DISEÑO DE UN CONTROLADOR DE MEMORIAS DRAM EN VHDL

Ing. Daniel Francisco Gómez Prado
dgomez6027@yahoo.com

Facultad de Ingeniería Electrónica de la Universidad Nacional Mayor de San Marcos, Lima, Perú

Resumen : La actualización de las (*DRAM*) memorias dinámicas es uno de los temas menos entendidos por los diseñadores electrónicos, debido en parte, a la diversidad de métodos y formas existentes para llevarla a cabo. Existen dos formas de realizar la actualización, en forma distribuida y en forma de ráfaga (*burst*); ambas pueden ser llevadas a cabo mediante diferentes métodos: con una actualización de solo RAS (*Row Address Strobe*), actualización de CAS (*Column Address Strobe*) antes de RAS y actualización tipo oculto. Además hay que tener en cuenta que para poder reducir las dimensiones del encapsulado las memorias dinámicas presentan un *bus* de direcciones multiplexado, por lo cual también se requiere que el controlador realice la decodificación de sus direcciones.

Este artículo presenta la teoría general de los controladores de memoria dinámicas, las máquinas de estados, los diagramas de tiempo y las ecuaciones usadas en el PLD (Dispositivo Lógico Programable) e implementadas en VHDL (*Very High Speed Integrated Circuits*, VHSIC *Hardware Description Language*) para probar el prototipo diseñado.

Abstract : DRAM refresh is one of the topic misunderstood by electronic designers due to the many ways refresh can be accomplished. There are two means of performing refresh, distributed and burst; and both can be accomplished by various ways: ROR (Ras Only Refresh), CBR (Cas Before Ras) and hidden refresh. Besides, it must be taken into count that in order to reduce the dimension of DRAM packages, the address bus in the DRAMs have been multiplexed; thus, it is also required that the DRAM controller performs address decoding. This paper contains general DRAM controller theory, state machine definitions, timing diagrams and the PLD equations implemented in VHDL used to test the prototype designed.

Palabras Claves : DRAM, Actualización, RAS, CAS, ROR, CBR, Ráfaga.

I. FUNDAMENTO TEORICO

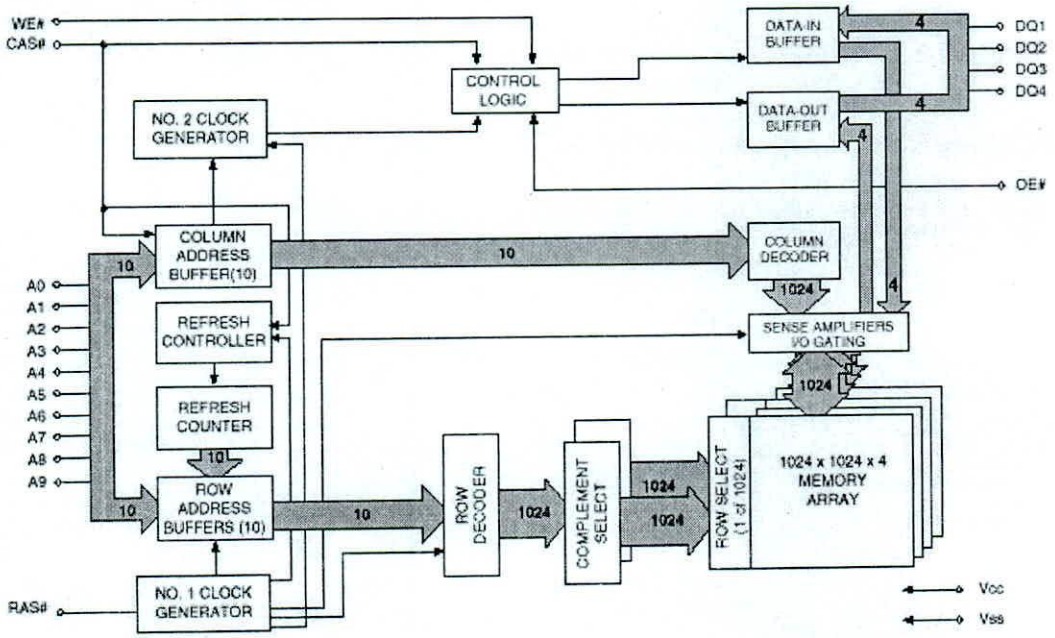
DRAM es un acrónimo de "*Dynamic Ramdon Acces Memory*" (Memoria dinámica de acceso aleatorio). El termino dinámico indica que para que la memoria mantenga los datos requiere que estos sean actualizados dentro de un cierto periodo de tiempo, así cuando estas memorias son desenergizadas pierden los datos que guardan. El termino acceso aleatorio indica que cada celda en la memoria puede ser leída o escrita en cualquier orden.

Los datos en una memoria DRAM están organizados en celdas de memoria donde cada celda contiene un número especifico de bits, por ejemplo una DRAM de 1Mx4 bit tiene 4 bits por celda, donde cada celda de

memoria tiene una única dirección determinada por una dirección de fila y una dirección de columna. Debido a que las celdas de memoria están compuestas por transistores y condensadores integrados, y la carga en dichos condensadores solo pueden ser mantenidos durante unos pocos milisegundos, las DRAM deben ser actualizadas periódicamente. El transistor integrado funciona como un conmutador que puede ser utilizado para controlar el flujo de corriente entre el corte y saturación. En una DRAM cada transistor conmuta un único bit, si el transistor esta saturado y la corriente puede fluir, entonces es un 1 lógico, si está en corte, entonces es un cero lógico. El condensador se utiliza para mantener la carga, pero esta carga, como ya se ha dicho, se pierde rápidamente, perdiendo el dato. Para solucionar este problema se tiene que añadir un circuíto de actualización que lea el valor antes de que desaparezca completamente la carga y vuelva a escribir una versión nueva de ella. La velocidad de actualización esta expresada en nanosegundos y es la figura representativa de la velocidad de la DRAM y depende del diseño de la celda de memoria y de la tecnología utilizada en su fabricación.

Las DRAM son fabricadas utilizando un proceso similar al empleado en la fabricación de microprocesadores, es decir, un substrato de silicio es insolado con patrones que forman los transistores y condensadores que almacenan cada bit. Su fabricación no es complicada debido a que esta formado por una serie de estructuras simples y repetitivas que no requieren complejidad de fabricación; y es más barata que una SRAM "*Static* RAM" (Memoria Estática no volátil) porque utiliza la mitad de transistores. En la actualidad cada celda de memoria además de los transistores y condensadores para almacenar los bits de información, tienen circuitos de soporte que en general incluye, ver Fig. 1:

- Amplificadores para la señal o caga detectada en una celda de memoria.
- Direcciones lógicas para seleccionar filas y columnas.
- Selección de la dirección de la fila [*] (**RAS#**) y selección de la dirección de columna (**CAS#**) para escoger la celda de memoria deseada. Estas señales de selección también sirven para iniciar y terminar las operaciones de lectura y escritura.
- Circuito para escribir y leer la información almacenada en las celdas de memoria.
- Habilitador lógico de salida para prevenir que los datos aparezcan en las salidas a menos que sea específicamente deseado.
- Contadores o registros internos para mantener una secuencia de actualización, utilizados en la actualización de **CAS** antes de **RAS**.



[*] El símbolo # significa activo bajo, así, **RAS#** = $\overline{\textbf{RAS}}$

Figura 1. Diagrama de bloques de una memoria DRAM

[Protopapas, 1992] Para evitar que la información contenida en las celdas de memorias se pierdan es necesario: Leer la tensión de cada celda; amplificarla y cargar el condensador otra vez con el voltaje original

A este proceso que debe ser repetido periódicamente se denomina actualización. Las **DRAM** permiten actualizar todas las celdas de una fila en una sola operación. Esta necesidad de actualizar las memorias se debe a que [Lilen, 1993] la carga se degrada después de 5 a 10 milisegundos, por lo cual se la debe actualizar periódicamente cada 2 milisegundos como medida de seguridad. En la actualidad estos tiempos son mucho mayores y varían con la memoria utilizada, ya que dependen de la tecnología y del método de fabricación utilizado en la construcción de las celdas de memorias.

En la Fig. 1 las celdas de las **DRAM** están organizadas en una matriz cuadrada de 1024x1024, en general estos arreglos son rectangulares, esto quiere decir que el número de filas y columnas no es necesariamente el mismo; aunque siempre que se selecciona una fila, se puede decir, que se selecciona una página; la cual tiene un ancho igual al número de columnas del arreglo.

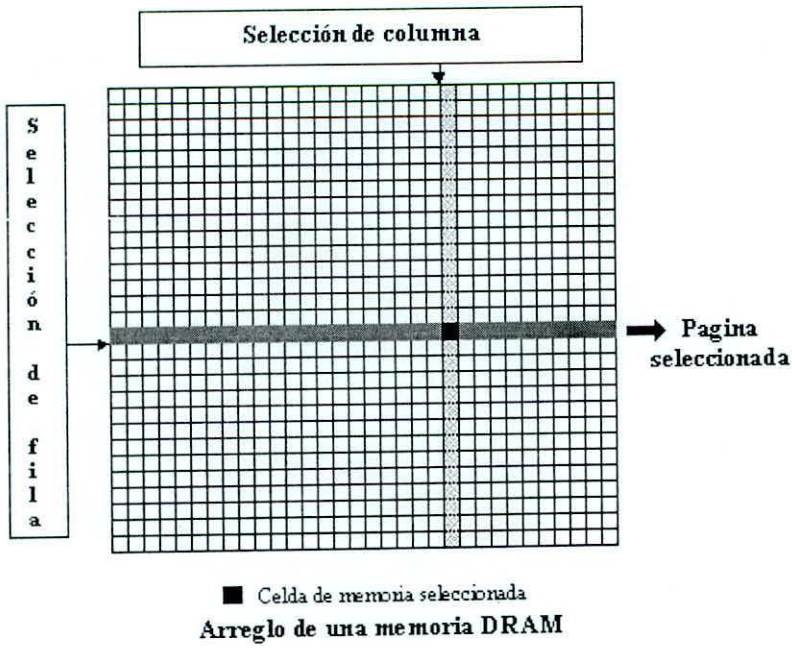


Figura 2. Arreglo de una memoria en celdas

Esta geometría de la matriz es importante para determinar el número de ciclos requeridos para actualizar completamente una **DRAM** debido a que seleccionando una página se puede actualizar por completo todas sus columnas en un único ciclo de lectura, por lo que, el número de ciclos de actualización necesarios para actualizar completamente una **DRAM** esta determinado por el número de paginas que esta tenga o lo que es igual al número de filas de la matriz.

