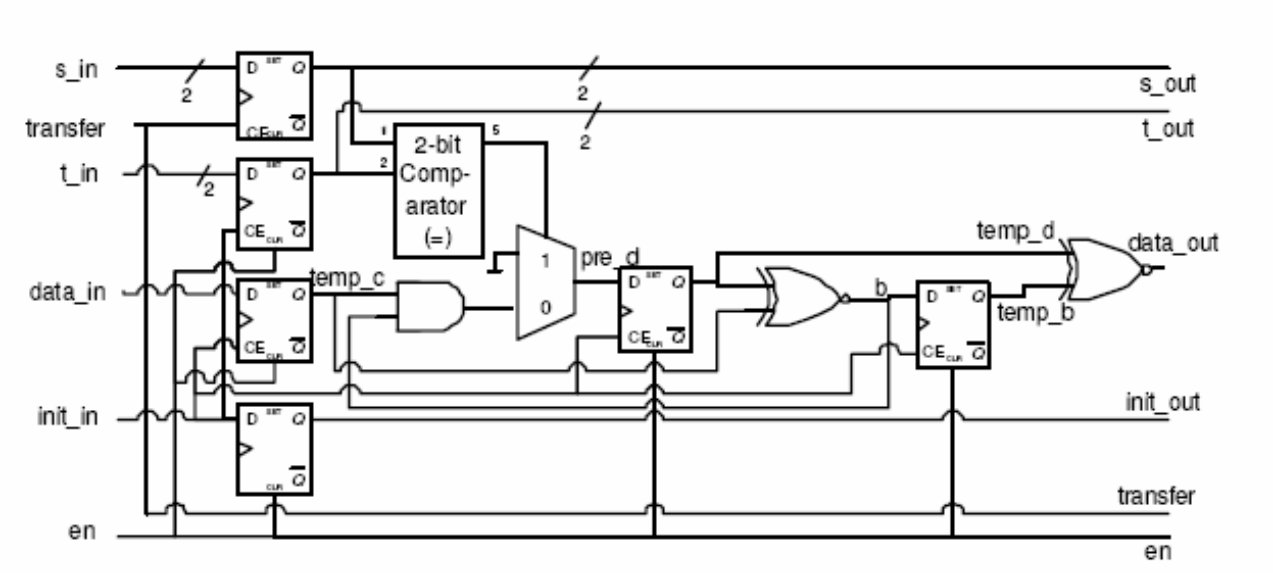


Figura 24 Elemento de procesamiento

2.4.2.2 Funcionamiento

El funcionamiento de la celda, es el siguiente:

Inicialmente se distribuyen los elementos S_i a través de todas las celdas antes de que el proceso de comparación comience. Esta distribución esta gobernada por la señal *transfer*, ya que está conectada al CE del biestable. Cuando *transfer* está en alta, la secuencia es desplazada por el conjunto de PEs.

Los elementos del Target van siendo propagados una vez ha empezado ya la comparación y mientras obtenemos la *d* de salida de la ecuación de la Figura 17.

Para calcular la *d*, como ya dijimos, necesitamos que *a*, *b* y c estén listos. En la actual implementación estos valores se obtienen de la siguiente forma:

1. *data_in*, es el valor propagado de la celda vecina y por lo tanto, *c*. Este valor es almacenado en un biestable. A la vez que lo guardamos, podemos calcular con el valor de la *d* del ciclo anterior almacenado en otro flip-flop, el valor de nuestra *b*, así: *b = temp_c XNOR temp_d*, siendo *temp_d* y *temp_c*, las salidas de los biestables.

2. El nuevo valor calculado de *b* es guardado en otro biestable. Al mismo tiempo, la salida del multiplexor se selecciona dependiendo de si $S_i = T_j$. Este valor, es decir, el resultado *d*, será 0 si son iguales ó *d = b AND temp_c*, si son diferentes. Este valor es almacenado en un flip-flop para utilizarlo en los ciclos siguientes para calcular la *b*.

3. Los valores de *b* y *d* calculados determinan la salida de la celda:
   dataout = temp_b XNOR temp_d. Este valor es conectado al *data_in* de la siguiente PE, y representa su *c*.

La señal de *init_in* debe estar a alta mientras las celdas adyacentes vayan aportando nuevos datos a la entrada de la celda que calcula una nueva *d*. Cuando está a baja los valores de los flips-flops se mantienen inalterados, ya que *init_in* va conectada a los CEs de los biestables a los que corresponden *data_in* y *t_in*.

La señal *en*, es un reset general a alta y va conectada a los *clear* asíncronos de todos los biestables.

A partir de esta celda construimos la nuestra modificando algunas partes y eliminando algunos componentes.

2.4.3 Nuestra celda básica

La primera transformación sencilla que realizamos fue introducir el Source en paralelo en el array sistólico, ya que la celda anterior los mete en serie, y se pierden, para inicialización, un número de ciclos tan grande como el tamaño del array, pudiendo usar sólo uno. Además con este cambio nos ahorramos la señal *transfer* que estaba dedicada exclusivamente a controlar la carga serie del Source.

Ahora, fijémonos cuidadosamente en el cálculo que se realiza en un ciclo de reloj en la celda ya explicada. Recordemos que para calcular el resultado *d*, necesitábamos, *a*, *b* y *c*.

En el flanco de reloj, se cargan todos los biestables de la celda, excepto los que guardan el Source que fueron inicializados antes de comenzar la comparación. Por lo tanto, un nuevo *t_in*, y un nuevo *data_in* son inyectados en la celda, asimismo también se cargan los valores que contenían *pred_d* y *b* antes del flanco, es decir los que se calcularon en el ciclo anterior.

Creemos que la celda propuesta en el artículo calcula el *data_out* de forma incorrecta, ya que lo hace con datos erróneos guardados en los biestables. Cierto es que la celda debe almacenar el valor resultado *d* (transformado en base a *b*), en cada ciclo en un biestable, porque éste será, en el ciclo siguiente el valor de su *b* para realizar los cálculos. Pero en la Figura 25 podemos observar cómo al realizar los cálculos se mezclan valores de distintos ciclos. Miremos lo que ocurre en el circuito después del multiplexor, la XNOR que calcula la *b* tiene como entradas *temp_d* y *temp_c*, pero el resultado que obtiene es incorrecto, porque el valor de *temp_d* no es *pred_d*, ése no se almacena hasta el siguiente ciclo, y *pred_d* es el que contiene nuestra *d* correcta, la de nuestro ciclo, porque es el resultado del multiplexor, que acaba de comparar las bases de entrada. Por lo que se está obteniendo *b* en función de un valor anterior e incorrecto, debería calcularse como la XNOR de la *c* y la *d* actual y no de la *c* y la *d* anterior, al ser este cálculo erróneo esto se propaga al valor calculado por la AND que entra al multiplexor. Y así la celda no calcula valores correctos.

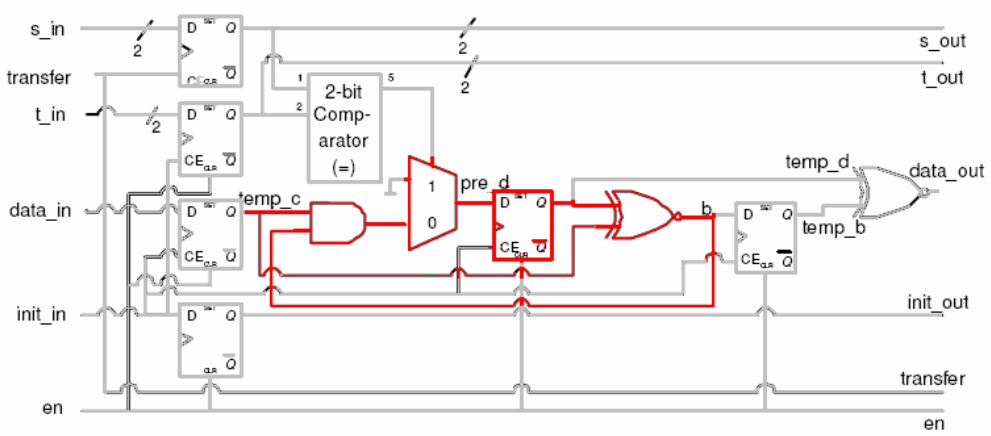


Figura 25 Zona problemática

2.4.3.1 Solución

Nosotros proponemos otra manera más sencilla e intuitiva (Figura 26). En cada flanco de reloj se carga *data_in* (*c*) y *b*, que no es más que la conversión de la *d* en base *b* del ciclo anterior, por lo tanto, directamente podemos calcular el valor de la nueva *d*, la transformación a su base *b* (para poder almacenarla en el siguiente ciclo en el biestable) y el valor del *data_out*. Para que esto funcione así, hemos eliminado el flip-flop que almacenaba *pred_d*, que era el que producía errores en la otra celda, y también introducimos otro cambio, precisamente para que se calcule todo en el ciclo con los valores adecuados. El biestable que guarda la *b*, ahora se engancha al circuito de forma diferente, su salida no va ahora a la XNOR de la que se obtiene el *data_out*, si no a la AND que entra al multiplexor. Así la celda resulta mucho más simple y correcta, ya que a la llegada del flanco todo lo que se tienen son datos "buenos" para trabajar con ellos directamente.

