Sistemas Informáticos
Curso 2005-2006

Desarrollo de un entorno hardware para la ejecución de aplicaciones de propósito general sobre FPGAs

Laura Sánchez Conde
Raquel Sánchez Delgado
Jose Antonio Valero Martín

Dirigido por:
Prof. Hortensia Mecha López
Dpto. de Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

Tabla de contenidos

Autorización 4
Resumen del proyecto 5
Project outline 6
Objetivo del proyecto 7
Diagramas de ejecución 8
Carga inicial 8
Lanzamiento a ejecución de una nueva tarea 9
Desarrollo del sistema 9
Capítulo 1: Entorno de trabajo 10
1.1. Field Programmable Gate Array (FPGA) 10
1.1.1. Estructura o arquitectura de las FPGAs 11
1.1.2. Metodología de diseño 12
1.1.3. Placa de trabajo 13
1.1.4. Aplicaciones típicas 13
1.2. FPGAs utilizadas en el proyecto 14
1.2.1. Spartan 2 14
1.2.2. Virtex II 15
1.2.3. Virtex II Pro 16
1.3. Placas de prototipado 17
1.3.1. XSA100 17
1.3.2. Multimedia Board 18
1.3.3. XUP 18
1.4. Analizador digital: 16702B Logic Analysis System 20
1.5. VHDL 22
Capítulo 2: Memoria RAM 24
2.1. Memoria ZBT SRAM (Virtex II) 24
2.2. Memoria DDR SDRAM (Virtex II Pro) 26
2.2.1. Conceptos 28
2.2.2. Operaciones 33
2.2.3. Controlador DDR SDRAM 40
2.2.3.1. Componentes 40
2.2.3.2. Forma de uso 43
2.3. DCM (Digital Clock Manager) 50
Capítulo 3: Herramientas software 56
3.1. Xilinx ISE 56
3.1.1. Project Navigator 57
3.1.2. ModelSim 63
3.2. iMPACT 65
3.3. MIG 007 66
Capítulo 4: Desarrollo del proyecto 70
Capítulo 5: Arquitectura Final 85
5.1. Kernel del Sistema Operativo 91
5.1.1. Unidad de control 92
5.1.2. Intérprete de órdenes 97
5.1.3. Lector de datos 105
5.1.4. Ejecutor de órdenes 111
5.2. Interfaz de memoria 120
5.3. Planificador 130

5.4. Drivers de Entrada/Salida 140
5.4.1. Teclado 141
5.4.2. VGA 144
5.5. Generador de señales de reloj 146
5.6. Ejemplos de funcionamiento 148
Capítulo 6: Conclusiones 157
Apéndices 158
A. Referencias 158
B. Glosario de términos 159
C. Índice de figuras 163
D. Índice de tablas 166
D. Palabras clave 168
F. Código del proyecto 169

Autorización

Los autores de este proyecto, Laura Sánchez Conde, Raquel Sánchez Delgado y Jose Antonio Valero Martín, autorizan mediante este texto a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos no comerciales, y mencionados expresamente a sus autores tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo. Laura Sánchez Conde	Fdo. Raquel Sánchez Delgado	Fdo. Jose Antonio Valero Martín

Resumen del proyecto

El hardware dinámicamente reconfigurable ofrece la posibilidad de dar a un único dispositivo hardware cualquier tipo de funcionalidad deseada por el usuario. Esto por ejemplo permitiría que nuestro dispositivo fuese hoy una calculadora y mañana un potente microprocesador.

Existen varios tipos de dispositivos hardware dinámicamente reconfigurables de entre los cuales nosotros utilizaremos las FPGAs.

La forma de trabajar con las FPGAs es diseñar mediante herramientas software un circuito hardware, transformarlo en un mapa de bits que es la representación física del circuito dentro de la FPGA y por último cargar dicho mapa de bits en la propia FPGA. Una vez hecho esto la FPGA se comportará de la misma forma que el circuito que se diseñó.

Nuestro proyecto consiste en ir un paso más allá del modo de funcionamiento normal de este tipo de dispositivos, ya no se tratará de cargar un único circuito en la FPGA sino que podremos tener varios circuitos diferentes almacenados de manera externa y el usuario podrá elegir cual es el que quiere ejecutar en cada momento.

Para que el usuario del sistema pueda elegir la tarea que desea ejecutar en cada momento hemos desarrollado un entorno sobre el cual se pueden introducir las órdenes por teclado y visualizar los resultados obtenidos en un monitor.

El entorno permite la carga y ejecución de hasta cuatro tareas de manera simultánea e independiente aunque el usuario tiene la posibilidad de lanzar todas las que desee en cualquier momento.

Las tareas cargadas permanecerán en ejecución hasta que finalicen, por lo que si el usuario lanzara más de cuatro tareas, éstas quedarán a la espera de que alguna de las que haya en ejecución termine, liberando así su zona de trabajo y permitiendo que una nueva tarea pase a ejecución.

El proceso consistente en cargar varias tareas sobre una misma FPGA en cualquier momento y sin que tengan porqué interferir al funcionamiento del resto, es la denominada reconfiguración parcial. Esta reconfiguración parcial es una técnica desarrollada recientemente que permite no sólo que un sistema varíe su comportamiento a lo largo del tiempo, de manera aleatoria o por voluntad de un usuario, sino que podría darse el caso de que el propio sistema sea capaz de determinar cual de sus partes no está siendo utilizada en un determinado momento, pudiendo así desactivarlos para ahorrar energía o disminuir la temperatura global del sistema. De forma contraria, también podría decidir si necesita más recursos de un cierto tipo, bien sea duplicando alguno ya existente o bien cargando un módulo nuevo que permita resolver el problema en un menor tiempo o con una mayor eficiencia o complejidad.

Project outline

The dynamically reconfigurable hardware offers the possibility that a hardware device changes its functionality depending on the user requirements. This for example would allow that our device was today a calculator and tomorrow a powerful microprocessor.

There are many types of dynamically reconfigurable hardware devices and we will use the FPGAs.

The way to work with the FPGAs consists in the design with software tools of a hardware circuit, to transform it in a bit map that is the physical representation of the circuit within the FPGA and finally to load this bit map in the configuration memory of the FPGA. Then the FPGA will behave like the circuit designed.

Our project tries to go far away of the normal function mode of this type of devices. The idea is to have several stored circuits and the user will be able to choose the circuit that he wants to execute at every moment.

In this way we have developed an environment on which the user can write commands using the keyboard and the system shows the results in a monitor.

The environment allows the user to load and run simultaneous a maximum of four tasks. However the user has the possibility of throws all the tasks that he wants in any moment.

The loaded tasks will remain in execution until they finalize. Then if the user throws more than four tasks, these tasks will be stopped until any task in execution finishes, allowing a new task to start the execution.

The process of loading several tasks on a same FPGA, in any moment and without they have to interfere to the operation of the other, it is the denominated partial reconfiguration. This partial reconfiguration is a technique recently developed that allows a system to change its behaviour throughout the time. It is also possible that the system would determine which of its parts is not being used at a certain moment, in that case the system would deactivate them to save energy or to diminish the globe temperature of the system. By opposite, the system also can decide if it needs more resources of a type and in that case it can duplicate one of the existing or load a new module that allows to solver the problem in a smaller time or with a greater efficiency or complexity.

Objetivo del proyecto

El objetivo fundamental del proyecto es la generación de un entorno para la ejecución de tareas sobre dispositivos dinámicamente reconfigurables.