Además, debido a que el *bus* de direcciones de las **DRAMs** están multiplexados, una memoria de 1Mx4 bits por ejemplo, no tendrá 20 pines para direccionar todas las celdas de memoria sino que presentara solamente 10 pines. Así, para lograr la selección de una celda de memoria se tiene que proveer los 10 primeros bits para seleccionar una fila; para lo cual se coloca a nivel lógico cero el pin de selección de filas **RAS#**, una vez seleccionada la fila o página de trabajo, se selecciona la columna deseada con los 10 siguientes bits. Para la selección de columna es necesario activar el pin de selección de la columna, **CAS#** a nivel lógico cero. [Barry, 1995] De esta manera la celda de memoria seleccionada corresponde a la intersección de la fila y columna

proporcionada como se muestra en la Fig. 2. Por lo tanto en nuestro ejemplo dicha celda contendrá 4 bits por tratarse de una memoria de 1Mx4 bits, es decir, primero se colocan en los 10 pines de direcciones de la memoria las direcciones de fila (A_0-A_9), y luego son almacenados en un registro interno de filas cuando la señal **RAS#** pasa a nivel lógico cero, luego las siguientes direcciones A_{10}-A_{19} se colocan en los mismos 10 pines de dirección y son almacenados en un registro interno de columnas cuando la señal **CAS#** se envía a nivel lógico cero; como se presenta en el diagrama de tiempos correspondiente al ciclo de lectura de una memoria **EDO DRAM** en la Fig. 3.

El acrónimo **EDO** viene de "*Extended Data Ouput*" y significa que las salidas en el *bus* de datos permanecerán validas (estarán latcheadas) hasta que el siguiente ciclo de lectura ó escritura utilice el *bus* de datos; esta permanecía de los datos en el *bus* **DQ**, ver Fig. 3, se proporciona para que el microprocesador no tenga que estar esperando a que la memoria **DRAM** proporcione los datos, sino que vaya ejecutando otras instrucciones sabiendo que cuando requiera los datos buscados estos le estarán esperando en el *bus* **DQ**. Este tipo de memoria **EDO** es una mejora de las memorias **FPM** (*Fast Page Mode*) en las cuales, el dato solo permanecía durante un tiempo en el *bus* **DQ** y luego pasaba a alta impedancia; teniendo que estar el microprocesador siempre esperando a la memoria, generando ciclos muertos de reloj y degradando el rendimiento del sistema.

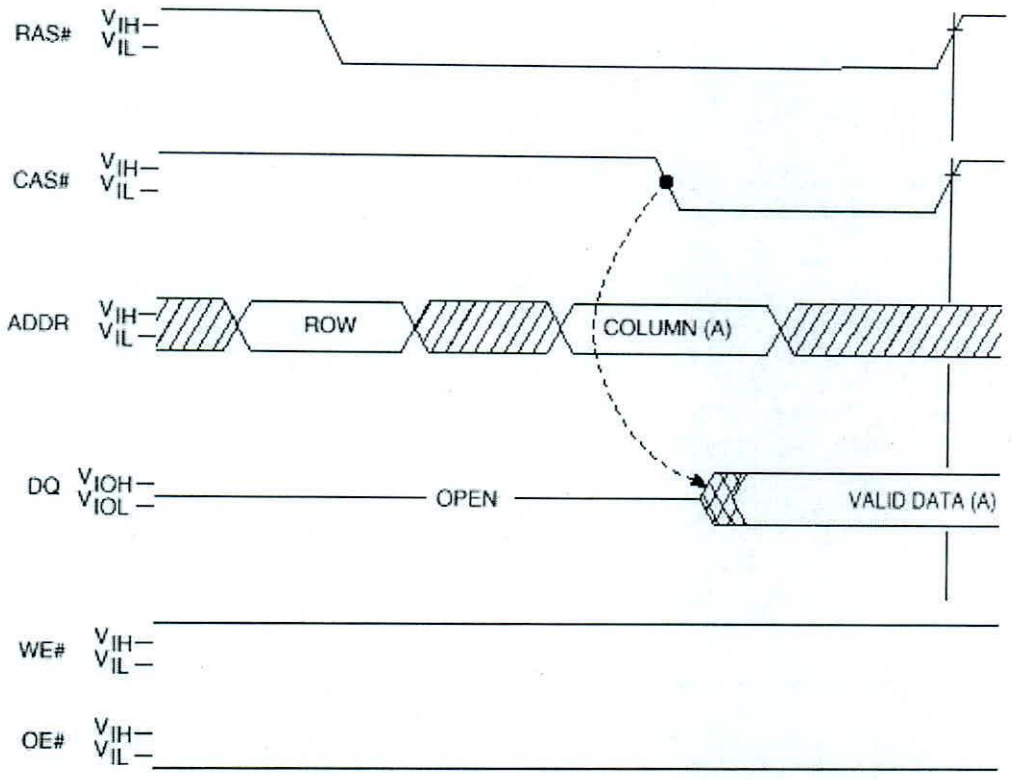


Figura 3. Diagrama de tiempos de una EDO DRAM

En el diagrama se observa que el *bus* de direcciones de la **DRAM** al estar multiplexado primero recibe la dirección de fila deseada, la cual debe estar presente antes y después del flanco de bajada de la señal de control **RAS#**; después de recibir la dirección de fila se recibe la dirección de columna, la cual es validada en el flanco de bajada de la señal de control **CAS#**. El *pin* **OE#** (*Ouput Enable*) utilizado para controlar las salidas **DQ** del *bus* de datos [Micron, 1997] en la mayoría de los sistemas esta conectado a nivel lógico cero debido a que las señales **RAS#** y **CAS#** controlan correctamente todos los estados del *bus* **DQ**, debido a que la señal **WE#** (*Write Enable*) esta en nivel lógico 1, que corresponde a un ciclo de lectura, en donde los datos validos se dan después

del flanco de bajada de **CAS#**. El ciclo de escritura de una DRAM es similar al de lectura con la diferencia que **WE#** esta a nivel lógico cero y los pines **DQ** se comportan como entradas.

Hasta ahora sólo hemos visto como deben trabajar las señales de control **RAS#**, **CAS#** y **WE#** en un ciclo normal de trabajo de una DRAM, ya sea en lectura o escritura, pero ¿cómo deben trabajar en un ciclo de actualización?. Para responder esta pregunta, primero debemos tener en cuenta que las DRAM son referidas como de dos tipos, de [Micron, 1997] actualización estándar y de actualización extendida. Todas las DRAM tienen un tiempo en el cual su contenido debe ser actualizado y si no son actualizados dentro de dicho tiempo los datos contenidos en sus celdas de memoria se pueden perder. A este tiempo limite se le conoce como tiempo de actualización; y junto con el número de páginas ó el número total de ciclos de actualización llevados a cabo en dicho tiempo determinan el tipo de memoria utilizada, así, si al dividir el tiempo en el cual se debe actualizar completamente la memoria entre el número de ciclos de actualización que se deben ejecutar se obtiene 15.6 s, se trata de una memoria de actualización estándar, si el resultado es 125 μ s, se trata de una memoria de actualización extendida. En la tabla 1 se lista algunas **DRAM** estándares y sus respectivas especificaciones de actualización.

Como se puede observar en la tabla 1, la mayoría de las memorias son de tipo estándar, las cuales a pesar de tener diferentes tiempos de actualización, tamaños y número de paginas tienen una misma taza de actualización (*Refresh Rate*). Este valor constante de la taza de actualización nos indica que podemos diseñar un controlador que ejecute un ciclo de actualización cada 15.6 μ s; a esta forma de actualizar la memoria se le conoce como actualización distribuido; y consiste en realizar varios ciclos de actualización, donde cada actualización está igualmente espaciado y son llevados a cabo cada 15.6 μ s, de tal forma que todas las filas serán actualizadas dentro del tiempo de actualización especificado por la memoria, así la memoria de 4Mx1 de la tabla 1 ejecutará un único actualización cada 15.6 μ s. Pero esta no es la única forma de actualizar una DRAM, también se puede realizar un actualización tipo ráfaga, la cual consiste en realizar una serie de actualizaciones, una tras otra, accediendo a todas las filas en dicho ciclo de actualización. [Intel, 1997] Durante la actualización ninguna otra petición es atendida quedando el *bus* de datos de la DRAM en alta impedancia. Es decir todas las filas son actualizadas una tras otra dentro del tiempo de actualización establecido para la memoria, así en la tabla 1, para la misma memoria de 4Mx1 por ejemplo, esto significa que los 1024 ciclos necesarios para actualizar todas las celdas de memoria son realizados uno tras otro antes de cada 16ms.

Tabla 1. Tiempos de refrescos para diferentes memorias DRAM

DRAM	REFRESH TIME	NUMBER OF CYCLES	REFRESH RATE
4 Meg x 1	16ms	1,024	15.6µs
256K x 16	8ms	512	15.6µs
256K x 16 (L version)	64ms	512	125µs
4 Meg x 4 (2K)	32ms	2,046	15.6µs
4 Meg x 4 (4K)	64ms	4,096	15.6µs

En la Fig. 4 se puede apreciar la diferencia entre la actualización distribuida y la actualización tipo ráfaga; la actualización distribuida es la más común de las dos formas de actualización y generalmente implica que el controlador permite que el ciclo actual de lectura ó escritura sea completado e inmediatamente después realiza la actualización de los datos, y no acepta ninguna petición de lectura ó escritura durante dicho proceso.

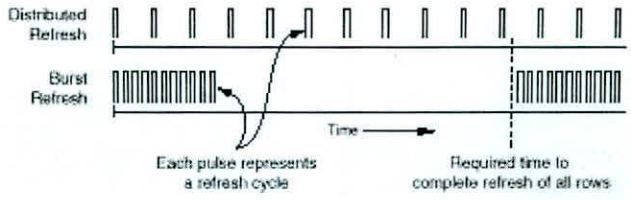


Figura 4. Comparación entre el refresco en ráfaga y el distribuido.

La ventaja de la actualización distribuida permite diseñar un controlador sin preocuparnos del tiempo de actualización, ya que esta se realizará cada 15.6*us*, actualizando completamente los datos de la memoria antes del tiempo de actualización especificado.

Sin importar la forma de actualización que se utilice, ya sea de ráfaga ó distribuido, la actualización puede ser llevado acabo de tres formas distintas, utilizando un actualización de solo **RAS**, de **CAS** antes de **RAS**, y actualización oculta.