Ruta de datos:

- Entradas:
  - *s_in*: consta de dos bits, corresponden a S_i.
  - *t_in*: consta de dos bits que representan a T_j.
  - *data_in*: datos provenientes de la celda vecina.
  - Señales de control: *en*, *init_in*
  - Reloj: *clk* que se distribuye por todo el sistema.

- Salidas:
  - *s_out*: valor *s_in* propagado a través de los biestables.
  - *t_out*: valor *t_in* propagado a través de los biestables.
  - *data_out*: salida de la celda.
  - *init_out*: valor *init_in* propagado a través de los biestables.
- Ocho biestables:
  - Dos utilizados para almacenar cada elemento del Source, S_i.
  - Dos para almacenar cada elemento del Target, T_j.
  - Un biestable para almacenar la señal de *init_in*.

- Un biestable para guardar la variable de entrada *data_in*.
- Un biestable para guardar un valor intermedio, b.

o Lógica combinacional:
- Dos puertas XNOR y una AND.
- Un comparador de dos bits, cuya función es comprobar si S_i y T_j son iguales.
- Un multiplexor 2 a 1, con la entrada de datos más significativa a fija a 0.

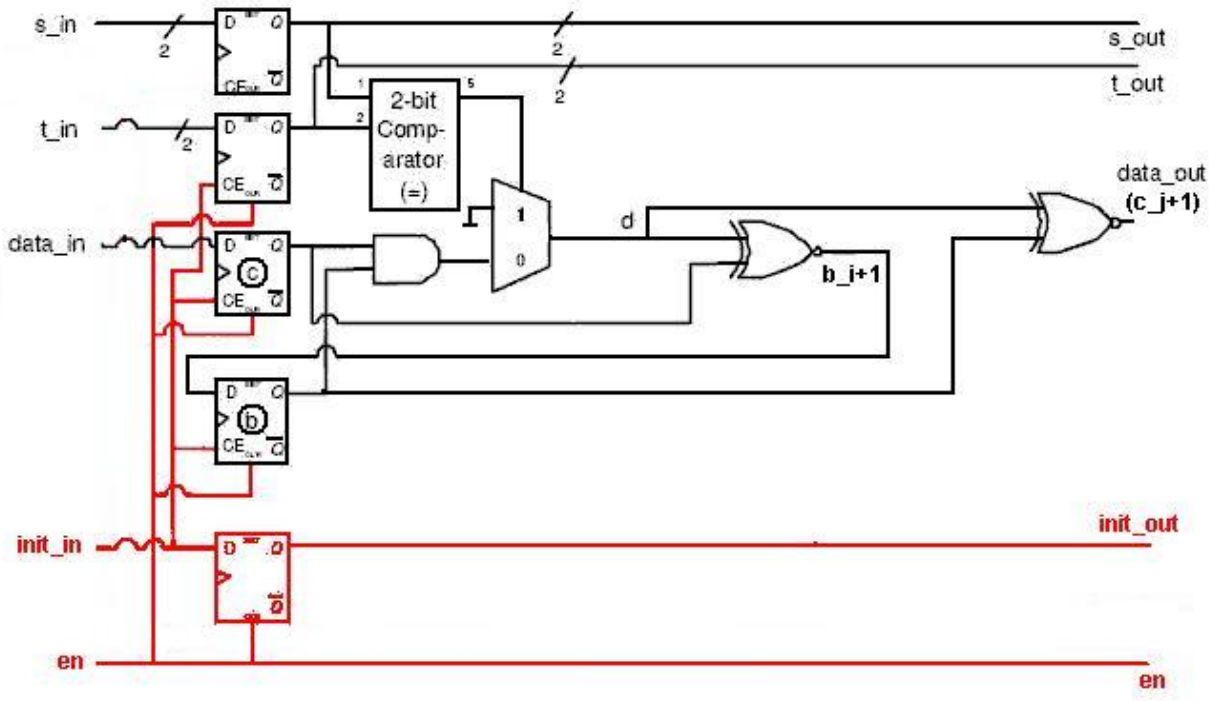


Figura 26 Nuestra celda sistólica básica (en rojo la lógica de control)

2.4.3.2 Funcionamiento

1. Tras la carga en paralelo del Source, el estado de los biestables del sistema es el siguiente (obviaremos el de la señal *init_in* por su sencillez): s_in contendrá S_i, t_in, el biestable *b* y el *c* contendrán todos cero mientras la señal de *en* esté activa. En el siguiente flanco de reloj a su desactivación, todos los biestables seguirán con el mismo valor excepto el *b*, que contendrá *d XNOR c* según nuestra implementación, y este valor será uno siempre, ya que *c* es cero y *d* es cero (o bien es *b AND c* ó cero). De esta manera, queda demostrada la inicialización interna nombrada anteriormente para la *b* que realiza una celda al computar el primer elemento de la columna de la matriz de comparación.

2. Mientras la señal de *init_in* está activa la celda aceptará sucesivamente nuevos elementos del Target y nuevos *data_in*, si la celda es primera del array recibe por su *data_in* sólo unos como ya se comentó.

Si no, recibirá el valor propagado de la celda vecina, irá generando sucesivamente valores de salida y almacenando su *b* para el ciclo siguiente hasta que *init_in* se desactive, que debe ser cuando ya no haya más elementos del Target para comparar por esa celda.

2.4.3.3 Diagrama temporal para la celda 0

A continuación presentamos un pequeño ejemplo de la ejecución de comparaciones para una celda básica. Corresponde a la columna en gris de la siguiente figura.

En el diagrama están representados el reloj (*clk*), los valores de salida de los biestables *s_in*, *t_in*, *b* y *c* (*temp_b* y *temp_c*), así como sus entradas b_{i+1} y *data_in*; el valor resultado *d* y el valor propagado a la celda siguiente *data_out.*

		A	C
	0	1	1
A	1	¿?	¿?
T	1	¿?	¿?

a) Inicio

		A	C
	0	1	1
A	1	0	¿?
T	1	1	¿?

b) Fin

Figura 27 Ejemplo de la celda 0

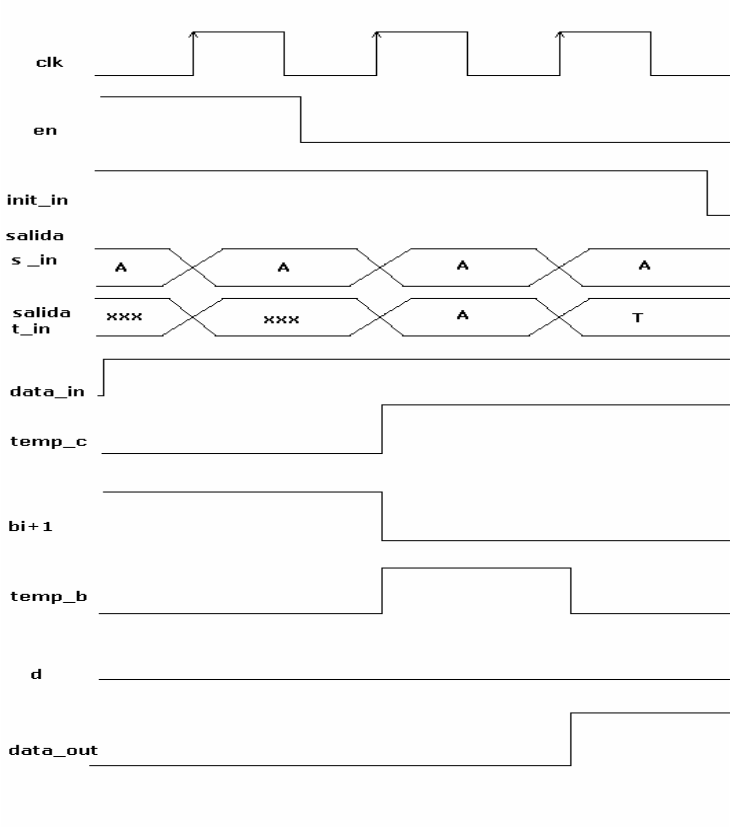


Figura 28 Diagrama temporal.

2.5 Sistema completo

El array sistólico que hemos presentado anteriormente tiene tantas celdas básicas como elementos tiene la mayor antidiagonal de la matriz que queramos calcular, el tamaño de la mayor antidiagonal dependerá del tamaño del Source y del Target que queramos calcular. Por tanto, no es útil tener que construir un array sistólico de 2xN celdas si queremos comparar cadenas de N elementos cada una.

Lo que hemos implementado es una máquina de estados que nos calcule lo mismo que el algoritmo de Smith-Waterman pero utilizando un array sistólico de 8 celdas básicas. Para simplificar hemos supuesto que el tamaño del Source y del Target (pueden ser diferentes) son múltiplos de 8.

Por otro lado, era necesario determinar dónde y cómo íbamos a guardar las cadenas a comparar, si éstas las íbamos a pasar en serie o si no iba a ser así. Lo que decidimos fue implementar el sistema suponiendo que todos los datos se encontraban disponibles en memoria (estos datos son: las cadenas a comparar (Source y Target) y sus tamaños). Por tanto podríamos dividir en sistema en dos partes diferenciadas:

- El sistema en sí que calcula el coste de transformar una cadena en otra con las restricciones que establecimos anteriormente para aumentar la eficacia del mismo.
- El sistema encargado de la comunicación con la memoria.