Dicho entorno será puramente hardware y estará formado por una serie de módulos siguiendo el esquema:

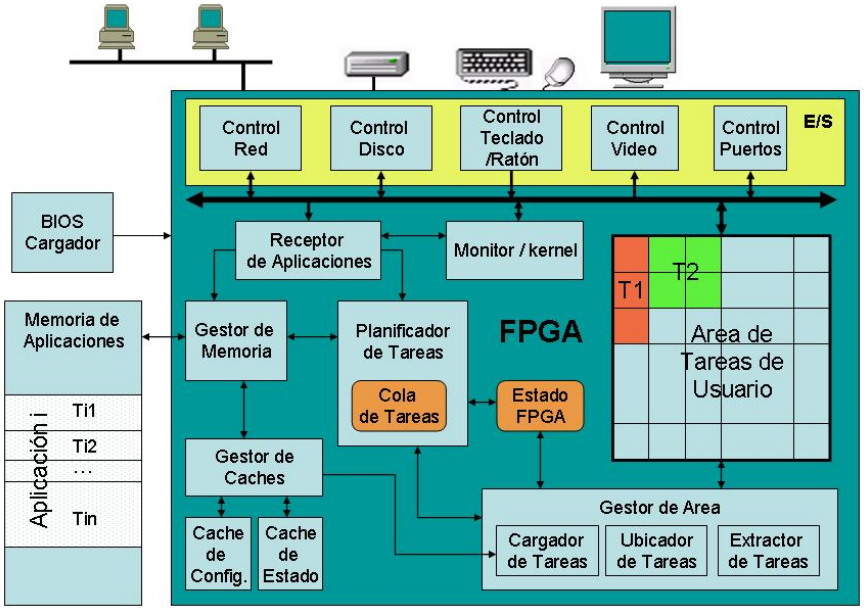


Figura 1: Esquema general del sistema

- Interfaz de comunicaciones: Es el módulo encargado de realizar la comunicación con los dispositivos externos de la FPGA. En nuestro caso estos dispositivos pueden ser de entrada como el teclado, de salida como el controlador de video y de entrada/salida como la memoria. Para cada dispositivo habrá un módulo controlador encargado de permitir la comunicación con él.

- Kernel: Este módulo controla el funcionamiento general del sistema. Es el encargado de recoger las órdenes introducidas por el usuario a través del teclado, procesarlas y generar todas las señales necesarias para poder llevar a cabo su ejecución.

- Planificador de tareas: Con este módulo el sistema es capaz de gestionar tanto las tareas nuevas que se van a cargar en memoria, las que se almacenan en cola hasta que puedan ser cargadas en la FPGA y eliminar las tareas que ya hayan finalizado su ejecución dejando espacio libre para que una nueva tarea pueda ser ejecutada.

- Cargador de tareas de inicio: Este es el módulo encargado de llevar a memoria las tareas almacenadas en un dispositivo de almacenamiento no volátil. Esta carga se realizará una sola vez al inicio de la ejecución del sistema. Este componente basa su funcionamiento a los módulos de interfaz de comunicaciones y receptor de aplicaciones.

Diagramas de ejecución

A continuación vamos a detallar alguna de las ejecuciones típicas soportadas por el sistema.

Carga inicial

Consiste en llevar las tareas desde un dispositivo de almacenamiento no volátil hasta la memoria principal para poder ser utilizadas.

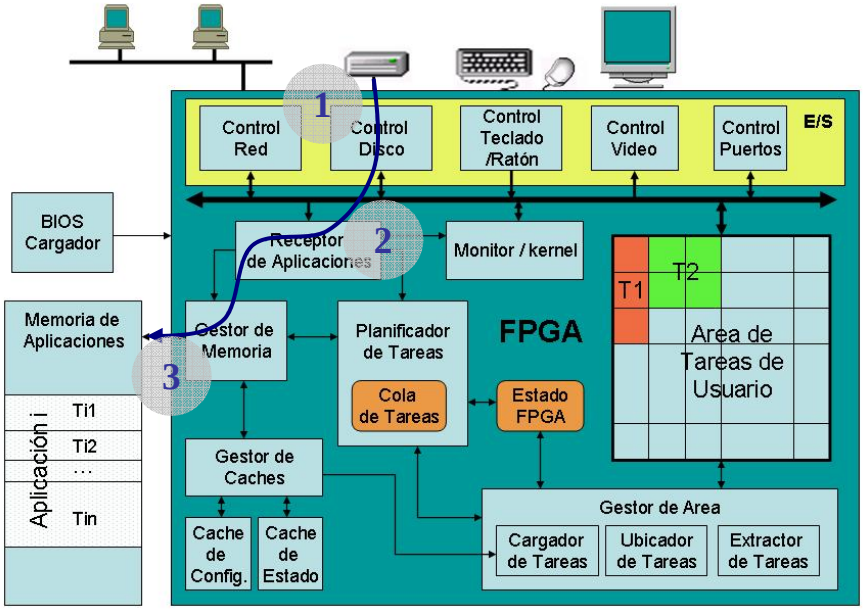


Figura 2: Ejecución de la carga inicial

Lanzamiento a ejecución de una nueva tarea

El usuario mediante la orden correspondiente le indicará al sistema que desea lanzar a ejecución una nueva tarea y éste procederá a leer de la memoria la tarea indicada, ponerla en una cola de ejecución y cargarla sobre un hueco de la FPGA cuando sea necesario.

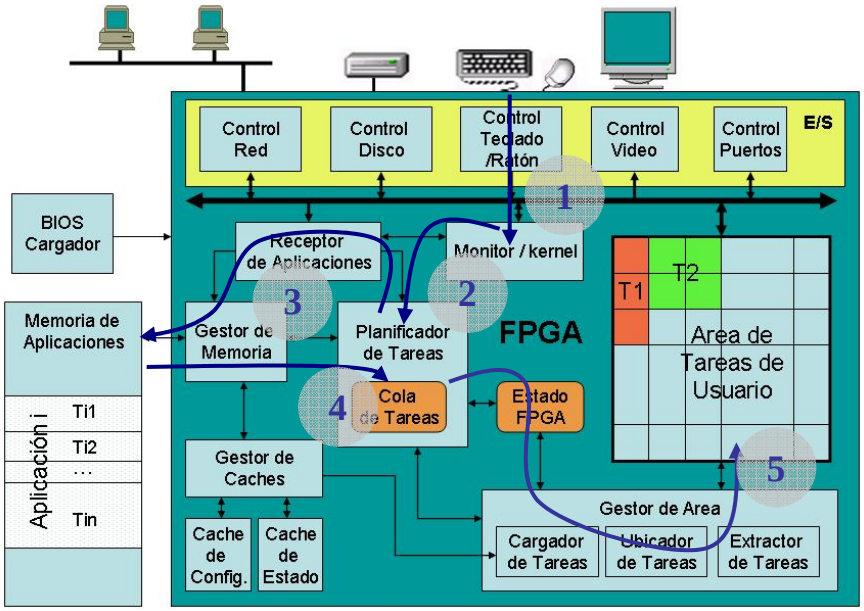


Figura 3: Ejecución del lanzamiento de una tarea

Desarrollo del sistema

El sistema se desarrollará en dos fases:

- En una primera fase se desarrollará una arquitectura básica en hardware capaz de ejecutar una única tarea de usuario. Deberá incluir los elementos básicos de un sistema operativo como son los controladores de entrada/salida y un monitor de órdenes que sea capaz de interpretar las órdenes básicas del usuario.

- En la segunda fase habría que extender la arquitectura para permitir la carga y ejecución simultánea de múltiples tareas de usuario. El área de la FPGA que puede ser utilizada para la ejecución de tareas estará dividida en cuatro partes en cada una de las cuales puede haber una única tarea. Además habrá que diseñar un planificador/cargador de tareas que se encargue de gestionar dicha área siguiendo una política de planificación tipo FIFO.

Capítulo 1: Entorno de trabajo

1.1. Field Programmable Gate Array (FPGA)

FPGA es el acrónimo de Field Progammable Gate Array (matriz de puertas programable).