En la actualización de solo **RAS,** referido en adelante como **ROR** (*Ras Only Refresh*), se selecciona una fila activando la señal **RAS#**, esto selecciona una pagina entera de memoria siempre y cuando la señal de **CAS#** permanezca desactiva; luego se realiza la actualización de todos los datos de la pagina seleccionada mediante un ciclo de lectura, para lo cual se lleva la señal **WE#** a 1 lógico. Este proceso se debe llevar a cabo para todas las filas, para que todas las celdas de memoria de la **DRAM** regeneren su contenido. [Micron, 1997] En este tipo de actualización, es función del controlador proveer las direcciones a ser actualizadas y asegurar que todas las filas son actualizadas dentro del tiempo apropiado.

Es importante mantener la señal de **CAS#** desactivada, nivel lógico cero, durante la actualización **ROR** para seleccionar de esta manera una página completa, y poder actualizar las celdas de memoria de dicha pagina con una lectura; la lectura realizada en el ciclo de actualización es interna ya que debido a que la señal de **CAS#** esta desactivada el bus de datos estará en alta impedancia, esto se puede apreciar en la siguiente figura.

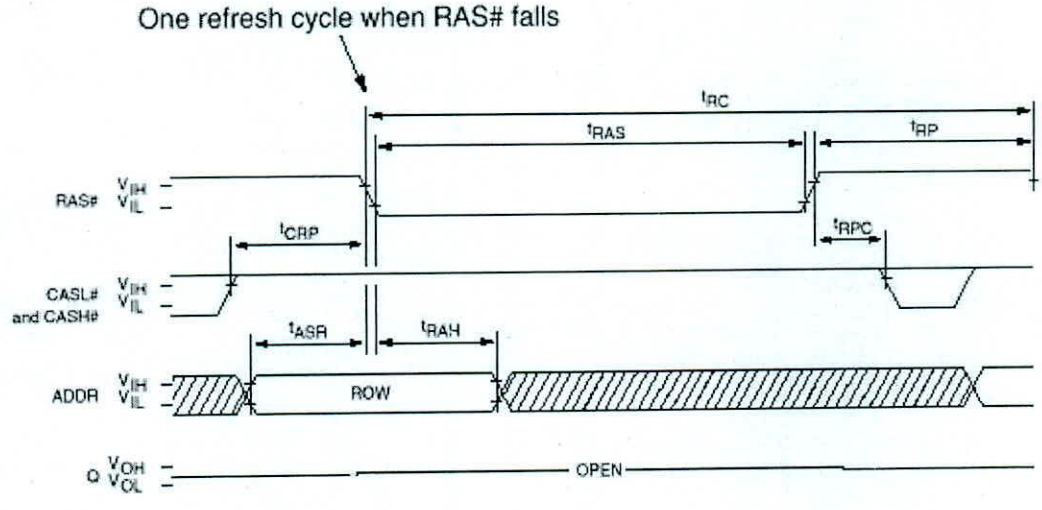


Figura 5. Diagrama de tiempos en la actualización tipo ROR.

La fila a actualizar debe proveerse a través de los pines de direcciones, para ello, el controlador debe utilizar un contador externo que genere todas las filas a actualizar, por lo tanto, debemos tener en cuenta el número de ciclos de actualizaciones necesarios para regenerar totalmente el contenido de la memoria. Además, dependiendo del tipo de actualización utilizado necesitaremos un contador que genere un actualización cada 15.6 μ s μ para el tipo distribuido, o un contador que active una ráfaga de actualizaciones antes del tiempo de actualización limite.

En la actualización de **CAS** antes de **RAS,** referido en adelante como **CBR** (*Cas Before Ras*), la señal de **CAS#** se activa (0 lógico) y luego se debe activar la señal de **RAS#**. Cada ciclo de actualización se realizara en el flanco de bajada de la señal de **RAS#**, mientras **CAS#** se mantenga activa. La señal **WE#** debe estar en alto en el flanco de bajada de **RAS#.**

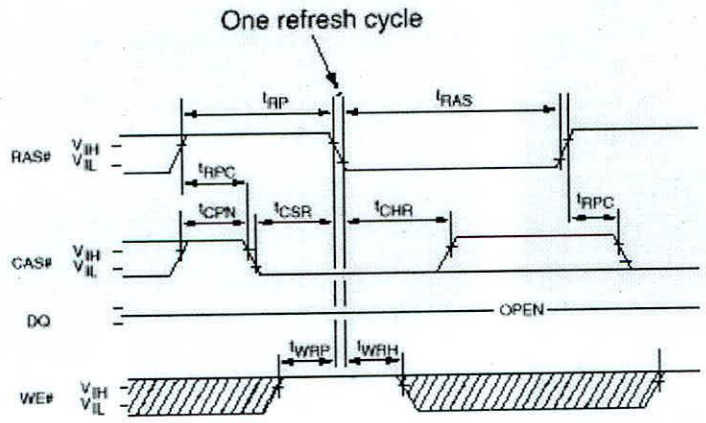


Figura 6. Un ciclo de refresco CBR

En este caso el *bus* de datos (**DQ**) también se mantendrá en alta impedancia durante la actualización. En la Fig. 6 se observa que la señal de **CAS#** va a 1 lógico y luego a cero lógico, antes de la señal de **RAS#**, para una serie de actualizaciones continuos (tipo ráfaga) esto no es necesario, pudiéndose mantenerse la señal de **CAS#** activa baja durante todo el ciclo de actualización, esto se observa mejor en la Fig. 7, en donde se realizan tres actualizaciones **CBR** en forma continua. En la misma figura se muestra que es posible tener la señal **WE#** en nivel lógico cero, pero para que se produzca la actualización es necesario que WE# este en 1 lógico (lectura), antes y después del flanco de bajada de la señal de **RAS#**.

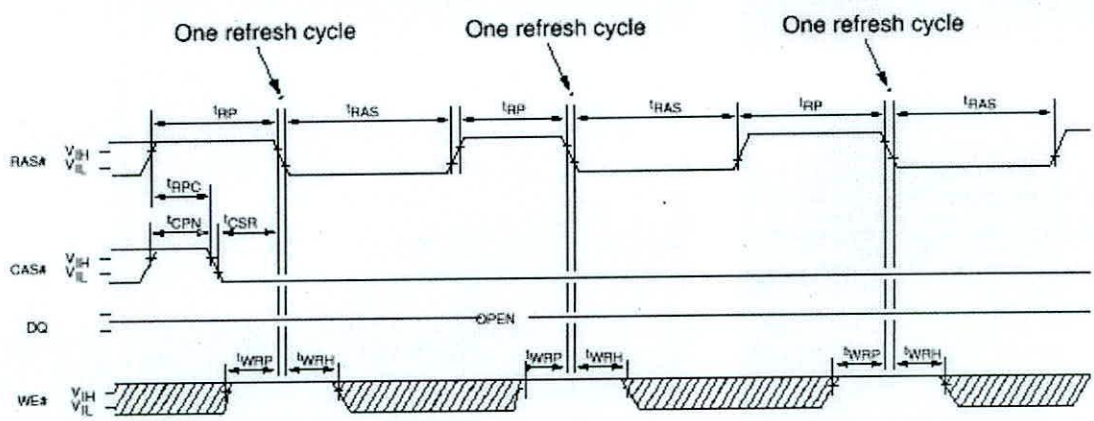


Figura 7. Actualización CBR tipo ráfaga

En la actualización **CBR** se aprovecha el contador interno de actualización de la **DRAM**. Este contador es inicializado a una cuenta en forma aleatoria cuando la memoria es energizada, y su cuenta se incrementa automáticamente cada vez que se realiza un actualización **CBR**; de esta forma no es necesario utilizar un contador externo y el controlador no necesita proveer la dirección de la fila a actualizar, lo cual es un beneficio en aplicaciones donde la potencia juega un papel importante ya que no es necesario gastar corriente adicional en conmutar las líneas de direcciones en el bus. Este contador interno no puede ser reseteado manualmente, sino que cuando el contador llegue a su cuenta máxima, automáticamente se inicializará y empezará nuevamente a contar.

La ventaja de implementar un actualización tipo **CBR** respecto a un actualización tipo **ROR**, no sólo está en el ahorro de potencia y en la simplificación del diseño del controlador, sino también en que realizando un actualización **CBR** de forma distribuida a una taza de actualización de 15.6*us* se puede trabajar con diferentes tipos de **DRAM** sin tener que preocuparse en el problema de la actualización.

II. DISEÑO DE UN CONTROLADOR EDO DRAM

El controlador de memorias es un sistema secuencial encargado de generar las señales que permiten utilizar correctamente la memoria **EDO DRAM.** Para realizar el diseño del controlador debemos primeramente definir cuales son las señales de entrada y salida del sistema. Así, en una primera aproximación tenemos que el controlador de memorias **EDO DRAM** requiere como entradas:

- El *bus* de direcciones, **ADDR.**
- La señal de control de lectura/escritura del microprocesador, **WR#.**
- La señal del reloj, **CLK.**

y como salidas:

- La señal de **RAS#.**
- La señal de **CAS#.**
- La señal de control de lectura/escritura **WE#.**
- El bus de direcciones multiplexado.

Se puede observar que el *bus* de datos no se considera en el diseño del controlador. Esto se debe a que aunque la memoria utiliza el bus de datos, el controlador de memorias en ningún momento ejerce un control directo sobre él. En las Fig. 5, 6 y 7 se mostraba el ciclo de actualización de una memoria dinámica mediante **ROR** y **CBR**, en dichas gráficas se observa que el *bus* de datos se encuentra en alta impedancia durante la actualización; pero no es el controlador de memorias el que produce que el bus de datos pase a alta impedancia; sino las señales de **RAS#** y **CAS#** generadas durante el ciclo de actualización. Es por esto que podemos desestimar cualquier control del bus de datos en el diseño del controlador de memorias.

De los requisitos de entrada y salida anteriormente establecidos podemos realizar un primer esquema del controlador a diseñar.