En este apartado nos centraremos más en el sistema completo sin adentrarnos en la comunicación con la memoria.

En líneas generales el sistema lo que hace es dividir la matriz que usaríamos en programación dinámica para resolver el problema en filas de 8 elementos y columnas de 8 elementos con lo que en cada pasada de nuestro algoritmo calculamos todos los elementos de la submatriz de 8x8 con el array sistólico.

La Figura 29 muestra un ejemplo del algoritmo que se sigue, primero se calcula la submatriz en la que nos encontramos si hemos consumido todos los elementos del Target entonces cambiamos al siguiente Source(siguiente columna de 8) y por tanto cargamos ese Source si no hemos consumido todas las bases del Target entonces cambiamos al siguiente Target (siguiente fila de 8).

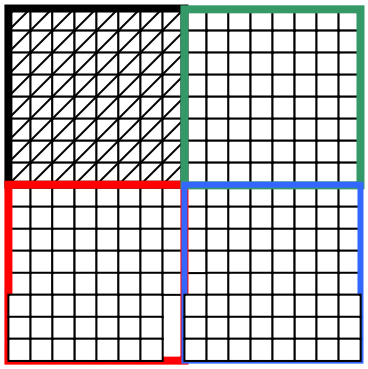


Figura 29 Matriz de 16x16

El orden de ejecución del algoritmo en este ejemplo es, primero la submatriz negra, luego la roja, la verde y por último la azul.

Para que el sistema funcione tenemos que introducir en serie sólo 8 bases del Target en el array sistólico y además tenemos que parar el array para cargar los nuevos datos que se van a usar para calcular la nueva submatriz cuando hayamos terminado con la actual. Para este cometido existe un contador denominado **contPasosArray** que nos dice cuantos ciclos de reloj lleva el array sistólico trabajando sobre esos datos.

Como sabemos que en una matriz de 8 x 8 hay 15 antidiagonales cuando **contPasosArray** haya contado 16 (un ciclo más para que le dé tiempo al up/down counter a contar) tendremos que cargar nuevos valores de Source y Target para el array sistólico además de pararlo, eso se hace gracias a la señal **init** que estará a uno mientras estemos cargando en serie los elementos del Target. Para guardar los valores sobre los que vamos a trabajar existen dos registros más **RS** y **RT,** además como el array sistólico consume elementos del Target de manera secuencial **RT** debe ser capaz de desplazar en un ciclo de reloj una base nitrogenada.

Los resultados que nos interesan salen del array sistólico cuando la señal **init_out** se pone a uno, esta señal se pone a uno en el ciclo en el que el array sistólico está calculando la antidiagonal más larga y se mantiene a uno hasta que se calcula la última antidiagonal de esa submatriz, con lo que los valores de las d's van saliendo en serie y se van guardando en un registro llamado **RDO** que nos servirá para guardar esos valores intermedios en memoria.

Para saber a qué submatriz debemos movernos o si hemos finalizado el cálculo debemos saber cuántos elementos del Source y del Target forman las cadenas a comparar, para eso existen en la memoria FIFO dos señales denominadas **NumSources** y **NumTarget** además del número de elementos del Source y del Target que hemos consumido: **ContSources** y **ContTargets** implementados mediante dos contadores.

De la ruta de datos nos quedan sólo dos elementos más por describir, el array sistólico del que ya hemos hablado anteriormente y el **Up/Down Counter,** este contador realmente pertenece al array sistólico pero como nosotros queremos calcular la similitud entre las cadena Source y Target completas y no sobre subcadenas suyas, está fuera del array sistólico, se inicializa con el valor del número de elementos del Source que hay (**NumSources)** y sólo contará cuando se esté calculando la última columna de 8 de la matriz total.

En el ejemplo anterior el contador podrá contar cuando se esté calculando la submatriz verde o la azul y en ambos casos desde que se haya terminado de calcular la antidiagonal más larga y hasta que se termine de calcular el último elemento de la submatriz. Para controlar el comportamiento del contador existe una señal denominada **fin** que se pone a uno cuando se esté calculando una submatriz de la última columna de 8 de la matriz completa.

En la siguiente figura mostramos la ruta de datos del sistema completo:

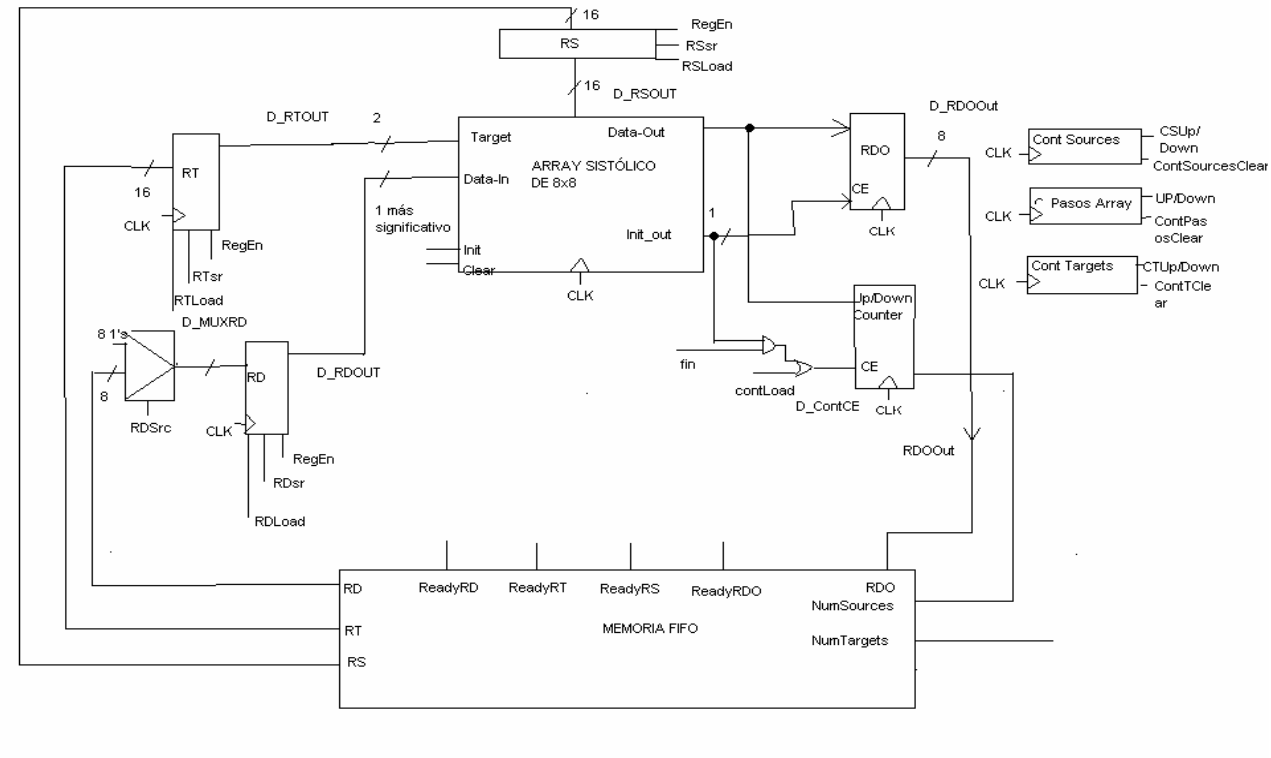


Figura 30 Ruta de datos del sistema

Nota: como el array sistólico es de 8x8 bases nitrogenadas y una base nitrogenada son 2 bits (Adenina, Guanina, Citosina, Timina) tanto RS, RT son registros de 16 bits.

La memoria fifo tiene como misión que el sistema no espere o espere lo mínimo imprescindible a que se acceda a memoria para cargar el nuevo valor de RS, RT, RD o guardar RDO. Por tanto el valor de las señales RD, RT, RS de la memoria FIFO es el siguiente dato que el sistema debe usar para calcular la similitud entre las cadenas. Para poder leer un RD, ReadyRD debe estar a 1. Para RT y RS ocurre lo mismo y cuando se lee un data_in, target o un source el Ready correspondiente se pone a cero.

El controlador (Figura 31) de este sistema tiene los siguientes estados:

S0: Esperamos a que NumSources esté listo (ReadyNumSources =1)

Reset cont. Sources (contNumSources cuenta el número de sources consumidos/8)

Reset cont. Targets (cuenta el número de targets consumidos)

Clr (array sistólico)

S1: Esperamos a que NumTargets esté listo (ReadyNumTargets =1)

Cargamos en el counter (en el contador del array sistólico UP/DOWN Counter) num. Sources (esta inicialización se explica en la sección Array Sistólico)

S2: Cargamos Rs si ReadyRS=1, si no esperamos hasta que lo sea (Rs es el registro donde se van guardando los sources que en esa vuelta del sistema se usarán como input)

Sumamos uno al contNumSources

S3: Cargamos Rt si ReadyRT es uno, sino esperamos hasta que lo sea (Rt es el registro donde se van guardando los targets que se usarán durante el cálculo de esa vuelta)

S4: Cargamos Rd (Rd es el registro donde se encuentran los resultados parciales de la columna anterior)

Si cont_sources=0 -> MuxRD=0 esto significa que cargaremos en RD ceros (esto se explica en la sección Array Sistólico)

Sino esperamos a que ReadyRD esté listo para cargar RD y lo cargamos

Cargamos source en el array sistólico

Reset contador pasos del array

S5: Si (contSource*8=numSource) entonces fin<=1(la señal fin, le indica al contador del array sistólico que puede empezar a contar cuando le llegue un uno del init_out, contará hacia arriba o hacia abajo dependiendo del valor de data_out).