Las FPGAs pertenecen al conjunto de hardware reconfigurable, es decir, el dispositivo se recibe como un circuito fabricado cuya funcionalidad podremos programar. Existen otros dispositivos electrónicos digitales programables de muy alta densidad como son PLA, PROM, PAL, CPLD…

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

Las ventajas que ofrecen las FPGAs frente a los ASIC (circuitos integrados de aplicación específica), son que mientras que la metodología de diseño de arquitecturas ASIC obliga al uso de herramientas de diseño asistido por computadora, laboratorios de fabricación y simulación complejos, el diseño de un FPGA puede realizarse en el momento y se puede contar con un sistema de prueba para su verificación sin necesidad de esperar la fabricación de prototipos. Además, en el diseño de un ASIC es importante no cometer errores, ya que una vez fabricado el dispositivo no se pueden corregir tales errores, teniendo que volver a rediseñar y fabricar el dispositivo, haciendo una gran inversión de tiempo y dinero; mientras que el diseño de las FPGAs es modificable dado que estos dispositivos se configuran a partir del contenido de una memoria o un fichero, una modificación que puede realizarse en pocas horas y con menos riesgo que en el caso de sistemas basados en diseños ASIC. **[R1]**

1.1.1. Estructura o arquitectura de las FPGAs

Los elementos básicos que forman parte de una FPGA son:

- CLB, bloque configurable lógico (configurable logic block).
- IOB, bloque de entrada/ salida (input /output block)
- Bloques de interconexión

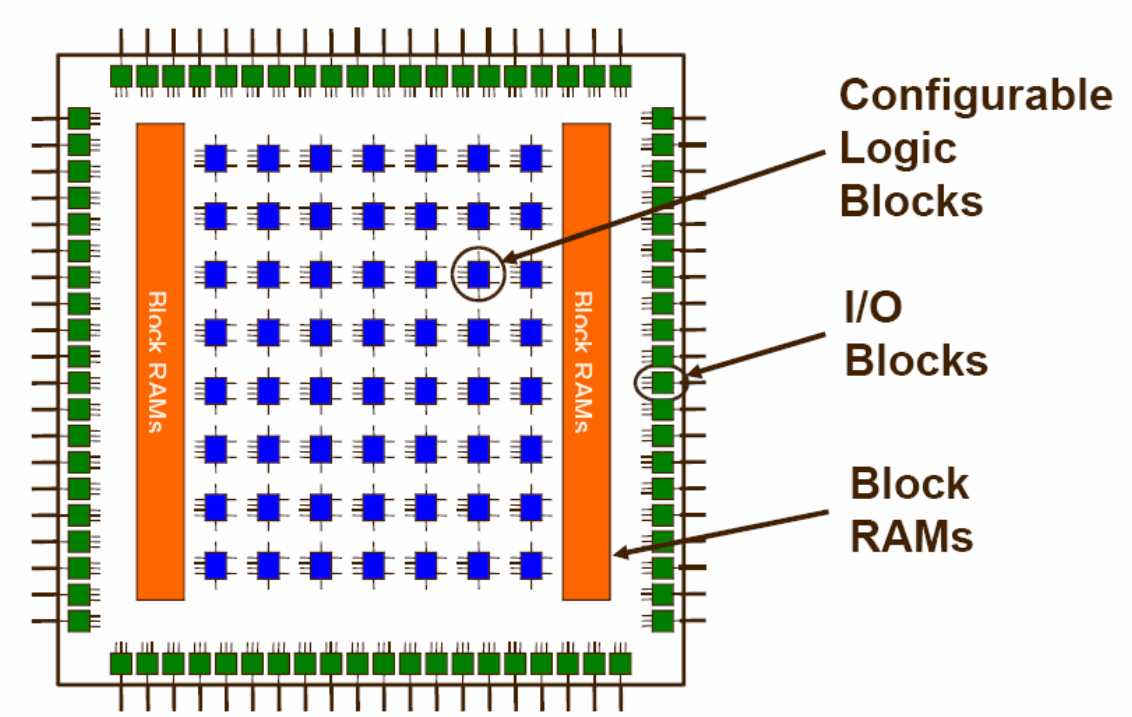


Figura 4: Estructura interna de una FPGA [R2]

Internamente una FPGA es una serie de pequeños dispositivos lógicos o CLBs dispuestos sobre el silicio. Estos dispositivos lógicos se organizan como una array de celdas establecido en filas y columnas.

Los CLBs contienen en su interior elementos hardware programables que permiten que su funcionalidad sea elevada. También es habitual que contengan dispositivos de memoria.

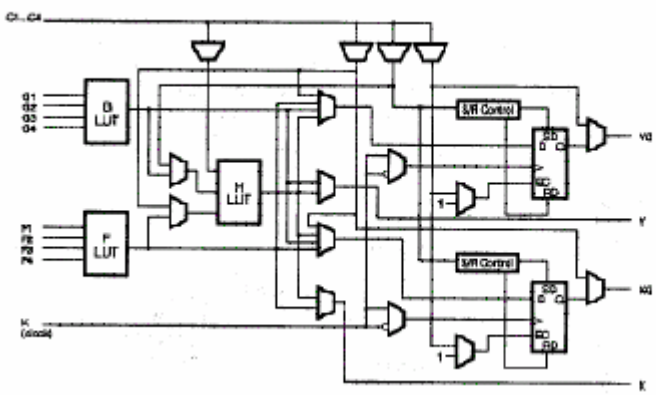


Figura 5: Estructura de un CLB

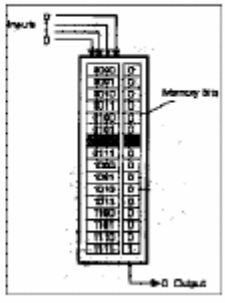


Figura 6: Estructura de una LUT

Para comunicarnos con el exterior existen elementos, también configurables colocados cerca de cada una de las patillas del chip, denominados IOB, estas celdas de entrada/salida tienen características programables para poder definir la forma de trabajo (entrada, salida, entrada/salida…) y están colocadas rodeando la matriz de CLBs.

La forma de conectar CLBs entre sí o con otras partes de la FPGA (entre las que están los IOBs) es a través de una estructura de interconexión. Ésta está formada por un gran número de bloques de interconexión que también pueden programarse y que pueden poseer distintas velocidades.

Esta estructura de interconexión está formada por un canal de rutado vertical y otro horizontal, a los que se pueden conectar cualquier CLB próxima mediante un punto de interconexión. Ambos canales de rutado se unen a través de una matriz de conmutación PSM (programmable switch matrix).

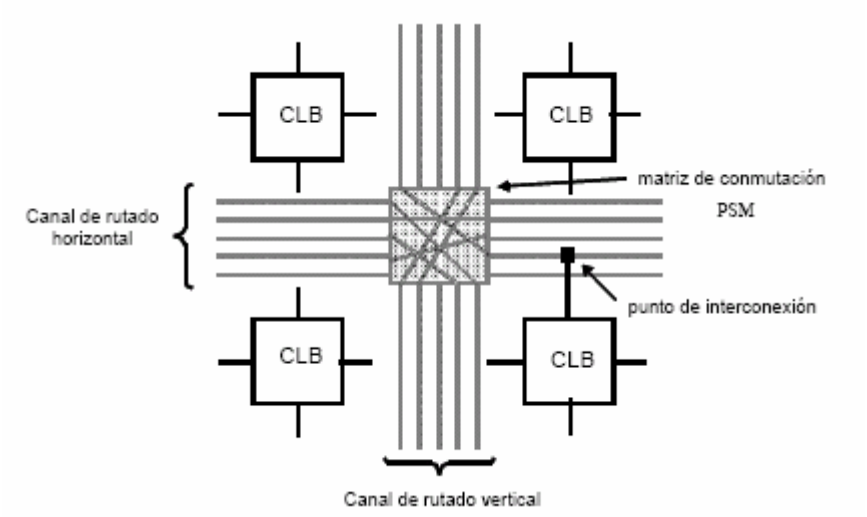


Figura 7: Estructura de interconexión de FPGA

Además de estos elementos también hay elementos de control o configuración que sirven para controlar los multiplexores, implementar LUTs (Look Up Table) y configurar el interconexionado.