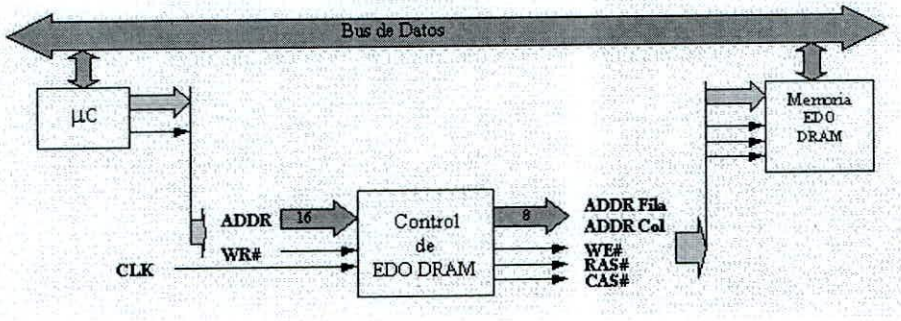


Figura 8. Entradas y Salidas del controlador a diseñar

Pero el controlador así representado aún no nos da una amplia información del diseño que debemos realizar, para detallar más en su arquitectura debemos tener en cuenta las tareas que debe realizar dicho controlador. Así podemos mencionar entre sus funciones básicas:

- La multiplexación del *bus* de dirección, en filas y columnas.
- La generación de las señales de **RAS#** y **CAS#** para el ciclo de escritura y lectura de la **EDO DRAM**.
- La generación de las señales de **RAS#** y **CAS#** para el ciclo de actualización, teniendo en cuenta que estas señales dependen del tipo de actualización a implementar, ya sea **ROR** ó **CBR**.
- Asegurar que la señal **WE#** permanecerá en nivel lógico uno (lectura) durante el periodo de actualización de la memoria.
- Asegurar que toda las celdas de memoria sean actualizadas antes del tiempo de actualización limite.

De las funciones básicas establecidas se desprende la necesidad de asegurar que la señal **WE#** se encuentre en el nivel lógico 1 durante la actualización y que sea igual a **WR#** durante el trabajo normal de la memoria; esto se puede lograr haciendo uso de un multiplexor cuya señal de control sea manejada por el circuito de actualización del controlador. Este circuito debe asegurar que el multiplexor seleccione el nivel lógico 1 para la señal **WE#,** mientras la memoria es refrescada para asegurar la lectura y también asegurar que toda la memoria sea completamente actualizada antes del tiempo límite. Además en el caso de que la actualización sea **ROR** hay que tener en cuenta que se debe tener un contador externo que genere las direcciones de memoria a actualizar, y deben ser estas las que lleguen al *bus* de direcciones de la **DRAM**.

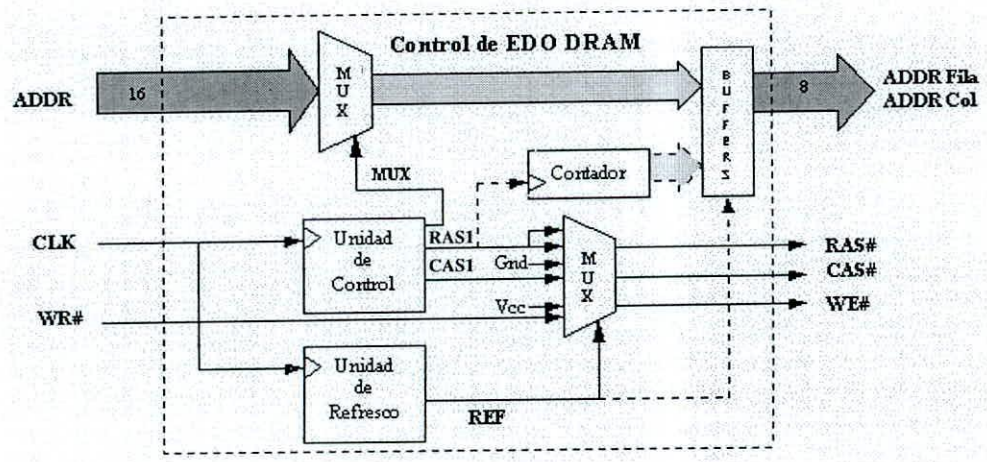


Figura 9. Diagrama de Bloques del controlador

En el esquema mostrado las líneas punteadas significan que sólo son necesarias en el caso de utilizar un actualización tipo **ROR.** Además se observa que se utilizan dos señales internas **MUX** y **REF#**, la señal de

MUX sirve para multiplexar el *bus* de direcciones y generar las direcciones de filas y columnas, y la señal de **REF#** para indicar que se está llevando a cabo la actualización de la memoria.

Haciendo un análisis de las figuras anteriormente mostradas se puede comprobar que hasta el momento no se había presentado ninguna señal de reloj; esto se debía a que las memorias utilizadas son *asincronas y su funcionamiento responde únicamente a las señales de entrada presentes. El controlador en cambio necesita de una señal de reloj con la cual sincronizar las señales de control generadas **RAS#** y **CAS#**, así como para temporizar las líneas de dirección y multiplexarlas en direcciones de filas y columnas (**MUX**).

Observemos nuevamente los ciclos de lectura y escritura de la memoria dinámica pero esta vez sincronizadas con el flanco de subida de una señal de reloj, ver Fig. 10 y Fig. 11.

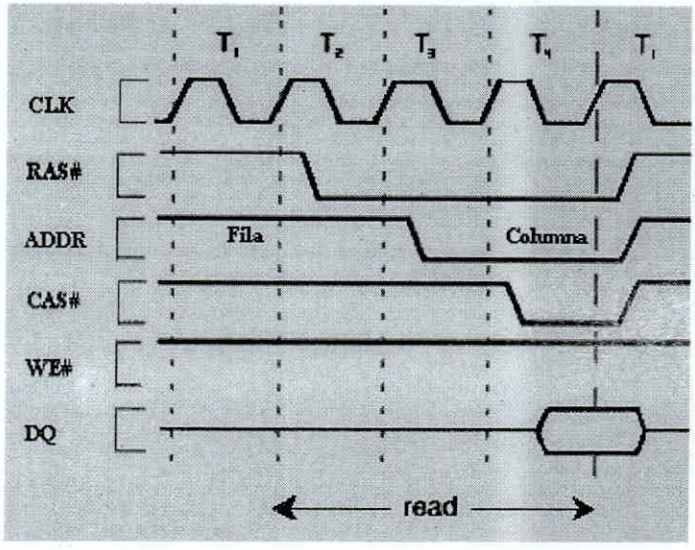


Figura 10. Diagrama de tiempo para un ciclo de lectura.

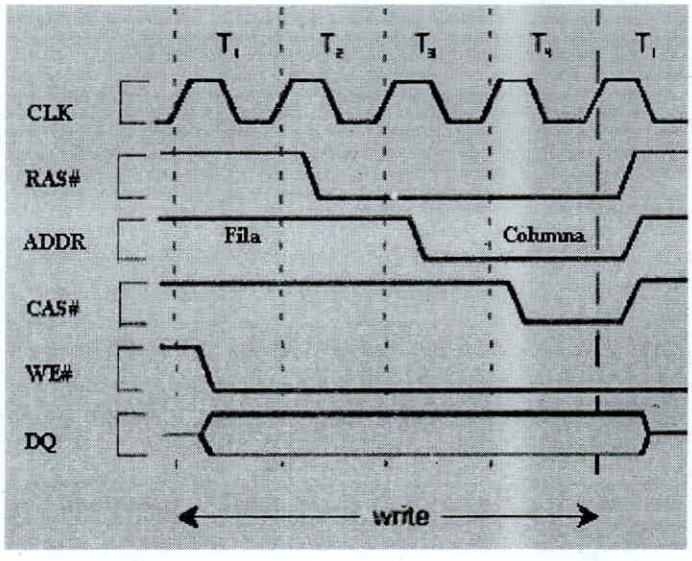


Figura 11. Diagrama de tiempo para un ciclo de escritura.

* En la actualidad ya existen memorias sincronas basadas en el protocolo I^2C

Como se puede apreciar tanto en el ciclo de escritura como el de lectura las señales de **RAS#, CAS#** y **ADDR** son iguales; además la señal **ADDR** del *bus* de direcciones nos indica que primero se deben proporcionar las filas y luego las columnas. Esto requiere de una señal que multiplexe el bus de dirección en direcciones de filas y columnas según el diagrama de tiempos de la señal **ADDR.** Como señal de control para multiplexar y temporizar las direcciones utilizamos la señal interna **MUX** generada por la unidad de control. En las Fig. 10 y 11 se observa que se necesitan 4 pulsos de reloj para completar un ciclo de trabajo de la memoria, observar en la Fig. 10 que en el 4to pulso de reloj, en un ciclo de lectura, la memoria utiliza el bus de dato DQ; y en el ciclo de escritura (Fig. 11) en el 4to pulso de reloj el microprocesador está terminando de utilizar el *bus* de datos DQ. Se denomina entonces latencia del sistema al número de ciclos de reloj que se tiene que realizar para obtener la respuesta deseada en el sistema, esto significa que en nuestro diseño tendremos una latencia de 4 debido a que cada petición de lectura o escritura demorará siempre 4 ciclos de reloj en ejecutarse.

2.1 Diseño de la unidad de Control

La unidad de control como se puede apreciar en la Fig. 9 tiene como entrada la señal de reloj **CLK** y como salidas las señales de control **RAS1, CAS1** y **MUX.** El diagrama de tiempos de las señales de salida se muestran en la Fig. 12, en donde las señales **RAS#** y **CAS#** representan a **RAS1** y **CAS1.** Esta diferencia de notación se utiliza para distinguir entre las salidas de la unida de control y la salidas del sistema (controlador de memorias).

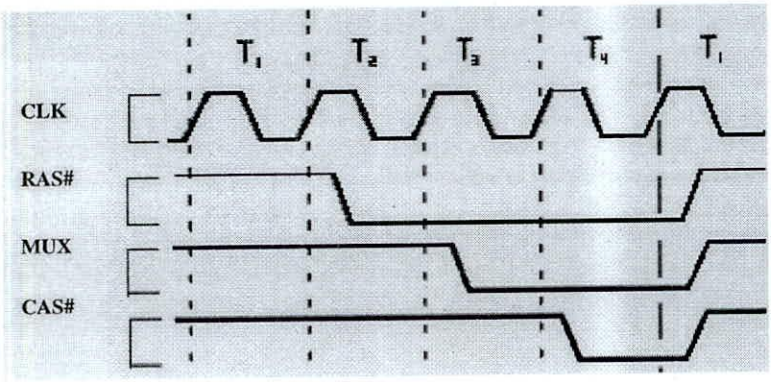


Figura 12. Diagrama de tiempos de la unidad de Control.

De donde podemos obtener la siguiente tabla de estados para la unidad de control.