Si cont_pasos_array<8 entonces INIT=1 esto implica que no se han cargado todos los targets

Cont_targets++

Cont_pasos_array++

Desplazar dos bases nitrogenadas en RT (registro de targets)

Si no

INIT<=0

Si ReadyRDO es uno entonces

guardar RDO en memoria

sino

esperar hasta que lo sea y luego guardarlo.

Si cont_pasos_array= fin(para nosotros 16) entonces ir al estado 6

Sino cont_pasos_array++ y volver al estado 5

S6: Si (cont_numTarget=numTargets)

Si (cont_numSource*8=numSource)

INIT<=0 ir al estado de fin

Sino

Clear del array sistólico, clear del contador de pasos del array sistólico

Ir al estado 2

Sino clear contador pasos array sistólico

Ir al estado 3

Fin: Pararlo todo y mantener el resultado en el contador

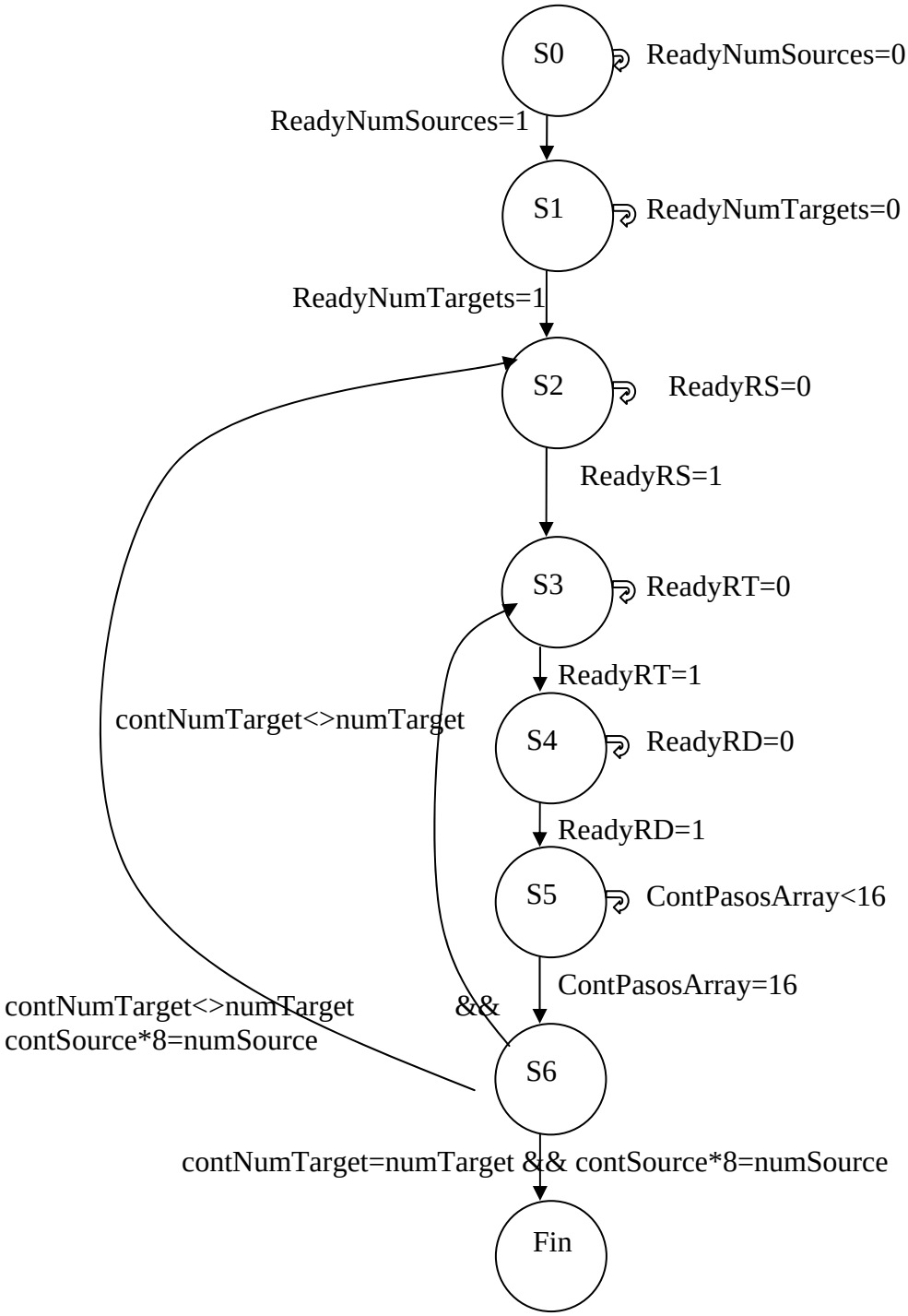


Figura 31 Estados del controlador del sistema

2.6 Comunicación con memoria

Antes de entrar en detalle en la descripción del interfaz de memoria implementado conviene explicar la forma en la que se realizan las peticiones de lectura y escritura. El sistema de memoria será empleado para almacenar las cadenas de ADN de entrada, así como para escribir los resultados intermedios generados durante el proceso de cálculo. Adicionalmente, se reservan dos posiciones de memoria para almacenar las longitudes de las cadenas Source y Target.

En la Figura 32 se muestra la matriz global dividida en cuadrados que representan las submatrices calculadas en cada iteración de la matriz sistólica. En cuanto a la interacción con la memoria se pueden distinguir seis casos, cada uno identificado con un color distinto.

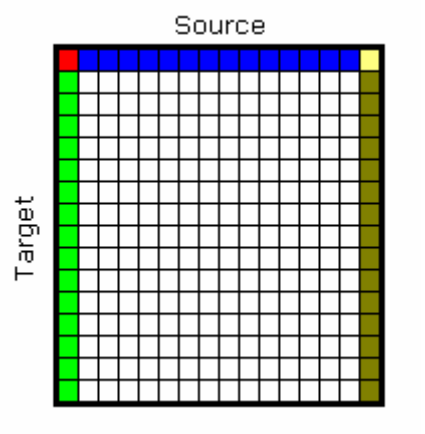


Figura 32 Distinción de submatrices según el acceso a memoria

La columna verde corresponde a submatrices que sólo precisan acceder a un elemento de la cadena Target, ya que aún no se han generado resultados intermedios y la subcadena de Source ya está almacenada en el array sistólico. En este caso también se necesita escribir en memoria los resultados del cálculo.

Al calcular la primera submatriz (en rojo) se realizan todas las operaciones posibles con memoria, con la excepción de la lectura de los resultados generados en la submatriz adyacente de la izquierda, ya que no existe.

Durante el cálculo de las submatrices coloreadas en azul, se realizan las cuatro operaciones posibles: se lee un fragmento de Source, otro de Target, otro de los resultados generados en la submatriz de la columna anterior y se escribe el fragmento generado en la etapa actual.

Las submatrices blancas constituyen la mayor parte del cálculo del sistema, sobre todo para cadenas de tamaño medio o grande. En este caso ya se dispone del fragmento de Source necesario. Todas las demás operaciones han de realizarse.

La última columna coloreada en amarillo claro y oscuro es equivalente a las columnas de su izquierda, con la salvedad de que al tratarse de la última, no es preciso almacenar los resultados generados en memoria, sino que éstos son utilizados para controlar el proceso de cuenta del contador que contiene el resultado final. Excepcionalmente, para poder mostrar dicho resultado al usuario, en la última submatriz se escribe el resultado final en memoria, aunque esto no sería preciso si se mostrase la salida del contador mediante otro dispositivo.

Dicho esto, es posible concluir ciertos aspectos que determinarán el diseño del interfaz de memoria del sistema. En primer lugar, se puede asegurar que siempre que se realice una lectura de la cadena Source, se realizará también una lectura de Target, por lo que en el diagrama de estados del controlador del interfaz debería haber una transición directa desde el estado que lea la subcadena de Source y el que lea la subcadena de Target.

Por otro lado, en la mayoría de los casos las operaciones con memoria son la lectura de un fragmento de Target y la lectura y escritura de resultados intermedios, así que sería deseable optimizar el comportamiento del interfaz para estas operaciones. Para ello sería preciso realizar lecturas paralelas de las posiciones de memoria correspondientes a dichos fragmentos para así minimizar esperas generadas por la latencia de la RAM, pero la placa empleada tiene un bus de datos de 16 bits y no permite trabajar de forma independiente con los módulos de la memoria. Por ello la mejor solución consiste en dividir el rango de direcciones en distintas zonas para cada tipo de datos (Source, Target, datos intermedios) y alternar las operaciones con memoria para cada zona.