Aparte de esta estructura, que es la básica, cada fabricante añade sus propias ideas, por ejemplo hay algunos que tienen varios planos con filas y columnas de CLBs; el diseño físico y la fabricación es independiente del diseño particular de cada fabricante.

1.1.2. Metodología de diseño

Los diseños no se fabrican, sino que se realizan programando adecuadamente los CLBs, IOBs y bloques de interconexión. El mapa de bits sirve para definir la función lógica que realizará cada uno de los CLB, seleccionar el modo de trabajo de cada IOB e interconectarlos todos.

Una FPGA es programable a nivel hardware, por lo que proporciona las ventajas de un procesador de propósito general y un circuito especializado.

El diseñador cuenta con la ayuda de herramientas de programación. Cada fabricante suele tener las suyas, aunque usan unos lenguajes de programación comunes. Estos lenguajes son los HDL o *Hadware Description Language* (lenguajes de descripción de hardware):

- VHDL
- Verilog
- ABEL

Cada bloque almacena su configuración en un dispositivo de memoria y dependiendo del método de almacenaje, el diseño volcado sobre la FPGA será o no volátil.

1.1.3. Placa de trabajo

Las FPGAs están soportadas sobre placas de trabajo que aportan a la FPGA salidas/entradas de video, audio, VGA, USB, ethernet… también aportan otros elementos como pueden ser switches, pushbutton o leds.

1.1.4. Aplicaciones típicas

Teniendo en cuenta que algunas de las características de las FPGA son su flexibilidad, capacidad de procesado en paralelo y velocidad; esto les convierte en dispositivos idóneos para:

- Simulación y depuración en el diseño de microprocesadores.
- Simulación y depuración en el diseño de microcontroladores.
- Procesamiento de señal digital, por ejemplo vídeo.
- Sistemas aeronáuticos y militares.
- Módulos de comunicaciones.

1.2. FPGAs utilizadas en el proyecto

Durante el desarrollo del proyecto hemos usado las siguientes FPGAs: Spartan 2, Virtex II y Virtex II Pro, que pasamos a comentar con mayor detalle en los puntos que vienen a continuación.

1.2.1. Spartan 2

La Spartan 2 soporta la mayoría de los estándares de entrada/salida, incluyendo aquellos que están optimizados para interfaces de memoria de alta velocidad.

La ventaja que se deriva de su uso, es la mejor relación coste/flexibilidad en soluciones de diseño que adopten arquitecturas FPGA, minimizando el riesgo y el tiempo de lanzamiento de nuevos productos al mercado.

El modelo de la FPGA, dentro de la familia Spartan 2, que usamos fue el XC2S100 que está formada por 2700 celdas lógicas, 100000 puertas del sistema, tanto lógicas como RAM, un array CLB de 20 filas por 30 columnas, que hacen un total de 600 CLBs, y 176 bloques IOB.

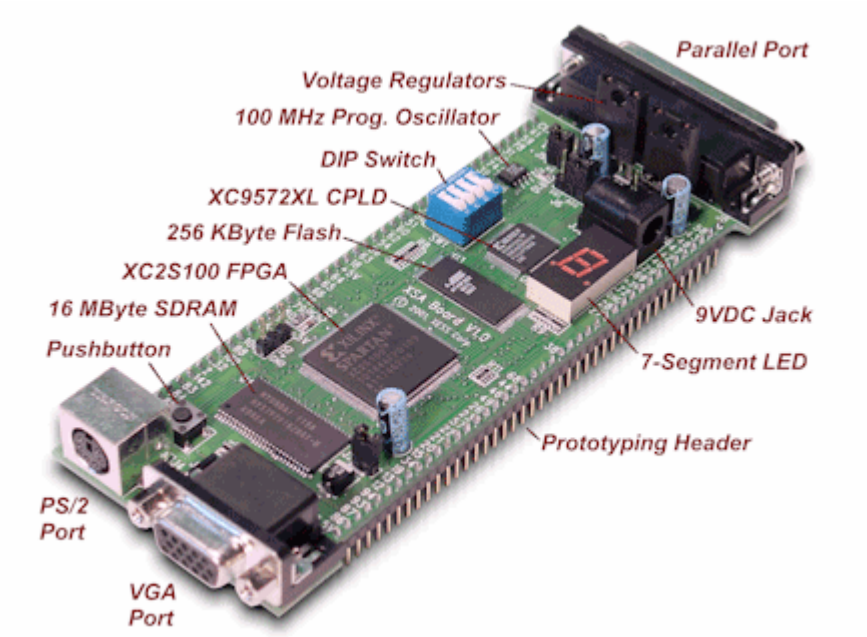


Figura 8: Placa de extensión de Spartan 2

1.2.2. Virtex II

Las FPGAs de la familia Virtex pertenecen a las familias de alto rendimiento de Xilinx y ofrecen muchas más posibilidades de configuración y *readback* que las anteriores generaciones de FPGAs de Xilinx. Una de las principales innovaciones es la posibilidad de realizar una *reconfiguración parcial* del dispositivo de forma *dinámica*, es decir, mientras está funcionando. Para conseguir realizar una reconfiguración parcial, partiremos de un *bitstream o mapa de bits* básico que será modificado. Inicialmente, la Virtex se configura con este mapa de bits y, cuando queremos realizar un cambio parcial en la configuración del dispositivo, simplemente se carga en la memoria de configuración la parte del *bitstream* que ha sido modificada. Se consiguen reconfigurar los recursos deseados ya que los *bitstreams* de configuración contienen una mezcla de comandos (indican los recursos que deben ser configurados) y de datos de configuración.

Para entender esto, es necesario conocer la arquitectura interna de la Virtex y sus modos de programación. En la figura que sigue se puede observar el esquema interno de un dispositivo de la familia Virtex:

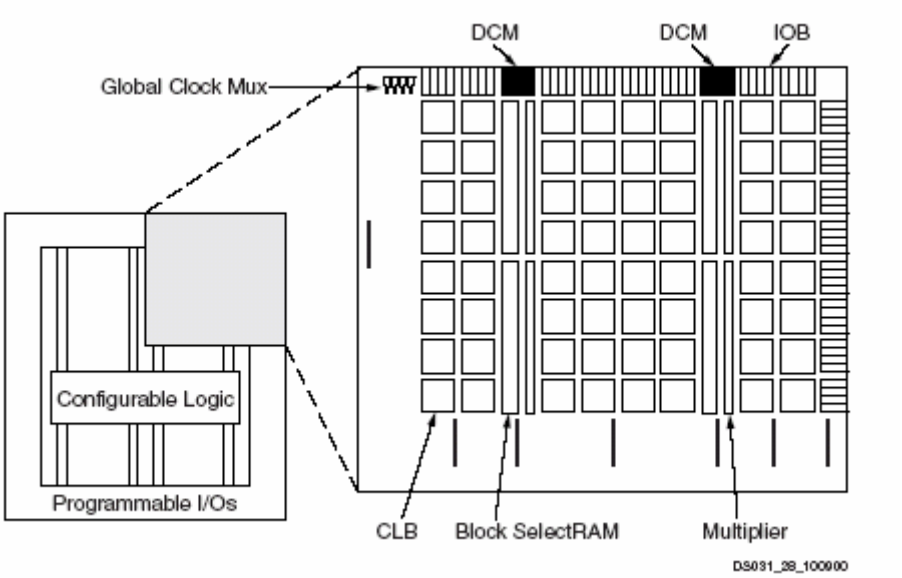


Figura 9: Arquitectura interna de Virtex

La lógica interna configurable incluye principalmente 4 elementos organizados en un array regular:

-Bloques lógicos confugurables (CLBs).

-Bloque SelectRAM, provee un amplio número de elementos de almacenamiento de 18 KBit de doble puerto RAM.

-Bloques multiplicadores de 18 bits x 18 bits

-DCM (Digital Clock Manager), que se comenta en el Capítulo 2.