Tabla 2. Estados de la unidad de control

ESTADO ACTUAL			ESTADO SIGUIENTE		
MUX	RAS1	CAS1	MUX*	RAS1*	CAS1*
1	1	1	1	0	1
1	0	1	0	0	1
0	0	1	0	0	0
0	0	0	1	1	1
1	1	1	1	0	1

Expresando dicha tabla en diagrama de estados tendremos

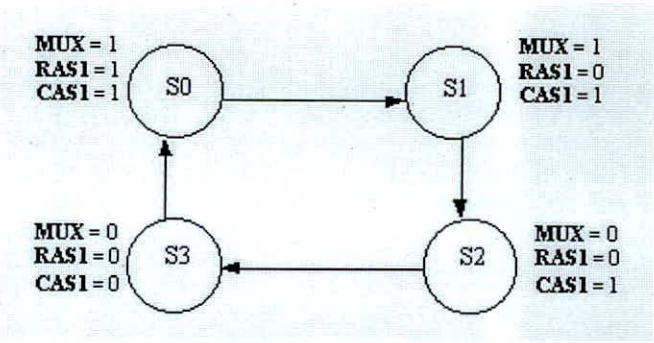


Figura 13. Diagrama de estados de la unidad de control.

Del diagrama de estados se observa que la unidad de control a diseñar es una máquina de estados **MOORE**, la cual cambia de estado con el flanco de subida del reloj (ver Fig. 12). Esta máquina de estados se puede describir con el leguaje de descripción de hardware **VHDL**, ver apéndice I.

La simulación de dicho programa es mostrada en la siguiente figura.

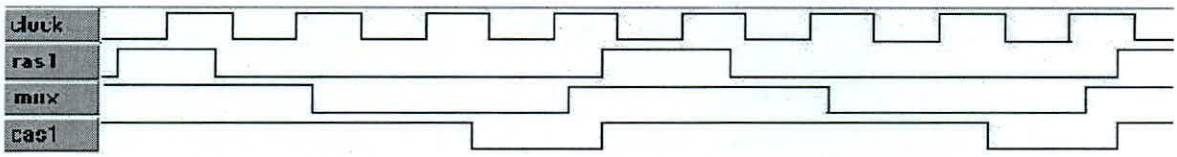


Figura 14. Simulación de la máquina de estados de la unidad de control.

La unidad de control diseñada nos proporciona el control de la memoria mediante las señales **RAS1** y **CAS1** durante los ciclos de lectura y escritura, pero no durante el ciclo de actualización. Antes de ver como debemos construir las señales para el ciclo de actualización, terminemos con el ciclo normal de trabajo, diseñando el *buffer* del *bus* de dirección.

2.2 Diseño del circuito de multiplexación

La dirección de la celda de memoria en la que se escribirá ó de la cual se leerá un dato, son separadas por la señal **MUX** de la unidad de control, en direcciones fila y columna; pasando al *bus* de dirección de la memoria sólo en un ciclo normal de trabajo. Cuando se realiza la actualización de la memoria este *bus* debe estar en alta impedancia. Si la actualización es de tipo **ROR** se debe tener en cuenta que el sistema debe proporcionar las líneas de dirección a actualizar, para ello se puede utilizar un contador que las genere y un *buffer* para almacenarlas. Se utiliza la señal generada por el circuito de actualización (petición de actualización) para determinar cual de los *buffers* estará activo y cual en alta impedancia. Así durante la actualización funcionará el *buffer* del contador y durante el ciclo de trabajo el *buffer* del *bus* de direcciones.

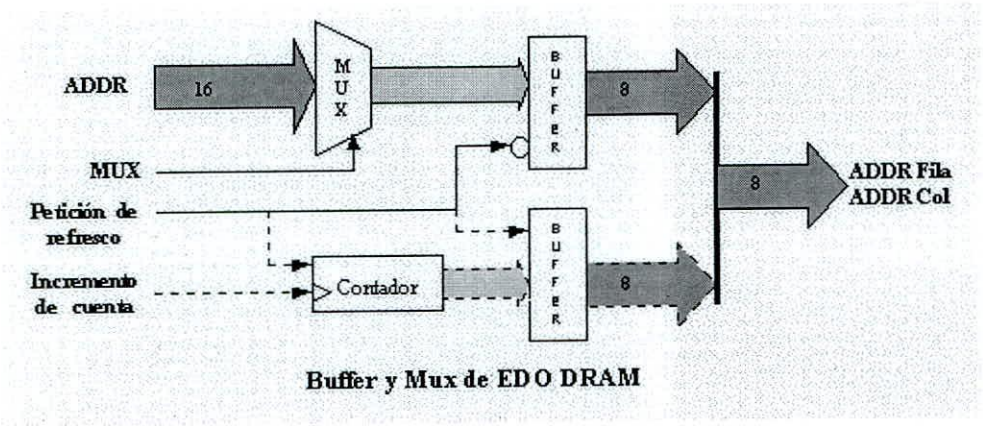


Figura 15. Diagrama del circuito de multiplexación.

En donde las líneas punteadas muestran las conexiones que son solo necesarias para una actualización tipo **ROR**.

Desarrollemos el diseño del circuito considerando que estamos utilizando un actualización tipo **ROR**, para ello el contador debe generar todas las líneas de direcciones de la memoria a actualizar, es decir el contador debe realizar tantas cuentas como filas tenga la memoria. Así para una memoria de 64kx8 con una matriz cuadrada de 2^8 filas y 2^8 columnas tendremos que utilizar un contador de 8 bits. Dicho contador debe activarse solamente cuando se realice la petición de actualización, **REF#** (*Refresh Request*) activo bajo, generada por el circuito de actualización. Cuando se realiza la petición de actualización se debe activar el *buffer* del contador y desactivar el buffer de las líneas de direcciones.

Además debemos tener en cuenta si se trata de un actualización tipo *burst* ó un actualización distribuido. En un actualización distribuido se realiza un solo actualización cada 15.6 *us* y cuando llega a su máxima cuenta vuelve a empezar, en cambio en la actualización tipo ráfaga donde se realizan varios ciclos de actualización seguidos es necesario mantener el contador desactivado mientras no se realiza actualización, esto se logra manteniendo activa la señal de limpiado del contador, forzando a cero el contador hasta que no se produzca la petición de actualización, **REF#** a nivel lógico cero. En la tabla 3 se puede observar el resumen de dicho funcionamiento.

Tabla 3. Estados de las señales de multiplexación en el ciclo de trabajo y actualización.

CICLO	REF#	CONTADOR	Buffer Contador	Buffer direcciones
Trabajo	1	00h	Alta impedancia	activo
Actualización	0	activo	activo	Alta impedancia

Hasta este punto se ha descrito el lineamiento general de la etapa de multiplexación pero aun queda por ver que señal se utilizara como reloj del contador. En realidad, esta selección no implica un gran problema sino mas bien un simple análisis del funcionamiento del circuito. El circuito de control tiene una latencia de 4, y las señales generadas por dicho circuito que sirven exclusivamente para el ciclo de trabajo de la memoria se deben adaptar y utilizar para el ciclo de actualización de la memoria; de aquí se deduce que el contador para actualizar una página de la memoria debe incrementar su cuenta cada 4 pulsos de reloj. Según esto podemos utilizar cualquiera de las señales generadas por la unidad de control como señal de reloj, debido a que tiene la frecuencia deseada. Pero antes, debemos tener en cuenta que la menor dirección de fila en la memoria es 00h, y está página también debe ser actualizada; para esto, se inicia el ciclo de actualización con la señal de **REF#** en activo bajo, el contador empieza en 00h y se mantiene en esa cuenta durante 4 pulsos de reloj, mientras se generan las señales de actualización, luego incrementa su cuenta para actualizar la pagina 01h, y así sucesivamente hasta terminar la actualización de la memoria en la página FFh. Luego se desactiva la señal **REF#** con activo alto y regresa al ciclo normal de trabajo de la memoria. Las cuentas se han incrementado en el final de cada ciclo de actualización

para permitir que la primera pagina sea refrescada. De esta forma se puede utilizar como señal de reloj para el contador la señal negada de **RAS1** proveniente de la unidad de control, la cual incrementara el contador en su flanco de bajada.

El programa se muestra en el apéndice II y su simulación en la siguiente figura.

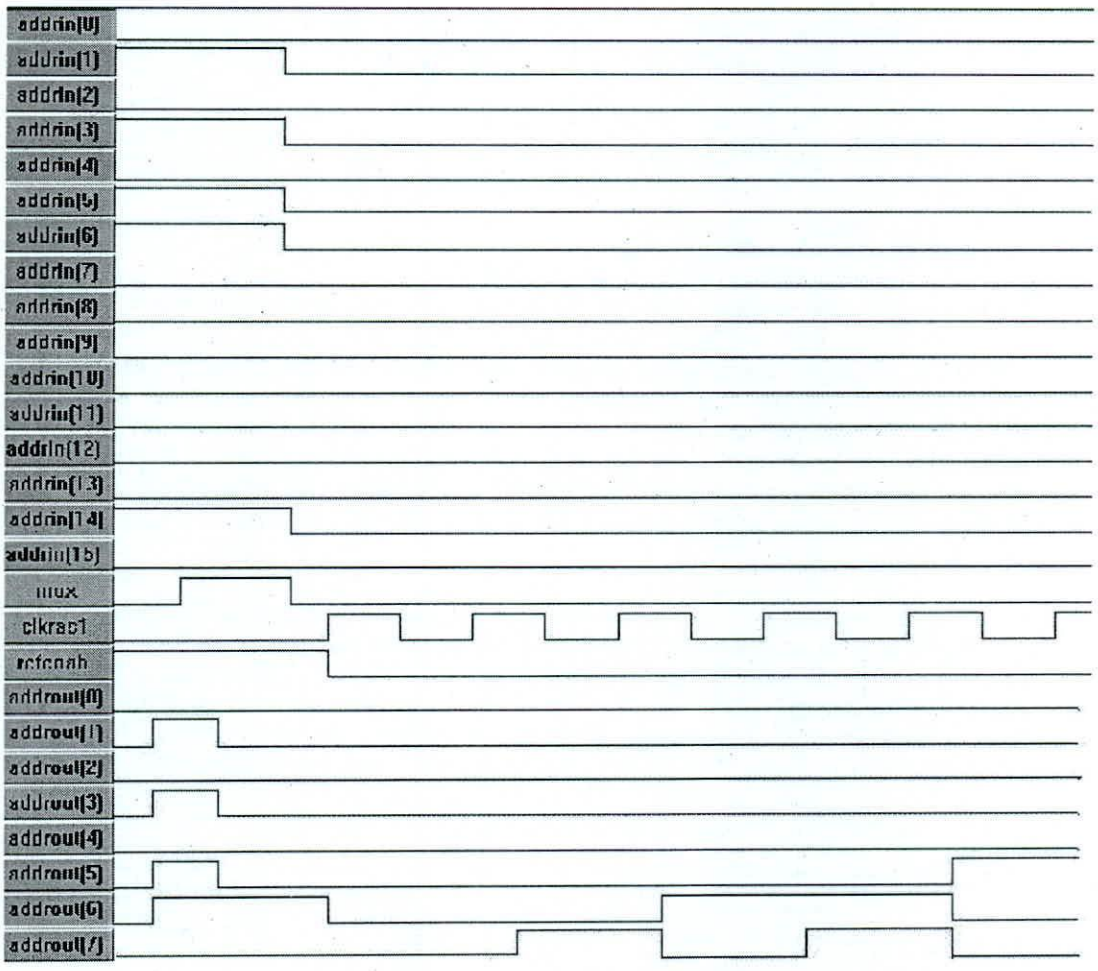


Figura 16. Simulación de la unidad de multiplexación y actualización.