Además, y sabiendo que el cálculo de la matriz se realiza por columnas, se sabe que la cadena Target necesita estar disponible durante el proceso completo de cálculo, por lo que una zona de la memoria se ha de emplear exclusivamente para esta cadena. Ocurre lo mismo con la cadena Source que, a pesar de poder ser eliminada según va consumiéndose, sólo libera una posición de memoria cada columna completa. En cambio, los resultados intermedios sólo requieren el tamaño de memoria correspondiente a una columna de la matriz, ya que cuando se consume una subcadena, el resultado del cálculo puede sobrescribir la posición que acaba de ser leída. Así pues, para un sistema de memoria de un solo bloque como es el caso, la memoria queda dividida en 3 regiones independientes, como indica la Figura 33, dos para las cadenas de entrada y otra para los resultados intermedios (*data*).

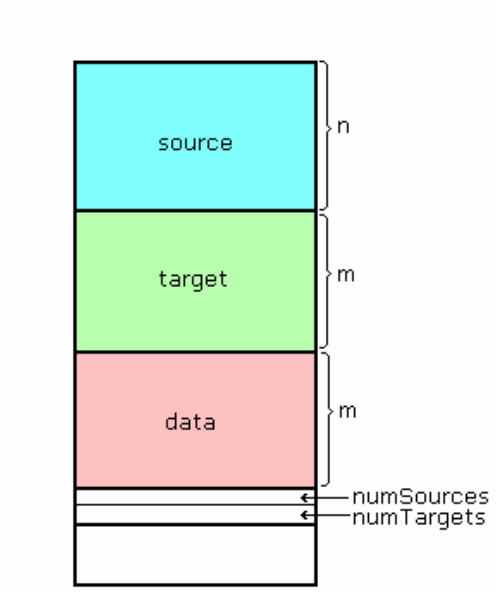


Figura 33 División del rango de direcciones de memoria

Adicionalmente, se reservan dos posiciones de memoria para almacenar al inicio de la ejecución las longitudes de las cadenas de entrada, que serán suministradas por el usuario junto con las cadenas en sí. La razón de situar estas dos posiciones después de las tres regiones anteriormente explicadas es la reducción de lógica combinacional necesaria para el cálculo de direcciones: de este modo, se necesita un sumador menos para calcular las direcciones de lectura y escritura, ya que, por ejemplo, para calcular la posición del *i*-ésimo elemento de la cadena Target hay que sumar la dirección base de la

cadena con *i*. Si las longitudes de las cadenas se almacenasen en las posiciones más bajas de memoria habría que incluir otro sumador que sumase 2 a la dirección anterior.

Una vez consideradas las limitaciones del sistema de memoria y conocida la forma en la que se accederá al mismo, se ha optado por diseñar el interfaz de memoria según el paradigma del productor/consumidor, incluyendo tres búferes (Source, Target y data) de lectura y otro de escritura. Por simplicidad, el tamaño de éstos es de un solo elemento, aunque se podría sacar partido a la operación en modo ráfaga de la memoria SDRAM y aumentarlo.

El interfaz (Figura 34) se comunicará con el resto del sistema mediante un *flag* asociado a cada búfer que indicará la disponibilidad de éstos. De igual modo, el propio interfaz deberá consultar estos *flags* para escribir el contenido de los búferes en memoria o realizar las lecturas pertinentes y actualizarlos, según sea el caso.

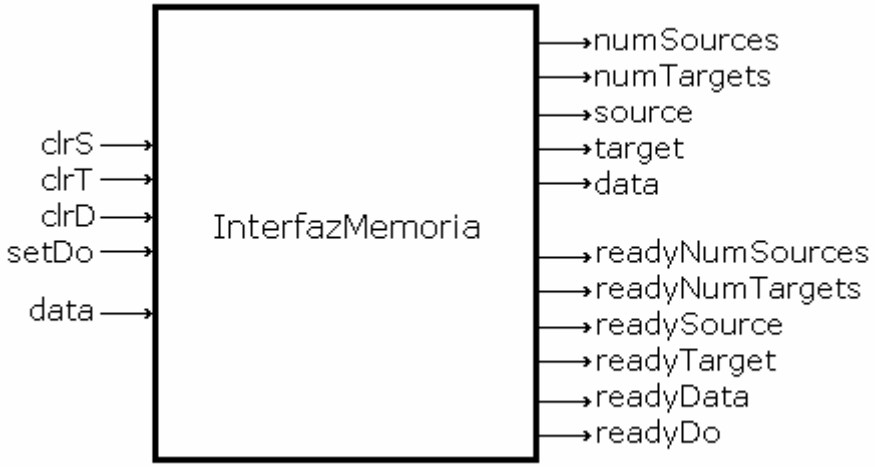


Figura 34 Bloque simplificado del interfaz de memoria

El interfaz dispone de salidas independientes para cada tipo de dato, para que sea posible leer varios elementos en un mismo ciclo, cuando esto sea posible. El estado de los *flags* se puede consultar mediante las señales de *ready*, que están activas cuando los búferes están llenos, es decir, listos para ser consumidos por el sistema en el caso de los de lectura, o vacío, es decir, listo para ser escrito en el caso del búfer de escritura. Cuando el sistema consuma el elemento de alguno de los búferes de lectura, debiendo haber consultado previamente su *flag*, deberá poner éste a 0 mediante las señales de *clear* que figuran en el dibujo. De igual forma, al escribir un elemento en el búfer de escritura, se deberá activar la señal `setDo` que actúa simultáneamente como señal de carga del búfer y del *flag* de escritura.

Para facilitar el proceso de diseño, se ha empleado un controlador de SDRAM suministrado por Xess (Figura 35), que permite acceder a la memoria de un modo sencillo, sin necesidad de gestionar el proceso de refresco ni controlar los tiempos de *setup* y *hold* impuestos por el chip de memoria. Este controlador específico para la placa empleada hace uso de los DLLs descritos anteriormente para sincronizar la memoria y dispone de una señal de lock que se activa cuando se ha completado este proceso. Además, dispone de buses separados para la lectura y la escritura, por lo que no es necesario incluir buses triestado que controlen el acceso al bus común.

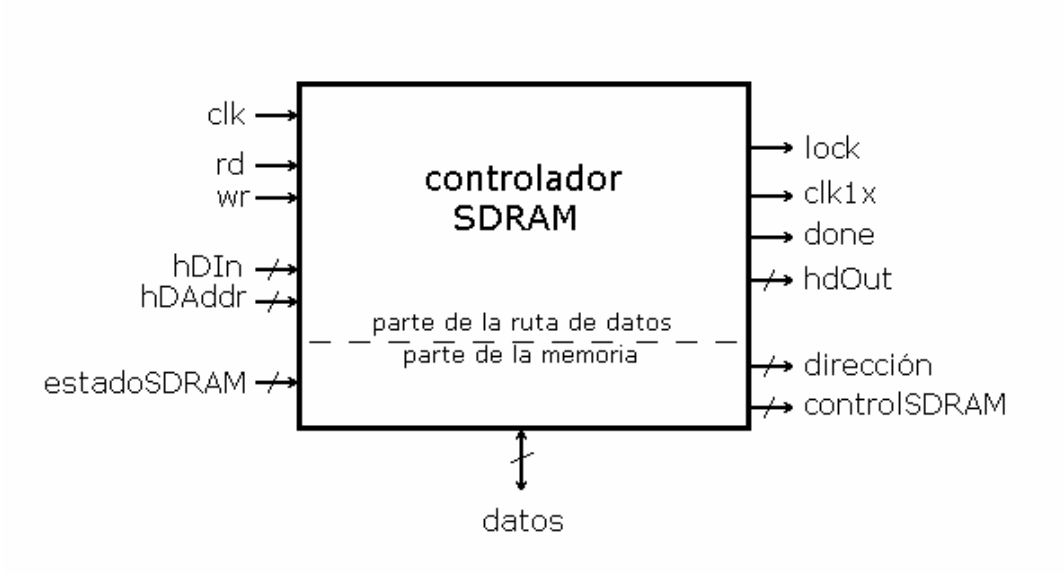


Figura 35 Controlador de SDRAM suministrado por Xess

La señal `clk1x` proviene del DLL y es la señal de reloj sincronizada, por lo que todo el resto del sistema deberá usar esta señal de reloj, en lugar de la que proviene del oscilador de la placa, que tan sólo se suministra al controlador para su sincronización. El modo de uso del controlador es de tipo SRAM, de modo que hay dos señales que indican qué operación se quiere realizar (lectura, `rd` o escritura, `wr`) y una señal que activa el controlador cuando la operación solicitada se ha completado (`done`). Las señales `hDIn`, `hDout` y `hDAddr` corresponden a los buses de escritura, lectura y direcciones, respectivamente.

A continuación se muestran en detalle los distintos bloques principales del interfaz de memoria implementado. En primer lugar, en la Figura 36 se detalla la estructura de los flags junto con los búferes de lectura y escritura. Se puede observar que la entrada de *clear* del *flag* de lectura de los datos intermedios está gobernada por dos señales. Una proviene del sistema, y se activa cuando se consume un elemento, y la otra proviene del controlador del interfaz de memoria, y se activa tras el cálculo de la primera columna, por motivos de consistencia de datos (recordar que en la primera columna no se realizan lecturas de data, por lo que hay que esperar al final de la misma para poner el *flag* de lectura de data a 0 y evitar lecturas de datos inexistentes). Las salidas de los búferes de lectura son salidas del interfaz, mientras que la salida del búfer de escritura se conecta a memoria.