La configuración de las FPGAs Virtex está basada en memorias SRAM donde se cargan los datos de configuración. De esta forma, la memoria de configuración se puede ver como una matriz rectangular de bits que se agrupan en líneas verticales llamadas tramas. Así, una trama es la porción mínima de memoria que puede ser reconfigurada. Las tramas se agrupan formando unidades mayores llamadas columnas. Cuando el dispositivo vaya a ser reprogramado, el *mapa de bits* contendrá una parte de direccionamiento, indicando de esta manera qué trama debe ser reconfigurada en la memoria de configuración.

Para conseguir una reconfiguración parcial dinámica de la Virtex se utilizará el modo SelectMAP que además es el modo más rápido de configuración ya que se cargan 8 bits cada ciclo de reloj. La Virtex se puede programar de forma que los pines del interfaz SelectMAP se mantengan para tareas de configuración.

La FPGA que hemos usado es la Virtex 2 XCV2000-FF896, sobre la placa "Multimedia", que está formada por una array de CLBs de 56x48, 10752 slices, un máximo de 624 bloques I/O, 56 bloques RAM de 18 KB, 56 bloques Multiplier. **[R2]**

1.2.3. Virtex II Pro

La familia Virtex II Pro incorpora las ventajas del alto rendimiento de las FPGAs y el núcleo de IBM PowerPC, todo integrado en un mismo producto y disponible a un significativo menor coste. [R2]

Las FPGAs Xilinx Virtex II Pro se desarrollan como una nueva plataforma para su uso en comunicaciones, almacenamiento, y aplicaciones de consumo. Además su arquitectura constituye la base de construcciones escalables de la próxima generación de sistemas programables. **[R3]**

La reconfiguración parcial de los dispositivos FPGA anteriores a la familia Virtex II Pro se realiza mediante el modo esclavo SelectMAP o el modo Boundary Scan (JTAG). Sin embargo, estos modos de reconfiguración requieren de un controlador externo para enviar los datos de reconfiguración. Con la llegada de las familias Virtex II Pro se incluyó una interfaz interna de reconfiguración que se conoce como ICAP. La interfaz ICAP incluye un conjunto reducido de la interfaz SelectMAP debido a que no se emplea para realizar configuraciones completas y no tiene soporte para diferentes modos de configuración.**[R4]**

Hemos usado la FPGA XC2VP30 de la familia Virtex 2 PRO soportada sobre la placa de prototipado XUP.

Cabe destacar que la FPGA está compuesta por 13969 slices, un array de CLBs de tamaño 80x46, RAM distribuidas de 428 KB, bloques RAM de 2448KB, 136 bloques *multiplier*, 8 DCMs y dos núcleos Risc PowerPC. **[R5]**

1.3. Placas de prototipado

Las FPGAs están soportadas sobre placas de trabajo o de prototipado que aportan a la FPGA salidas/entradas de vídeo, audio, VGA, USB, ethernet… también aportan otros elementos como pueden ser switches, pushbutton o leds.

1.3.1. XSA100

La placa base que soporta la FPGA es la XSA-100 v1.2 y la de expansión es XST v1.3.2. La placa base posee una memoria SDRAM de 16 MB, Flash 256 KB, CPLD modelo XC9572XL, salida de audio, salida VGA de 64 colores, puerto PS2, 12 switches, 5 pulsadores, 10 leds, 3 displays de 7 segmentos [RDAS]

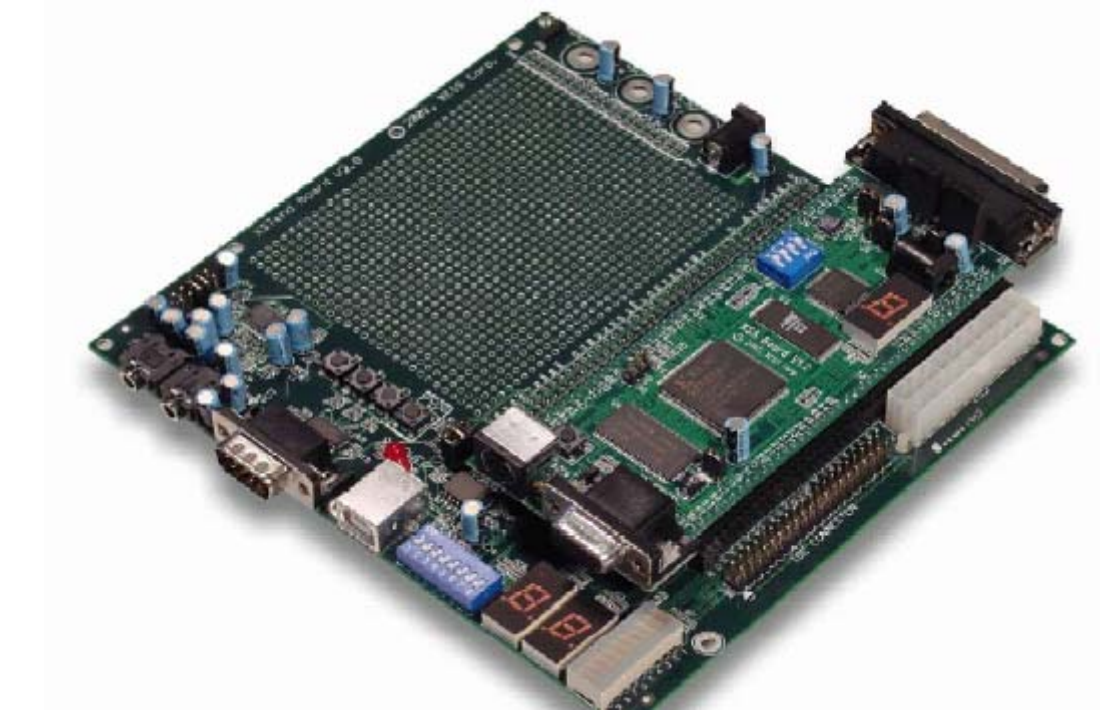


Figura 10: FPGA Spartan 2, placa de prototipado XSA-100

1.3.2. Multimedia Board

La placa sobre la que está es la placa Multimedia, que posee una Compact Flash de 128 MB. Algunas de las características de la placa base son tener salida de audio, entrada/ salida de supervídeo, puerto ethernet 10/100, puerto para teclado y ratón, salida de altavoz con volumen ajustable, 10 pulsadores de entrada para el usuario y dos switches con 2 leds. [R2]

Figura 11: Foto Virtex II Multimedia

1.3.3. XUP

Hemos usado la FPGA XC2VP30 de la familia Virtex 2 PRO soportada sobre la placa de prototipado XUP Virtex-II Pro Development System

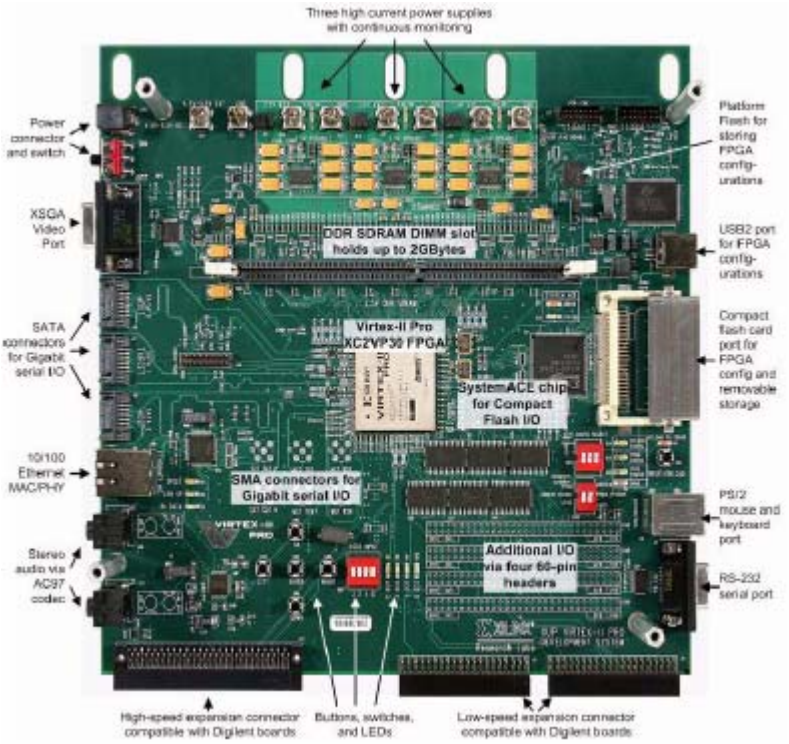


Figura 12: Foto Virtex 2 Pro sobre placa XUP [R5]