En donde ADDRIN es el *bus* de direcciones de entrada y *ADDROUT* es el *bus* de direcciones multiplexado que se obtiene a la salida del sistema de actualización. En la Fig. 16 se observa que mientras **REF#** está en 1 lógico, la señal que se obtiene como salida del *bus ADDROUT* corresponde a las direcciones multiplexadas de filas y columnas. Cuando **MUX** está en nivel lógico 1 pasan las filas y cuando está en 0 lógico pasan las columnas al *bus ADDROUT*. Además mientras **REF#** se mantenga en 1 lógico el contador permanecerá limpio, quedando en cero, pero cuando **REF#** pasa a 0 lógico se activa el ciclo de actualización y el contador; pasando su valor al buffer conectado al *bus ADDROUT*. Su valor es usado para actualizar completamente una fila en la actualización tipo **ROR.**

2.3 Diseño del circuito de actualización

El circuito de actualización se encarga de generar la señal de petición de actualización **REF#** y de adaptar las señales de control **RAS1, CAS1** y **WE#** para generar las señales de control **RAS#, CAS#** y **WR#** que sirven para controlar la memoria tanto en el ciclo de trabajo como en el de actualización.

Este circuito antes del tiempo limite de actualización genera la petición de actualización y mantiene dicha petición hasta que toda la memoria sea completamente refrescada. La actualización utilizada es del tipo **ROR** por ráfaga; por lo que se tiene que actualizar toda la memoria en una sola petición de la señal REF. Para poder hacer esto el circuito solamente debe utilizar un contador que temporice el tiempo en el que puede trabajar y el tiempo necesario para actualizar toda la memoria antes de que sus datos se pierdan.

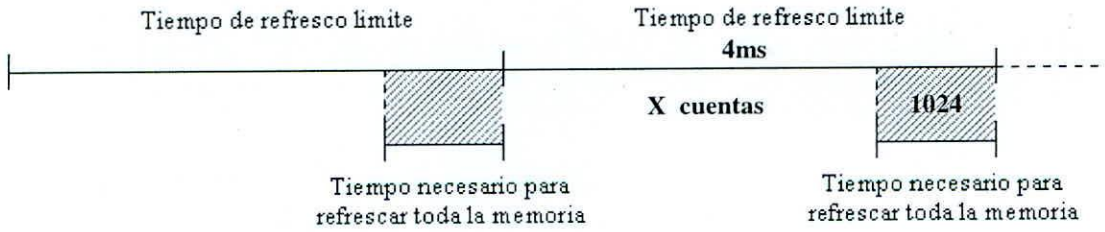


Figura 17. Tiempos de trabajo y de actualización de la memoria DRAM

La memoria 64Kx8 bits utilizada tiene una pagina de 8 bits, lo cual como ya se ha indicado requiere un contador de 8 bits que genere las 256 líneas de actualización y como cada ciclo de actualización tiene una latencia de 4, el tiempo necesario para actualizar la memoria es de 4x256 por el periodo del reloj. Es decir se requieren 1024 cuentas para actualizar completamente la memoria. Además en la actualización por ráfaga se requiere tener en cuenta el tiempo limite de actualización característico de cada memoria, ver tabla 1. Para una memoria EDO de 64kx8bits este tiempo es de 4ms [Texas instrument, 1996].

$$4ms \leq T_{trabajo} + T_{actualización}$$
$$4ms \leq (X+1024)xPeriodo\_reloj$$

Entonces cada 4ms la memoria debe ser completamente refrescada, de donde los X primeros ciclos de reloj serán utilizados para el trabajo normal de la memoria y los 1024 ciclos finales para la actualización de la memoria. Como se puede apreciar según este planteamiento aparentemente faltarían datos para poder determinar los valores de X y Período_reloj, esto se debe a que no se esta utilizando las especificaciones de las memorias, por ejemplo nosotros sabemos que nuestro sistema tiene una latencia de 4, entonces conociendo el tiempo del ciclo de lectura o escritura (t_{RC} ó t_{WC} respectivamente) podemos determinar la frecuencia del reloj. Para la memoria utilizada de 64Kx8bits el t_{RC} es de 600ns, por lo cual.

$$600ns = 4xPeriodo\_reloj$$
$$Frec\_reloj = 6.6667\ MHz$$

Sustituyendo la frecuencia del reloj en la primera ecuación observamos que el ciclo del controlador esta formado por 26668 cuentas de las cuales X = 25644 están dedicadas al ciclo normal de trabajo de la memoria y las otras 1024 al ciclo de actualización.

Tener las 26668 cuentas significa que debemos implementar un contador que cuente de 0 (0000h) hasta 26667 (682Bh) dentro de ese "tiempo" se deben realizar las 1024 cuentas necesarias para la actualización. Por ejemplo las podemos realizar desde la cuenta 25642 (642Ah) hasta la cuenta 25642 + 1023 (6829h).

Durante esas 1024 cuentas dedicadas a la actualización de la memoria la señal *REF# debe estar activa es decir en 0 lógico, para indicar que estamos en un ciclo de actualización. La señal de **CAS1** debe pasar a 1 lógico, la señal WE proporcionada por el CPU también debe pasar a 1 lógico para asegurar un ciclo de lectura, la señal de **RAS1** proporcionada por el circuito de control permanece igual ya que se ajusta al diagrama de tiempo necesario especificado por la actualización **ROR**. En la tabla 4 se muestra un resumen de estas especificaciones.

Tabla 4. Señales generadas durante los ciclos de trabajo y actualización de la DRAM

CICLO	REF#	CAS	RAS	WE
Trabajo	1	CAS1	RAS1	WR
Actualización	0	1	RAS1	1

Observando esta última tabla se puede notar que se trata de un multiplexor simple 6 a 3 controlado por la señal **REF#.**

El diseño realizado se puede describir en VHDL como se muestra en el apéndice III. Y la simulación de dicho programa nos da como resultado para el ciclo de trabajo:

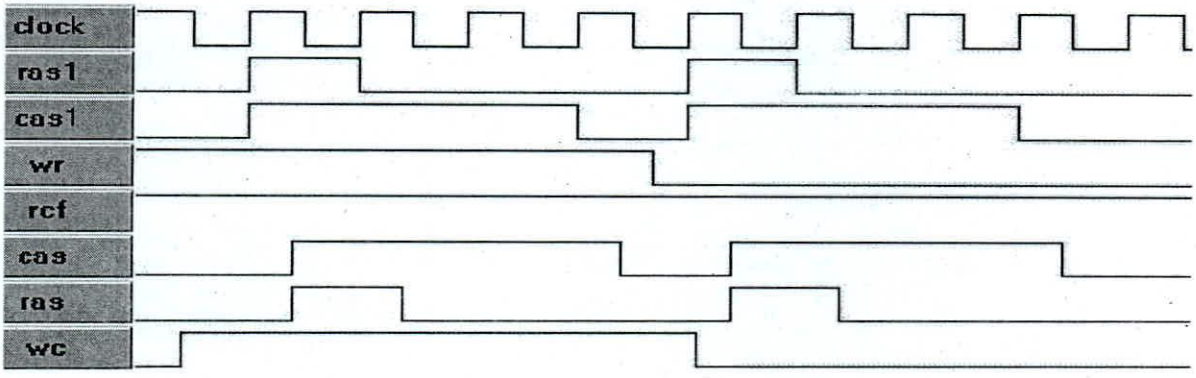


Figura 18. Diagrama del controlador en el ciclo de trabajo.

Y para el ciclo de actualización (REF# activo bajo)

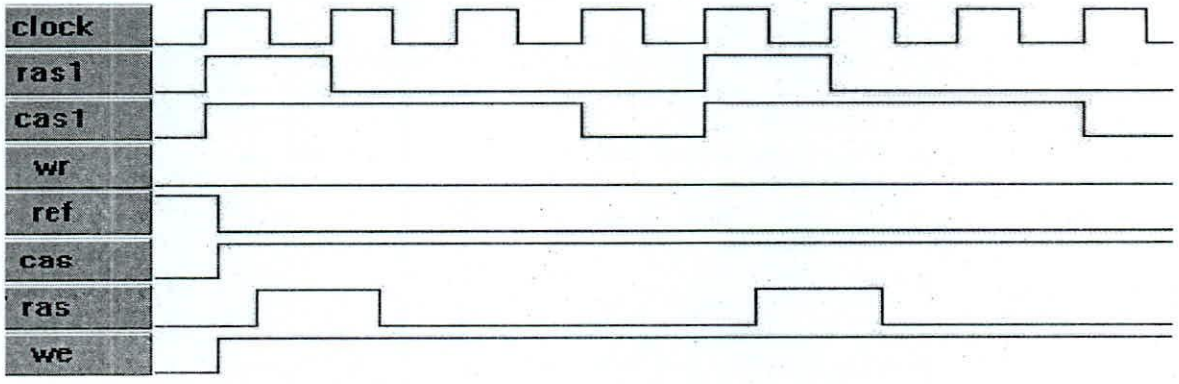


Figura 19. Diagrama del controlador en el ciclo de actualización.

* observar que la señal REF# es representada en el programa como REF, esto se debe a que el VHDL no puede utilizar el símbolo # como parte del nombre de una señal

III. VARIANDO EL DISEÑO PARA OTRAS MEMORIAS Y/O TIPOS DE ACTUALIZACIÓN

En el diseño anterior se realizó un controlador para memorias dinámicas EDO de 64Kx8 bits con un tiempo de ciclo de lectura t_{RC} = 600ns utilizando una actualización ROR por ráfaga. Pero ese mismo diseño se puede variar para realizar cualquier otro modo de actualización, ya sea ROR tipo distribuido, CBR por ráfaga ó CBR tipo distribuido, y para cualquier otra memoria con especificaciones distintas.