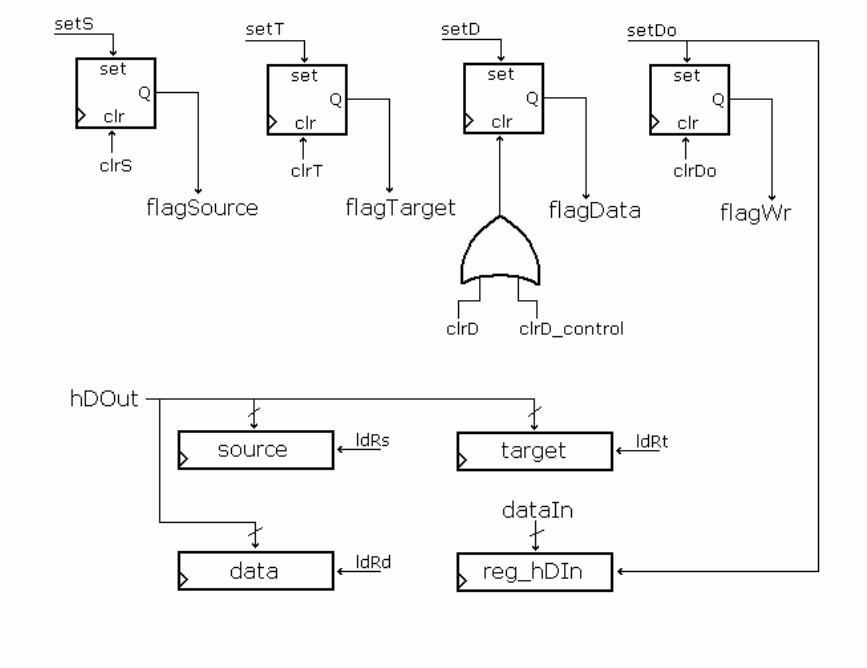


Figura 36 Búferes y flags del interfaz de memoria

El elemento crítico en lo que respecta a la velocidad de cálculo del interfaz de memoria es el que agrupa los registros que contienen los punteros de lectura y escritura de los búferes (Figura 37). Se trata de 4 contadores que indican la posición de memoria en la que se realizará la siguiente operación en cada caso. Los contadores se ponen a 0 cuando alcanzan el límite de su zona de memoria correspondiente. En el caso de Source, el puntero vuelve a la posición 0 cuando se han leído todas las posiciones de la cadena, equivalentemente, cuando se alcanza el valor almacenado en `numSources` (longitud de la cadena Source). Realmente, esto no debería ocurrir nunca, ya que se trata de una cadena que sólo se lee una vez, pero evita posibles accesos a la zona reservada para la cadena Target. Los otros punteros funcionan de igual modo, aunque lógicamente leen el valor de `numTargets`, que es el que contiene la longitud de sus zonas correspondientes de memoria. Los sumadores calculan la dirección efectiva de memoria sumando la dirección base de cada zona con el valor de cada puntero. En la implementación actual, el sistema sólo es capaz de funcionar a 50 MHz, lo cual es debido al camino crítico que atraviesa estos sumadores.

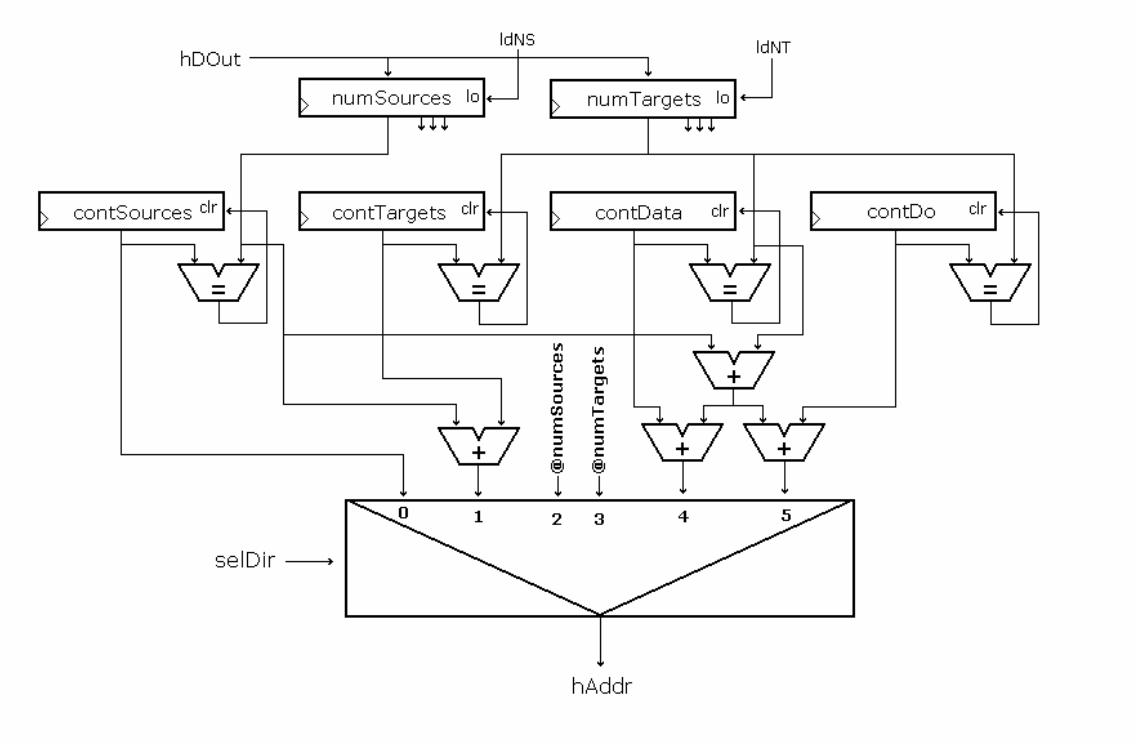


Figura 37 Punteros de lectura y escritura del interfaz de memoria

El controlador del interfaz, que se muestra en la Figura 38, se encarga de realizar las operaciones de memoria pendientes (consultando los *flags*) y actualizar los valores de los *flags* debidamente (señales de *clear* y *set*). Además, es el que se ocupa de gestionar el controlador de SDRAM de Xess (`rd, wr, done, lock`). Tiene una salida que gobierna el multiplexor de direcciones, `selDir`. También se ocupa de cargar los búferes (señales de *load*).

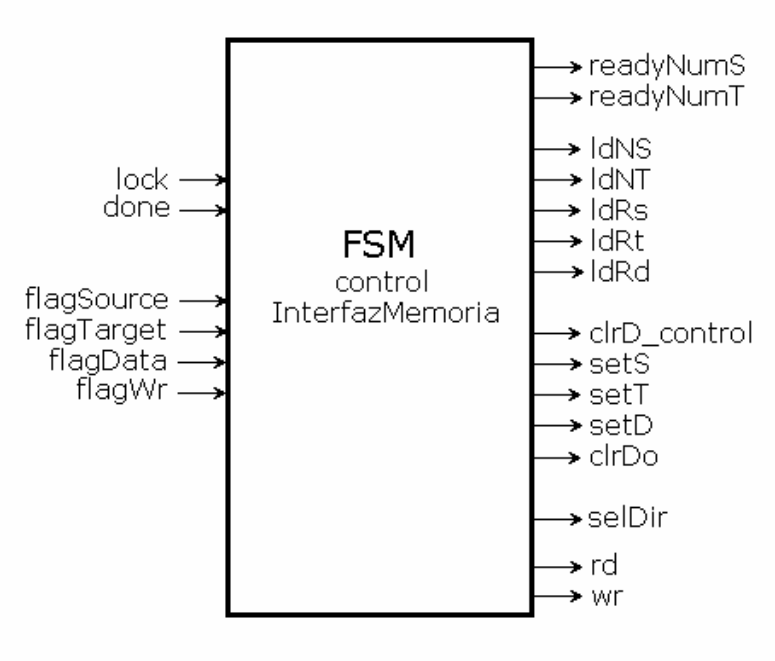


Figura 38 Controlador del interfaz de memoria

El diagrama de estados (Figura 39) es bastante simple e intuitivo en cuanto a los estados de que consta, aunque tiene un gran número de transiciones. Los tres primeros estados corresponden a la inicialización del sistema (lectura de `numSources` y `numTargets` tras sincronizar la memoria). Después, el controlador pasa a leer de memoria el primer elemento de Source, ya que será necesario para la primera submatriz. El resto de transiciones responden siempre al mismo patrón: si hay un búfer que requiere una operación de memoria, se pasa al estado correspondiente. Si todos están actualizados, se pasa al estado de espera, en el que no se accede a memoria. Hay transiciones que por simplicidad en la función de cambio de estado no se han incluido, por ejemplo, la que va del estado `leerData` a `leerTarget`. La razón es que se supone que siempre que se realiza una lectura de data, ya se habrá leído el elemento de Target de la misma submatriz. No obstante, si esto no fuera así, tras volver al estado de espera, se actualizaría el búfer con la penalización de un ciclo de reloj.