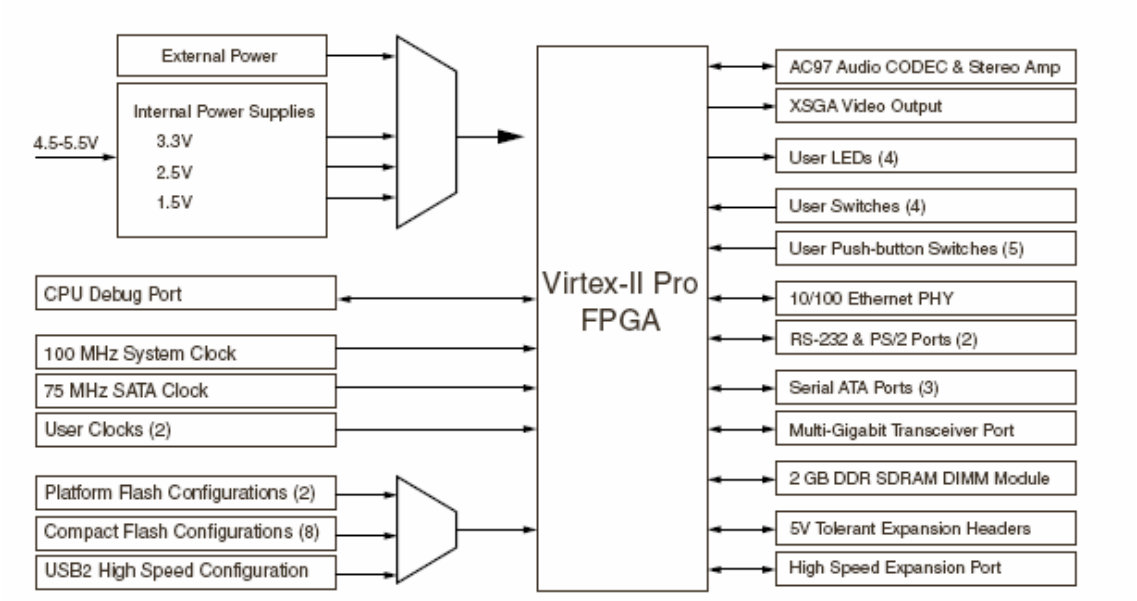


Figura 13: Conexión Virtex 2 Pro con puertos [R5]

Como se puede observar en las figuras anteriores la FPGA Virtex 2 PRO está soportada sobre una placa base XUP, que contiene una memoria DDR SDRAM de 64MB, un puerto Ethernet 10/100, 8 pins de entrada/ salida, una memoria flash de 8 MB, 2 puertos serie PS/2, 4 leds, 4 switches y además posee salida de audio, puerto XSGA video, puerto USB, puerto para conectar otras tarjetas y conectores SATA entre otras características.

1.4. Analizador digital: 16702B Logic Analysis System

El sistema de análisis lógico Agilent 16702B combina una pantalla táctil con una interfaz con ventanas. Una de las ventajas de la gran pantalla que posee es que se pueden ver más formas de onda o estados al mismo tiempo.

La potencia, flexibilidad y fácil uso de este analizador ayuda a solucionar más rápidamente problemas de diseño e integración. Además también se incluye un ratón y teclado en el estándar.

Figura 14: Foto de 16702B [R6]

Algunas de las características del 16702B son el acceso rápido a través del teclado a formas de onda, sistema, ayuda y gestores de archivos de Windows, poseer pantalla táctil de 800x600, ratón, teclado, un disco duro de 18 GB, memoria RAM de 128 MB y CD-ROM [R8].

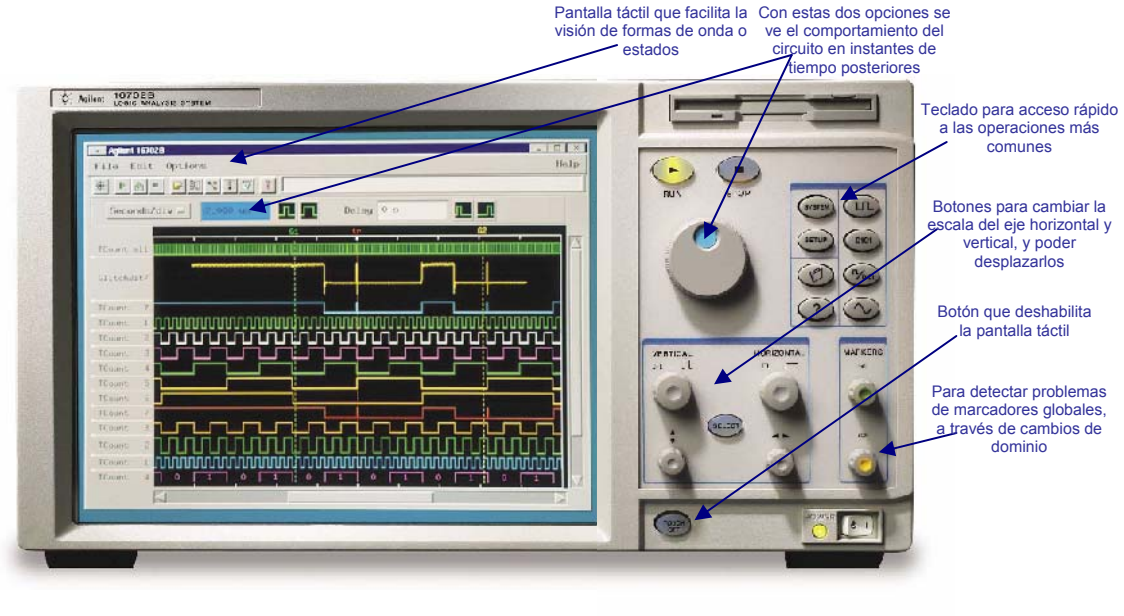


Figura 15: Algunas características del 16702B [R7]

El 16702B se puede comportar como un osciloscopio, puede generar patrones y realizar análisis de tiempo y estados [R9].

Figura 16: Muestra del comportamiento del sistema dado por la serie 16700 [R7]

El uso que principalmente le dimos fue el análisis de la forma de onda de las señales y de los estados.