Para ayudar a entender se realizara unas pequeñas variaciones al programa para adaptar el diseño realizado, a una memoria EDO de Texas [Texas, 96] TMS418169 de 1024Kx16bits utilizando un actualización ROR tipo distribuido. Esta memoria tiene una especificación de 1024Kx16bits lo cual marca la pauta para nuestro primer cambio, el nuevo *bus* de direcciones será de 20 líneas, por lo cual habrá que variar el diseño de la unidad de multiplexación. La unidad de control no requiera variación alguna ya que sigue generando las señales para el ciclo de trabajo de la memoria. El circuito de actualización debe ser modificado para realizar la petición de actualización cada 15.6us para realizar un actualización distribuido. Con este lineamiento de los cambios a introducir se empezara el nuevo diseño.

3.1 Diseño de la unidad de multiplexación

El nuevo programa debe multiplexar el *bus* de direcciones de 20 a 10. Si mantenemos la actualización ROR, el contador deberá generar las direcciones a actualizar, por lo cual se debe ver si la memoria tiene un arreglo cuadrado ó rectangular, siendo esta geometría de la matriz la que determina el número de páginas a actualizar y por ende el modulo del contador a utilizar.

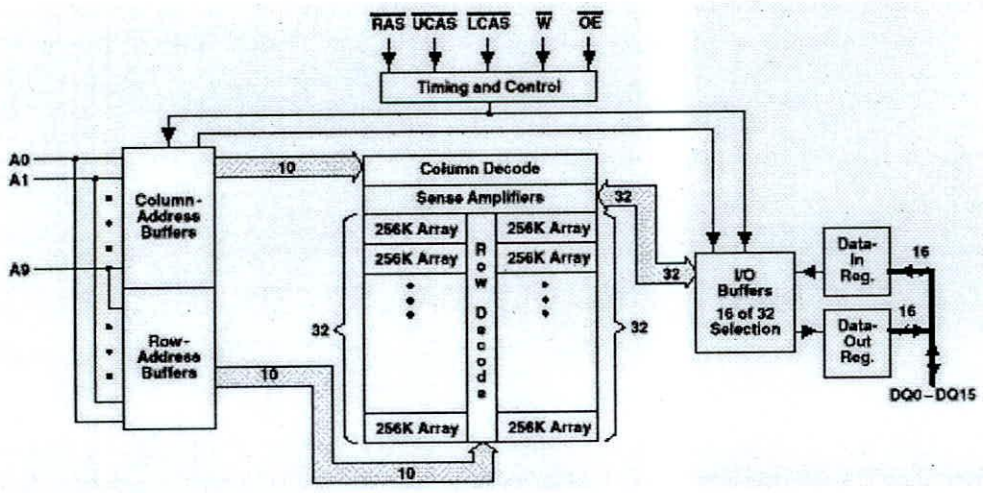


Figura 20. Diagrama de bloques de la memoria TMS418169.

Se observa que la memoria tiene una matriz cuadrada de 10x10, es decir tiene 10 filas y 10 columnas, esto significa que debemos generar con el contador 2^{10} direcciones a actualizar y para ello requerimos un contador modulo 10.

Un cambio adicional a tener en cuenta es que al realizar un actualización tipo distribuido el contador no debe ser limpiado cada vez que termine un ciclo de actualización, sino que este debe seguir aumentando la cuenta en cada ciclo hasta que llegue a la cuenta máxima y vuelva a empezar.

El programa en VHDL de la unidad de multiplexación se muestra en el apéndice IV.

3.2 Diseño del circuito de actualización

En esta etapa se desea implementar una actualización tipo distribuido y ya no una actualización tipo ráfaga, para ello debemos realizar la petición de actualización cada 15.6us. En este diseño ya no interesa el tiempo límite de actualización de la memoria, si es cada 4ms 16ms ó 32ms debido a que se trata de una actualización distribuida. Y el cálculo del módulo del contador se realiza teniendo en cuenta el tiempo de lectura t_{RC} = 150ns, y como sabemos que la actualización dura 4 ciclos de reloj, entonces tenemos:

Frec_reloj = 4/150ns = 26.667Mhz

Esta frecuencia de reloj nos indica la máxima frecuencia con la cual se puede utilizar la memoria, pudiendo ser menor, pero para menores frecuencias de reloj se tendrá una respuesta menos rápida de la memoria. Ahora para determinar el número de ciclos que debe estar trabajando la memoria antes de realizar una petición de actualización, se debe calcular:

X = 15.6*us**26.667Mhz = 416 cuentas

Entonces el contador debe realizar 416+4 cuentas, es decir 420 cuentas, para ello necesitamos un contador modulo 9 que realice cuentas desde 0 hasta 419. Por lo tanto el contador debe funcionar como sigue:

- de 000000000 a 110011111 (de 0 a 415) son los 15.6*us* en los cuales **REF#** = 1, la memoria esta en ciclo de trabajo.
- de 110100000 a 110100011 (de 416 a 419) son los 150*ns* necesarios para realizar la actualización, aquí **REF#** = 0.

El nuevo programa en VHDL para el circuito de actualización se muestra en el apéndice V.

Cuando se implementa el diseño en la memoria TMS418169 se observa que dicha memoria presenta dos señales de **CAS** (**UCAS** y **LCAS**) ver la siguiente figura.

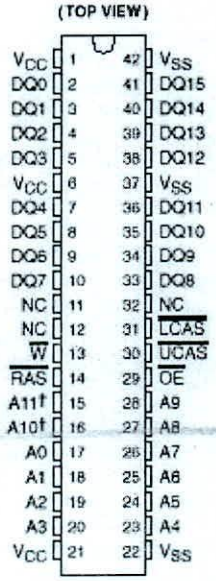


Figura 21. Encapsulado de la memoria TMS418169 de Texas.

Esta aparente complejidad no es mas que eso, aparente, ya que las señales de **CAS** generalmente se dividen en dos para poder controlar en forma independiente el *bus* de datos, es decir **LCAS** para DQ0-DQ7 y **UCAS** para

DQ8 a DQ15. Si se quiere acceder a los 16 bits de datos en forma simultanea la señal de CAS generada por nuestro sistema simplemente se conecta a ambas **LCAS** y **UCAS** a la vez.

IV. IDEAS Y SUGERENCIAS

La variación presentada anteriormente no es la única que se puede realizar, por ejemplo si se quiere implementar un actualización **CBR** podemos añadir a nuestro sistema de control una unidad de actualización la cual este conformada por una máquina de estados **MOORE** que genere las señales **RAS2** y **CAS2** necesarias para el ciclo de actualización. Estas señales serian multiplexadas en el circuito de actualización con las señales **CAS1** y **RAS1** (generadas por la unidad de control) para generar las señales **RAS** y **CAS.**

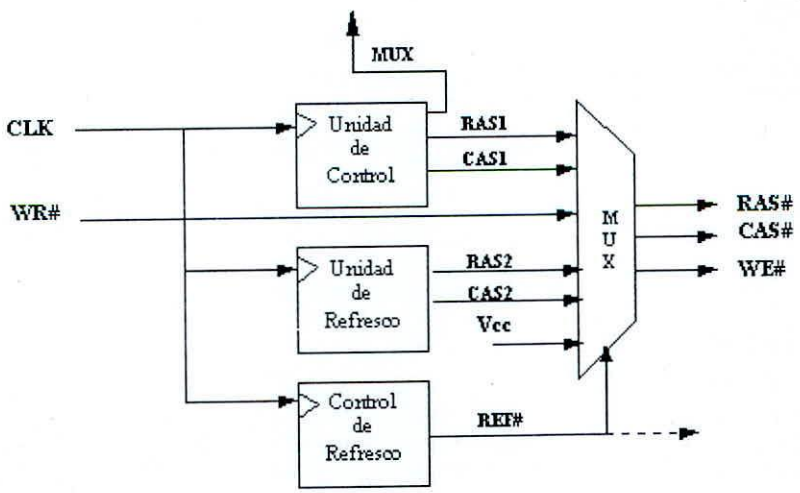


Figura 22. Diagrama simplificado del controlador DRAM

La máquina de estados para la unidad de actualización se diseña de la misma manera que se diseño la unidad de control. Es decir revisando el diagrama de tiempos de las señales **RAS** y **CAS** para el ciclo de actualización.

Se pueden realizar otras modificaciones a la unidad de control, como la de generar ciclos de lectura y/o escritura tipo ráfaga, en la que se debe notar la diferencia entre una lectura y una actualización ráfaga. Cuando hablamos de una lectura ráfaga significa que en una sola operación de lectura se pueden acceder a varias posiciones consecutivas de memoria reduciendo la latencia del sistema. En nuestro actual diseño tenemos una latencia de 4, si queremos realizar varias lecturas a posiciones de memorias consecutivas demoraremos igual en cada caso, por ejemplo 4 lecturas tendrán una latencia de 4-4-4-4, si se ha comprendido bien el funcionamiento de las memorias dinámicas se intuirá que esta latencia se puede disminuir ya que para direccionar 4 posiciones de memoria consecutivas solo se requieren los 2 bits menos significativos del *bus* de direcciones que corresponden a las direcciones de columnas en el diseño. Es decir para realizar este método de acceso no requerimos generar una nueva señal de **RAS** para realizar la 2^{da}, 3^{era} ó 4^{ta} lectura ya que la pagina seleccionada es la misma en todos los casos, pudiendo disminuir la latencia del sistema a un 4-3-3-3. Esta latencia la podemos mejorar aun mas si es que se tiene en cuenta que no es necesariamente necesario generar nuevas señales de **CAS** para cada lectura consecutiva, ya que son solo los 2 últimos bits los que varían ¿por qué debemos generar un nuevo **CAS** si de las 10 líneas del *bus* son solo 2 las que varían? Visto desde esta forma podemos reestructurar el arreglo de la memoria en 4 arreglos independientes en donde todas comparten el mismo **RAS** y **CAS** pero donde los 2 últimos bits de direcciones no están multiplexados sino que son utilizados para habilitar 1 de los 4 arreglos de memoria, reduciendo de esta manera la latencia a 4-2-2-2. En la actualidad esta técnica es muy utilizada en la fabricación de bancos de memoria para su utilización en los ordenadores.