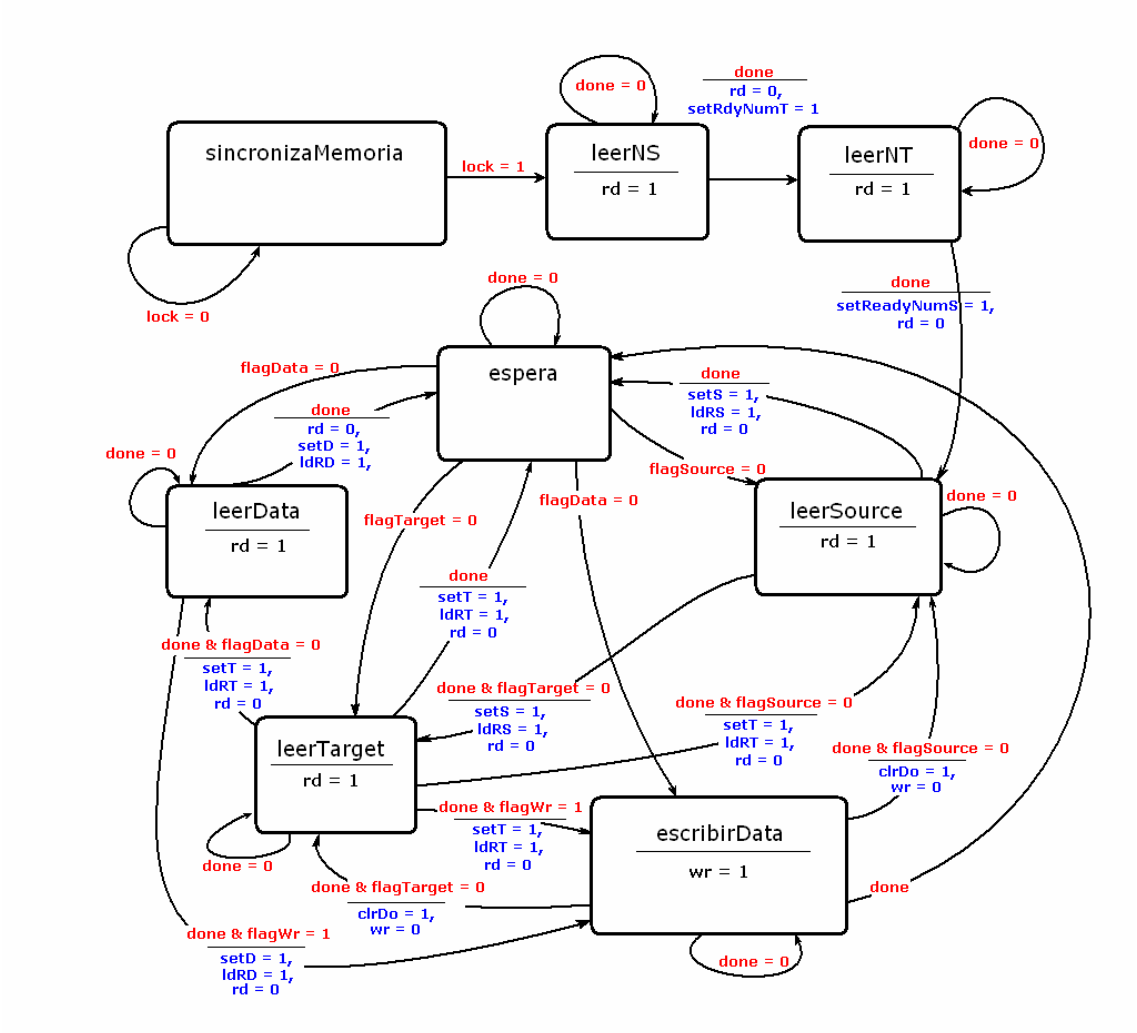


Figura 39 Diagrama de estados del controlador del interfaz de memoria

2.6.1 Descripción de la aplicación de carga de datos

Dado que para cargar las cadenas a comparar necesitamos transformarlas a formato .xes (uno de los formatos que entiende la utilidad GSXLOAD) y éstas son

de un gran tamaño, hemos implementado una pequeña aplicación que transforma sources y targets formadas por los elementos A, T, G, C, U a este formato.

En la siguiente figura mostramos la visión general de nuestra aplicación:

Figura 40 Aplicación para cargar cadenas

La aplicación permite editar los dos cuadros de texto que se observan en la figura anterior además de cargar ficheros de texto tanto para el Source como para el Target. Para cargar desde un fichero un Source, el usuario tendrá que acceder al menú **Archivo**, seleccionar la opción de **Abrir Source** y navegar por el árbol de directorios hasta que encuentre el fichero deseado, para el Target la situación es similar. Para generar el fichero de interfaz .xes una vez introducidos el Source y el Target se deberá pulsar la opción **Guardar Como** del menú **Archivo** y seleccionar un nombre de fichero.