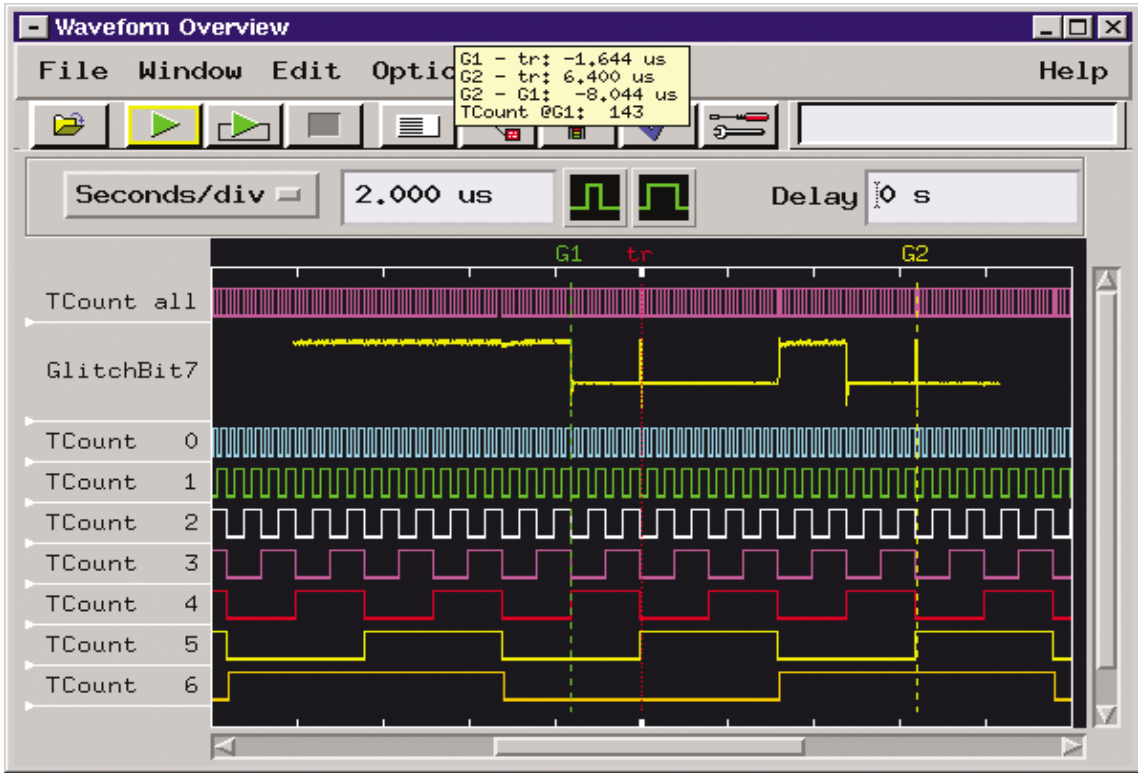


Figura 17: Muestra de un análisis de la forma de onda de señales [R7]

1.5. VHDL

El lenguaje que vamos a usar para la descripción de los circuitos es el VHDL (VHSIC Hardware Description Languaje).

El VHDL es un lenguaje de descripción hardware que fue desarrollado como estándar de IEEE, lo que favoreció su adopción en la industria lo que se ve reflejado en las constantes mejoras en las herramientas. Debido a su estandarización, un código en VHDL puede ser portado a diferentes herramientas y también, puede ser reutilizado en diferentes diseños.

Este lenguaje proporciona una sintaxis amplia y flexible que permite el modelado estructural, en flujo de datos y de comportamiento de hardware.

Dentro del VHDL hay varias formas con las que podemos diseñar el mismo circuito y es tarea del diseñador elegir la más apropiada:

- Funcional: describimos la forma en que se comporta el circuito. Esta es la forma que más se parece a los *lenguajes de software* ya que la descripción es secuencial.

- Flujo de datos: describe asignaciones concurrentes (en paralelo) de señales.

- Estructural: se describe el circuito con instancias de componentes. Estas instancias forman un diseño de jerarquía superior, al conectar los puertos de estas instancias con las señales internas del circuito, o con puertos del circuito de jerarquía superior.

- Mixta: combinación de todas o algunas de las anteriores.

En VHDL también existen formas metódicas para el diseño de máquinas de estados, filtros digitales, bancos de pruebas etc.

Flujo de diseño

El flujo de diseño de un sistema implementado en VHDL podría ser:

- División del diseño principal en módulos separados. La modularidad es uno de los conceptos principales de todo diseño.

- Implementación de los módulos del diseño

- Simulación funcional, es decir, comprobaremos que lo escrito en el punto anterior realmente funciona como queremos, si no lo hace tendremos que modificarlo. En este tipo de simulación se comprueba que el código VHDL ejecuta correctamente lo que se pretende.

- Síntesis. En este paso se adapta el diseño anterior (que sabemos que funciona) a un hardware en concreto. Hay sentencias del lenguaje VHDL que no son sintetizables, por lo que hay expresiones que no pueden ser transformadas a circuitos digitales. Durante la síntesis se tiene en cuenta la estructura interna del dispositivo, y se definen

restricciones, como la asignación de pins. El sintetizador optimiza las expresiones lógicas con objeto de que ocupen menor área, o bien son eliminados las expresiones lógicas que no son usadas por el circuito.

- Simulación post-síntesis. En este tipo de simulación se comprueba que el sintetizador ha realizado correctamente la síntesis del circuito, al transformar el código HDL en bloques lógicos conectados entre sí.

- Placement y routing. El proceso de placement consiste en situar los bloques digitales obtenidos en la síntesis de forma óptima, tratando de que aquellos bloques que se encuentran muy interconectados entre si se sitúen próximamente. El proceso de routing consiste en rutar adecuadamente los bloques entre sí, intentando minimizar retardos de propagación para maximizar la frecuencia de funcionamiento del dispositivo.

- Back-annotation. Una vez ha sido completado el placement & routing, se extraen los retardos de los bloques y sus interconexiones, con objeto de poder realizar una simulación temporal (también llamada simulación post-layout).

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

- Programación en el dispositivo. Se vuelca el diseño en el dispositivo final y se comprueba el resultado.

Capítulo 2: Memoria RAM

La memoria RAM (*Random Access Memory* o Memoria de Acceso Aleatorio) es uno de los componentes más importantes de los actuales equipos informáticos, y su constante aumento de la velocidad y capacidad ha permitido a los computadores crecer en potencia de trabajo y rendimiento.

Se trata de una memoria de semiconductor en la que se puede tanto leer como escribir. Es una memoria volátil, es decir, pierde su contenido al desconectar la energía eléctrica. Se utiliza normalmente como memoria temporal de trabajo, bien sea para almacenar datos o los programas que se están ejecutando.

En este tipo de memorias se puede acceder de manera directa a la palabra deseada, que generalmente suele ser un byte, sin que dependa de la palabra leída o escrita en momentos anteriores. Su tiempo de acceso tampoco depende a la posición accedida sino que es el mismo para todos los casos.

Las RAMs se dividen en estáticas y dinámicas. Una memoria RAM estática (SRAM) mantiene su contenido inalterado mientras esté alimentada. La información contenida en una memoria RAM dinámica (DRAM) se degrada con el tiempo, llegando ésta a desaparecer, a pesar de estar alimentada. Para evitarlo hay que restaurar la información contenida en sus celdas a intervalos regulares, operación denominada refresco. Existe además un tipo de memoria que duplica el rendimiento ya que permiten leer y escribir el doble de datos por ciclo de reloj. Estas memorias son las denominadas DDR (Double Data Rate).

Durante el desarrollo de nuestro proyecto hemos trabajado básicamente con dos tipos de memoria RAM que son la memoria SRAM y la memoria DDR, que se encontraban en la placa de prototipado de la Virtex II y Virtex II Pro respectivamente.

2.1. Memoria ZBT SRAM (Virtex II)

Los dispositivos ZBT SRAM (Zero Bus Turnaround) son memorias SRAM síncronas que proporcionan un gran rendimiento ya que utilizan todos los ciclos de reloj. Como su propio nombre indica este tipo de memorias no requieren ciclos de espera entre operaciones de lectura y escritura, por lo que se aprovecha al máximo el ancho de banda. Por esto, las memorias ZBT proporcionan buenos rendimientos en aplicaciones que realizan numerosos accesos aleatorios en los que mezclan operaciones de lectura y escritura, en oposición a largas ráfagas de lecturas/escrituras.

La placa de prototipado en la que está incluida la FPGA Virtex II tiene un total de cinco módulos independientes de ZBT SRAM de un tamaño de 512K x 32. Funcionan a una frecuencia máxima de 130 MHz y a pesar de soportar un bus de 36 bits de datos, las limitaciones del *pinout* de la FPGA obligan a utilizar 4 como bits de paridad por lo que el ancho se reduce a 32. Las señales de control, direcciones, buses de datos y el reloj son independientes en cada uno de los bancos y no existen señales compartidas entre ellos. Las señales de reloj de los cinco bancos tienen la misma longitud, además hay una señal de reloj especial de realimentación que se usa para alinear los relojes de los pines del dispositivo con un reloj interno de la FPGA. Este tipo de memorias soportan la

lectura/escritura en ráfaga pero en los módulos incluidos en la placa únicamente está permitido el uso de ráfagas lineales.