V. CONCLUSIONES

El diseño de un controlador de memoria dinámica no requiere de una gran complejidad, sino de un simple estudio de su funcionamiento. Principalmente su diseño consiste en generar las señales apropiadas de control para la utilización de la memoria durante el ciclo de trabajo y actualización.

La necesidad de señales de control durante el ciclo de trabajo se debe a que las memorias dinámicas tienen el bus de dirección multiplexado, por lo cual es necesario crear una etapa que brinde a la memoria primero las direcciones de filas y luego de columnas. El ciclo de actualización es necesario debido a que las memorias dinámicas están construidas con condensadores cuya carga se degrada cada 5ms, por lo cual es necesario asegurar que su contenido sea refrescado periódicamente para que no pierda sus datos.

En ambos casos, sea en el ciclo de trabajo ó actualización, las señales que se deben generar son **RAS, CAS** y **WE,** y se entenderá que se esta en el ciclo de trabajo ó actualización según el diagrama de tiempo de dichas señales.

Para el ciclo de trabajo solo hay un diagrama de tiempos, pero para el ciclo de actualización se pueden tener hasta 3 diferentes diagramas de tiempo:

- El de un actualización **ROR** (actualización de solo ras)
- El de un actualización **CBR** (actualización de cas antes de ras)
- La actualización **Hidden Refresh**

Cada una de estas tres formas de actualización se puede realizar mediante dos maneras distintas:

- Mediante un actualización distribuido
- Mediante un actualización por ráfaga

En el presente trabajo, se ha realizado el diseño de un controlador de memorias con actualización **ROR** utilizando actualización distribuido y ráfaga; y se deja planteado una alternativa de solución para la implementación con actualización **CBR.**

La utilización de memorias dinámicas no debe estar solamente limitada a su uso dentro de una computadora, también debe ser utilizada en aplicaciones con microcontroladores en donde se requiera almacenar temporalmente grandes cantidades de información. Para ello el diseño realizado propone una interfaz económica y sencilla entre la memoria DRAM y el microcontrolador.

REFERENCIA BIBLIOGRÁFICA

Barry, B, Brey. *The Intel Microprocessors 8086/8088, 80186, 80286, 80386 and 80486 Architecture, Programming and Interfacing*, 1995.
ISBN 0-03-314250-2
Cypress. *Programmable Logic Data Book*, 1997.
Cypress. *VHDL Training for PLDs, CPLDs and FPGAs,* 1999.
Intel. *A Simple DRAM Controller For 25/16 Mhz i960® CA/CF Microprocessors.*
Technical Note 272628-001, 1997.
Liilen, H. *Du Microprocesseur au Micro-ordinateur*, 1993.
ISBN: 2-7091-0641-8
Micron. *EDO Compatibility with FPM DRAMs, Technical* Note TN-04-40, 1997. (a)
Micron. *Various Methods of DRAM Refresh, Technical* Note TN-04-30, 1997. (b)
Protopapas, D. A. *Microcomputer Hardware Design*, 1992.
ISBN: 0-13-582115-0

Sjholm. L.. *VHDL for Designers*, 1996.
ISBN 0-13-473414-9
Skahill, K. *VHDL for Programmable Logic*, 1998.
ISBN 0-201-89586-2
Sudhakar y. *VHDL Starter's Guide*, 1998.
ISBN 0-13-519802-X
Texas instruments. *MOS Memory Glossary, Technical Note* SMYV001. 1995
Texas instruments. *MOS Memory Data Book*, 1996.

APÉNDICE I.

```
library ieee;
use ieee.std_logic_1164.all;
entity unidad_control is port(
        clock:  in std_logic;
        ras1,cas1,mux: out std_logic);
end unidad_control;

architecture behavioral of unidad_control is
type state is (s0,s1,s2,s3);
signal d: state:=s0;
begin

-- Unidad de control genera señales mux ras1y cas1
A: process (clock)
                begin
                if (clock='1' and clock'event) then
                        case d is
                        when s0 =>
                                d<=s1;
                        when s1 =>
                                d<=s2;
                        when s2 =>
                                d<=s3;
                        when s3 =>
                                d<=s0;
                        end case;
                end if;
end process A;

B: process (d)
                begin
                case d is
                        when s0 =>
                                mux<='1';
                                cas1<='1';
                                ras1<='1';
                        when s1 =>
                                mux<='1';
                                ras1<='0';
                                cas1<='1';
                        when s2 =>
                                ras1<='0';
```

```
                                    mux<='0';
                                    cas1<='1';
                            when s3 =>
                                    cas1<='0';
                                    ras1<='0';
                                    mux<='0';
                    end case;
end process B;
end behavioral;
```

APÉNDICE II.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity muxbuff is port (
 *ref,mux,clkras1: in std_logic;
 addrin: in std_logic_vector(15 downto 0);
 addrout: out std_logic_vector(7 downto 0));
end muxbuff;

architecture behavioral of muxbuff is
  signal notclkras1: std_logic;
  signal address: std_logic_vector(7 downto 0);
  signal q: std_logic_vector(7 downto 0);
  alias addrow: std_logic_vector(7 downto 0) is addrin (15 downto 8);
  alias addcol: std_logic_vector(7 downto 0) is addrin (7 downto 0);
begin
notclkras1<=not clkras1;

-- Contador_linea_actualización
contador: process (ref,notclkras1)
 begin
 if ref='1' then
   q <= X"00";
 elsif (notclkras1='0' and notclkras1'event) then
   q <= q + 1;
 end if;
end process contador;

-- mux 16 a 8 de las líneas de dirección de entrada
mux: process (mux)
 begin
  if mux='0' then
     address<=addcol;
  elsif mux='1' then
     address<=addrow;
  end if;
end process mux;
```

* La señal es activa bajo y debería escribirse REF# pero el VHDL no soporta el símbolo #

```
-- buffer de las direcciones multiplexadas y del contador_linea_actualización
addrout(0)<=address(0) when ref='1' else q(7); -- actualización
addrout(1)<=address(1) when ref='1' else q(6); --"aleatorio"
addrout(2)<=address(2) when ref='1' else q(5); -- pines invertidos
addrout(3)<=address(3) when ref='1' else q(4);
addrout(4)<=address(4) when ref='1' else q(3);
addrout(5)<=address(5) when ref='1' else q(2);
addrout(6)<=address(6) when ref='1' else q(1);
addrout(7)<=address(7) when ref='1' else q(0);

end behavioral;
```

APÉNDICE III.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity refreq is port(
 clock,wr,ras1,cas1:  in std_logic;
 ref:  buffer std_logic;
 ras,cas,we: out std_logic);
end refreq;

architecture behavioral of refreq is
signal res: std_logic;
signal q: std_logic_vector(14 downto 0):="000000000000000";
begin

A: process (clock,q,res)
  begin
  if (clock='0' and clock'event) then
   if res='1' then
    q<="000000000000000";
     res<='0';
   else
     q <= q + 1;
   end if;
   if q="110100000101011" then    --cuenta de 0001h hasta 682Bh
            res<='1';
   end if;
  end if;
end process A;

B: process (q)
  begin
    if q>="110010000101010" then          --si q>=642Ah y
            if q<="110100000101001" then  --   q<=6829h
             ref<='0';                    --ref es 0
            else
     ref<='1';                   --sino ref es 1
            end if;
           else
     ref<='1';
```

```
   end if;
end process B;

-- mux 6 a 3 de las lineas de control
C: process (ref)
 begin
  if ref='0' then
      ras<=ras1;
                    cas<='1';
                    we<='1';
  elsif ref='1' then
                    ras<=ras1;
                    cas<=cas1;
                    we<=wr;
  end if;
end process C;

end behavioral;
```

APÉNDICE IV.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity muxbuff is port (
  ref,mux,clkras1: in std_logic;
 addrin: in std_logic_vector(19 downto 0);
 addrout: out std_logic_vector(9 downto 0));
end muxbuff;

architecture behavioral of muxbuff is
  signal notclkras1: std_logic;
  signal address: std_logic_vector(9 downto 0);
  signal q: std_logic_vector(9 downto 0);
  alias addrow: std_logic_vector(9 downto 0) is addrin (19 downto 10);
  alias addcol: std_logic_vector(9 downto 0) is addrin (9 downto 0);
begin
notclkras1<=not clkras1;

-- Contador_linea_actualización
contador: process (ref,notclkras1)
 begin
 if (ref='0' and (notclkras1='0' and notclkras1'event)) then
   q <= q + 1;
 end if;
end process contador;

-- mux 20 a 10 de las líneas de dirección de entrada
mux: process (mux)
 begin
```

```
 if mux='0' then
    address<=addcol;
 elsif mux='1' then
    address<=addrow;
 end if;
end process mux;

-- buffer de las direcciones multiplexadas y del contador_linea_actualización
addrout(0)<=address(0) when ref='1' else q(9); -- actualización
addrout(1)<=address(1) when ref='1' else q(8); --"aleatorio"
addrout(2)<=address(2) when ref='1' else q(7); -- pines invertidos
addrout(3)<=address(3) when ref='1' else q(6);
addrout(4)<=address(4) when ref='1' else q(5);
addrout(5)<=address(5) when ref='1' else q(4);
addrout(6)<=address(6) when ref='1' else q(3);
addrout(7)<=address(7) when ref='1' else q(2);
addrout(8)<=address(8) when ref='1' else q(1);
addrout(9)<=address(9) when ref='1' else q(0);

end behavioral;
```

APÉNDICE V.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity refreq is port(
 clock,wr,ras1,cas1:  in std_logic;
 ref:  buffer std_logic;
 ras,cas,we: out std_logic);
end refreq;

architecture behavioral of refreq is
signal res: std_logic;
signal q: std_logic_vector(8 downto 0):="000000000";
begin

A: process (clock,q,res)
  begin
  if (clock='0' and clock'event) then
    if res='1' then
     q<="000000000";
      res<='0';
    else
      q <= q + 1;
    end if;
    if q="110100011" then    --cuenta de 1 hasta 419
      res<='1';
    end if;
  end if;
end process A;

B: process (q)
```

```
 begin
  if q>="110100000" then          --si q>=416 y
        if q<="110100011" then  --   q<=419
         ref<='0';                      --ref es 0
        else
   ref<='1';                --sino ref es 1
        end if;
       else
   ref<='1';
  end if;
end process B;

-- mux 6 a 3 de las lineas de control
C: process (ref)
 begin
  if ref='0' then
      ras<=ras1;
                  cas<='1';
                  we<='1';
  elsif ref='1' then
                  ras<=ras1;
                  cas<=cas1;
                  we<=wr;
  end if;
end process C;

end behavioral;
```