El aspecto del fichero generado será el siguiente:

```
- 10 0000 00 00 a8 00 00 00 00 00 00 00 00 00 00 00 00 00
- 10 0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- 10 0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- 10 0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Cada una de las líneas del fichero consta por ese orden de: un guión, el número de bytes que se van a cargar en esa línea, la dirección base donde se van a guardar

los datos que contiene esa línea y finalmente los bytes que deseamos guardar en memoria.

2.7 Programas utilizados

En el desarrollo de este proyecto hemos utilizado para la parte hardware fundamentalmente el programa Xilinx ISE 7.1 y las XSTOOLS.

Hablaremos brevemente de en qué consiste cada una de ellas.

2.7.1 XSTOOLS

Son unas pequeñas aplicaciones que proporciona la Xess Corp. que ofrecen una interfaz de comunicación sencilla entre el PC y la FPGA al usuario. 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 41) y accionando el botón TEST, se hace ejecutar un pequeño programa que adecuado para cada placa, comprueba todos sus componentes. La placa sacará por uno de sus leds el resultado del test, si todo funciona correctamente mostrará O, mientras que si la prueba falla mostrará E y una ventana con los posibles errores.

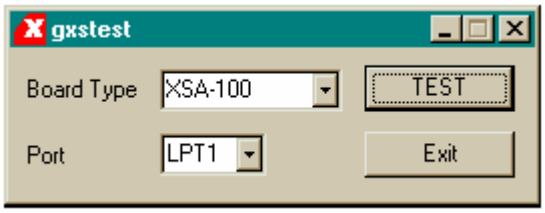


Figura 41 Interfaz GXSTEST

- 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 la utilidad es muy sencillo, simplemente consiste en introducir en la ventana de programa (Figura 42) el factor por el que queremos dividir el reloj, el tipo de placa y el puerto paralelo.

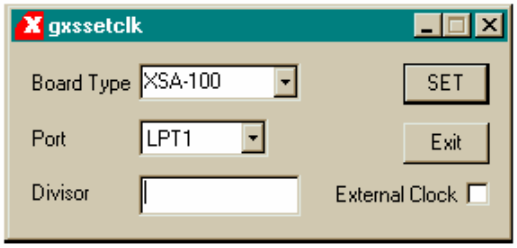


Figura 42 Interfaz GXSSETCLK

- 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 43), seleccionar el tipo de placa y el puerto paralelo y dar a LOAD.

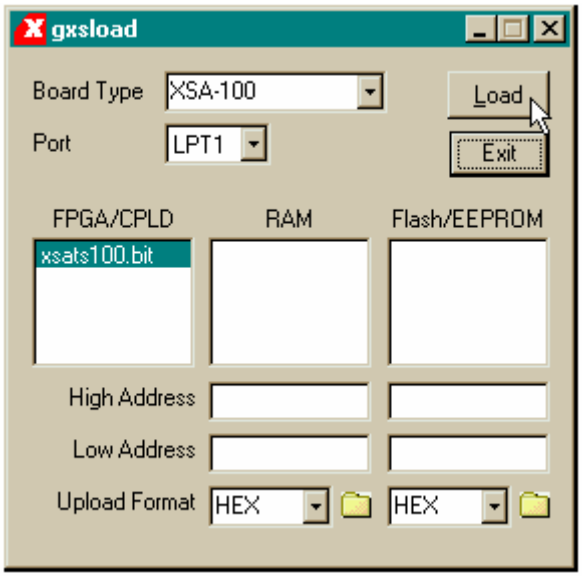


Figura 43 Interfaz GXSLOAD

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 que queremos obtener, 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.

2.7.2 XILINX ISE 7.1

El Integrated Software Environment (ISE™) 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 de 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.

3 Escalabilidad

3.1 Posibles mejoras del sistema

La mejora más evidente del sistema es la de aumentar el número de bits de NumSources y NumTargets de 16 bits a 32 para poder calcular cadenas de casi cualquier tamaño. Esta mejora sólo conlleva además de aumentar esos registros modificar la memoria FIFO para que cuando cargue estos datos en su buffer cargue para cada uno de ellos 32 bits, dos palabras.

Como hemos comentado antes en la descripción del array sistólico, su principal característica es que si aumentamos el número de celdas sistólicas para calcular una submatriz de mayor tamaño seguimos manteniendo que en cada ciclo de reloj (independientemente del tiempo de ciclo, siempre que el tiempo de ciclo sea lo suficientemente alto para que termine su trabajo una celda básica) el array va a calcular una antidiagonal completa. Esto implica que (suponiendo que el coste de acceder a los datos es nulo, por abstraernos de ese problema) cuantas más celdas contenga el array sistólico más vamos a mejorar el rendimiento del sistema completo porque vamos a calcular con un coste en tiempo lineal una submatriz cada vez más grande.

Por ejemplo en una matriz 8x8 hay 15 antidiagonales, luego un array sistólico de 8 celdas con lo que hemos dicho antes tardará 16 ciclos en calcularnos la similitud de las cadenas (uno más que antidiagonales porque le tienen que ir llegando los datos al UP/DOWN counter). Ahora supongamos una matriz de 16x16 y un array sistólico de 16 celdas, esa matriz tiene 31 antidiagonales luego nuestro nuevo array sistólico tardará 32 ciclos en darnos un resultado.

Si hubiéramos tenido que comparar un target y un source de 16 bases nitrogenadas cada uno con un array sistólico de 8 celdas hubiéramos tenido que aplicarlo sobre cada una de las submatrices de 8x8 que existen, como hay 4 y suponiendo que el sistema es ideal y no se pierden ciclos en verificar a que submatriz hay que ir sino que eso se sabe tardaríamos en ese caso hipotético 16x4=64 ciclos mientras que con un array sistólico de 16 celdas lo hubiéramos calculado en 32 ciclos.

Esta mejora de más del doble de velocidad tiene como contrapartida el aumentar el hardware que compone el array sistólico hasta conseguir el doble de celdas en total, pero además implica en el sistema completo un impacto de mayor importancia que es la comunicación con memoria.

La memoria que hemos utilizado realiza sus intercambios mediante palabras de 16 bits una de las razones por las cuales decidimos que el array sistólico tuviera 8 celdas, por tanto que calculara submatrices de 8x8, fue porque eso implicaba introducir 8 bases nitrogenadas en cada vuelta del bucle, como una base nitrogenada son 2 bits, tenemos 16 bits = 1 palabra. Los intercambios con memoria los hacemos directamente mediante palabras.

Si se aumenta el tamaño del array sistólico hay que aumentar los registros RS, RT pero también hay que modificar la memoria FIFO para que cargue en vez de una palabra, dos. Eso implica aumentar el tráfico de datos con la memoria pero al haber implementado la memoria FIFO como una máquina de estados independiente del sistema, mientras éste está realizando sus cálculos nuestra memoria FIFO está cargando los siguientes elementos del Source , Target y d’s en su búfer, esto amortigua el acceso a memoria.

Pero si nos fijamos, la memoria actual de las FPGAs suele ser SDRAM, la principal característica de este tipo de memoria es que además de que se sincroniza con la señal de reloj tarda varios ciclos en darnos el primer dato de un bloque contiguo que necesitemos pero las siguientes palabras las proporciona de ciclo en ciclo.

Los siguientes gráficos pueden ilustrar el acceso a memoria tanto para la lectura de un bloque como para la escritura de un bloque:

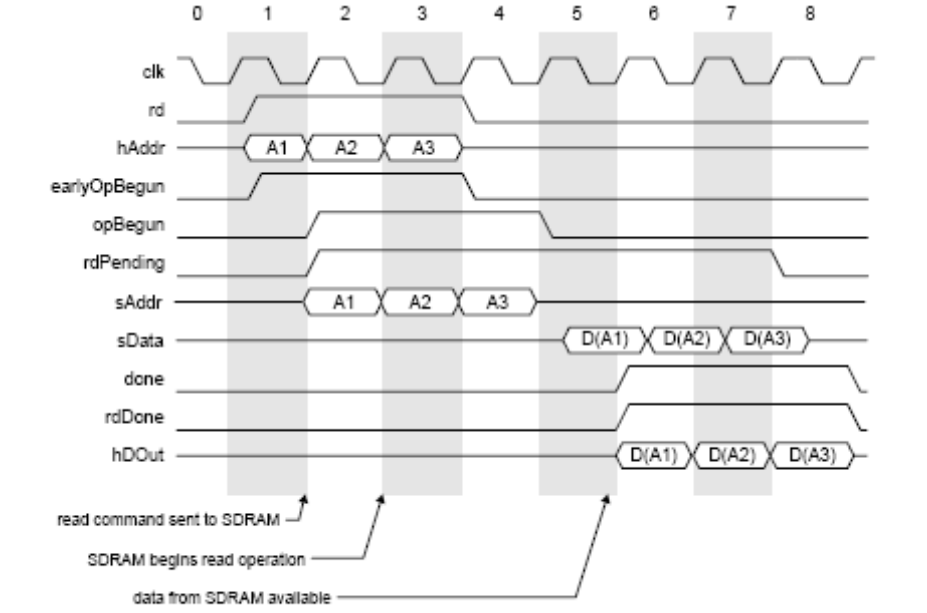


Figure 4: SDRAM controller pipelined read operations.

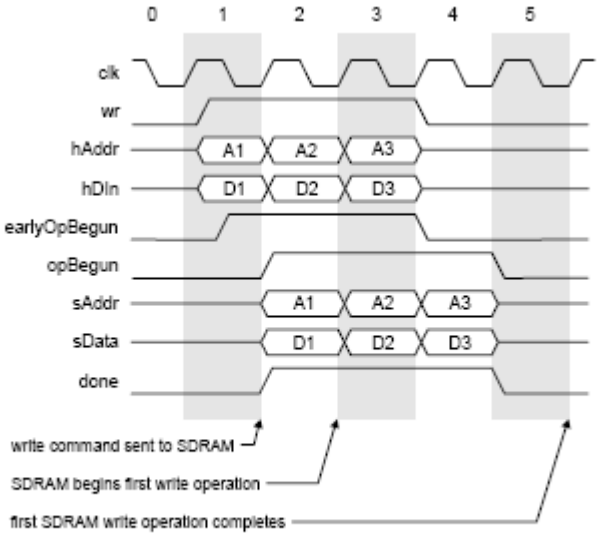


Figure 5: SDRAM controller pipelined write operations.

Figura 44 Acceso segmentado a la memoria

Con lo que en el peor de los casos si tenemos que cargar 32 bits para los elementos del Source, 32 para los del Target y 16 bits para las d's tanto como para acceder a ellas como guardarlas. Nos costaría según el gráfico anterior 8+8+7+2= 25 ciclos mientras que empleamos para el cálculo de la submatriz 32 sin tener en cuenta los estados de control del sistema.

3.2 Mejoras del sistema de memoria

En cuanto al modo de comunicación con la memoria, las posibles optimizaciones del sistema son las siguientes:

- Simplificar el cálculo de direcciones de cada zona (Source, Target y data) mediante el uso de varios bloques de memoria independientes.

- Segmentar la operación del cálculo de direcciones para así aumentar la frecuencia máxima de funcionamiento del circuito.

- Implementar un sistema de memoria caché por medio de la SRAM integrada en la FPGA.

- Emplear el sistema de máscaras de la SDRAM para escribir dos fragmentos de data en cada posición de memoria (actualmente, por simplicidad, se emplea una posición de memoria de 16 bits para almacenar los 8 bits de data generados en cada submatriz).

- Dividir la memoria en 3 zonas con dirección base fija para simplificar el cálculo de direcciones. Supondría una penalización en cuanto al aprovechamiento de espacio de la RAM.

- Aumentar el tamaño de los búferes de lectura y escritura y realizar las operaciones de memoria en ráfagas, disminuyendo la latencia de la SDRAM.

- Hacer uso de otros elementos de almacenamiento disponibles en la placa. Por ejemplo, la memoria flash o un chip de SRAM montado en la placa de expansión XSA-Board 2.0 de Xess.

- Implementar un módulo que se encargara de guardar en la memoria de la FPGA las cadenas a comparar y sus tamaños, comunicándose mediante el puerto serie o con cualquier otro puerto con un ordenador que utilizara nuestro sistema para comparar gran cantidad de datos de manera eficiente.

REFERENCIAS

- [1]: Base biológica: fai.unne.edu.ar/biologia/macromoleculas/adn.htm
- [2]: Smith T.F. & Waterman, M.S. (1981). Identification of common molecular subsequences. *Journal of Molecular, Biology*, 147, 195, 197.
- [3]: Needleman S. B., Wunsch C. D. (1970): A general method applicable to search for similarities in amino-acid sequence of two proteins. Journal of Molecular Biology, Vol. 48. pp 443-453.
- [4]: FASTA: http://fasta.genome.gp/
- [5]: BLAST: http://www.ncbi.nlm.nih.gov/Education/BLASTinfo/
- [6]: Pearson WR, Lipman DJ K (1990): Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods Enzymol.183:63-98.*
- [7]: RJ Lipton, DP Lopresti (1985): A Systolic Array for Rapid String Comparison
- [8]: XILINX: http://www.xilinx.com/
- [9]: Altera: http://www.altera.com/
- [10]: Xess: http://www.xess.com/
- [11]: H. Kung, Charles Leiserson (1978): Systolic arrays (for VLSI), in Sparse Matrix Proceedings
- [12]: C.W. Yu, K.H. Kwong, K.H. Lee and P.H.W. Leong : *A Smith-Waterman Systolic Cell*, http://www.cse.cuhk.edu.hk/~phwl/mt/public/archives/papers/sw_fpl03book.pdf