Dentro de la FPGA, los relojes son distribuidos usando árboles de reloj que aseguran que las señales de reloj lleguen a los flip-flop simultáneamente o al menos con el mínimo skew. Si las entradas de reloj de las memorias ZBT salen de la FPGA podrían llegar retardadas un tiempo que es igual a la suma de los tiempos de propagación de los pines de salida y el retardo de propagación a través de la placa. Para corregir el posible skew es necesario que las entradas de reloj lleguen con un cierto desfase, de forma que los flancos de subida alcancen la memoria al mismo tiempo que alcanzan todos los registros de la FPGA. La forma de conseguir el desfase del reloj es mediante el uso de un DLL (delay-locked loop), cuyo funcionamiento explicamos en el apartado dedicado al DCM.

En nuestro diseño utilizábamos un módulo controlador proporcionado por Xilinx para controlar estos bancos de memoria. A pesar de que el controlador permitía distintas formas de uso, como la lectura/escritura en ráfagas, nosotros únicamente realizábamos operaciones de lectura/escritura de un único byte cada vez. Además el controlador tenía una serie de restricciones de funcionamiento entre las que cabe destacar que para cada operación realizada sobre la memoria había que esperar cuatro ciclos para obtener el resultado debido a que las señales tenían que atravesar cuatro biestables antes de llegar a su destino.

Este módulo se encarga de controlar las señales de control de los bancos de memoria. Estas señales son:

- **Clock input (CLK)**: Todas las señales del interfaz son síncronas con el flanco de subida de este reloj.

- **Clock enable (~CEN)**: Esta señal es activa en baja por lo que cuando está en alta se ignoran las transiciones del reloj CLK.

- **Chip enable (~CE)**: Esta señal activa en baja capacita el chip de memoria para que pueda ser utilizado.

- **Output enable (~OE)**: Con esta señal se controla el bus de datos para que pueda ser utilizado para lectura y para escritura. Cuando la señal está en baja el bus permite leer datos de él y cuando está en alta escribirlos.

- **Write enable (~WE)**: Cuando esta señal está activa quiere decir que la operación que se va a realizar es una escritura.

- **Byte write enables (~WEA, ~WEB, ~WEC, ~WED)**: Cada una de estas señales controla un byte de escritura permitiendo escrituras de palabras parciales.

- **Advance/load (ADV/~LD)**: Esta señal controla el modo interno de ráfaga. Cuando esta señal está en baja la dirección usada en la memoria procede de los pines de entrada de dirección. Cuando está en alta, la dirección de memoria es la misma que en el ciclo anterior pero incrementada. Hay que tener en cuenta que

el contador de ráfaga es solo de dos bits por lo que la ráfaga máxima podría ser de cuatro ciclos. En la mayoría de los casos es más sencillo evitar el modo ráfaga.

2.2. Memoria DDR SDRAM (Virtex II Pro)

Los dispositivos DDR SDRAM (Double Data Rate SDRAM) son memorias síncronas que se caracterizan porque permiten leer y escribir datos dos veces por cada ciclo de reloj, de modo que trabajan al doble de la velocidad del bus sin necesidad de aumentar la frecuencia de reloj. Se presentan en módulos DIMM (Dual In-line Memory Module) de 184 contactos. Internamente están configuradas como un quad-bank de DRAMs, por lo que se trata de memorias dinámicas y por tanto es necesario realizar un refresco de su contenido cada cierto tiempo evitando así que se pierda.

El funcionamiento de este tipo de memorias es mucho más complejo que en el caso de una memoria SRAM, por ejemplo, ya que no basta con especificar la dirección a la que se quiere acceder y el tipo de operación que se desea realizar, sino que requiere la ejecución de una serie de comandos que permiten realizar además de operaciones de lectura o escritura sobre la memoria, operaciones de control.

Estos comandos son DESELECT, NO OPERATION, ACTIVE, READ, WRITE, BURST TERMINATE, PRECHARGE, AUTO-REFRESH y LOAD MODE REGISTER y su significado será explicado más adelante.

Otra característica importante de estas memorias es que funcionan con dos relojes diferenciales, CLK y CLK_Z, a partir de los cuales se generará una única señal de reloj cuyo flanco positivo coincidirá con el de subida del reloj CLK y con el de bajada del CLK_Z. Los comandos (direcciones y señales de control) son registrados en cada flanco positivo de reloj.

Las lecturas y escrituras están orientadas a ráfagas, de forma que los accesos comienzan seleccionando una determinada posición y continúan durante un número programado de posiciones en una secuencia programada. El tamaño de la ráfaga es programable y puede tomar valores 2, 4, 8 o por páginas completas, tanto para lectura como para escritura. Los tamaños de ráfaga permitidos dependen de la especificación de la DDR SDRAM usada en el módulo DIMM. Esta información se obtiene de la EEPROM de detección de presencia serie (SDP). En ella además es posible encontrar información del tipo de módulo así como parámetros de tiempo, los cuales son programados por el fabricante.

Al igual que ocurría en el caso de las memorias ZBT, las señales de reloj pueden llegar a la memoria con cierto retardo, el cual hay que eliminar. Para ello será necesario hacer uso de un DLL.

A continuación mostramos el esquema de un módulo de memoria DDR SDRAM:

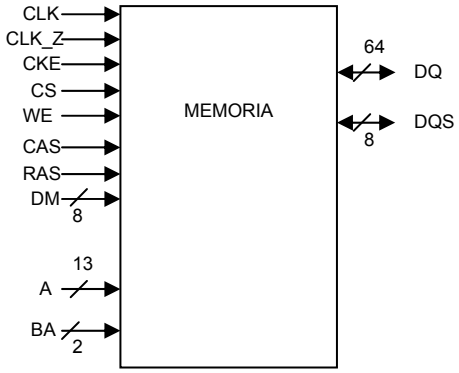


Figura 18: Esquema general de un módulo DDR SDRAM

El significado de cada una de las entradas y salidas es el siguiente:

Tabla 1: Entradas de un módulo DDR SDRAM

Entradas	Descripción
CLK/CLK_Z	- Entradas de reloj diferenciales - Las señales de control y las direcciones son registradas en los flancos positivos de CLK y en los negativos de CLK_Z
CKE	- Capacitación del reloj - Cuando está en alta activa el reloj interno, los buffers de entrada y los drivers de salida. En baja los desactiva
CS	- Capacita el módulo encargado de realizar la decodificación de los comandos - Funciona a baja
WE/CAS/RAS	- Definen los comandos - Funcionan a baja
DM [7:0]	- Máscara para escrituras
A [12:0]	- Entradas de dirección - Especifican: - La dirección de una fila para el comando ACTIVE - La dirección de una columna y el bit de activación o desactivación de la auto-precarga para los comandos de READ y WRITE - El código de operación para el comando de LOAD MODE REGISTER
BA[1:0]	- Dirección de banco - Define sobre que banco se aplica cualquiera de estos comandos ACTIVE, READ, WRITE y PRECHARGE

Tabla 1: Entrada/Salida de un módulo DDR SDRAM

Entrada/Salida	Descripción
DQ	- Bus usado para la transmisión de datos
DQS	- Strobe de los datos - Usado para detectar la presencia de nuevos datos en el bus - Es salida en las operaciones de lectura y entrada en las de escritura - Está alineada por flanco con los datos para las lecturas y alineada en el centro para las escrituras

2.2.1. Conceptos

Una vez dada una visión general de las características principales de la memoria y de sus conexiones vamos a explicar algunos conceptos necesarios para comprender mejor su funcionamiento.

Mode register

Es un registro que contiene información sobre la configuración de uso de la memoria. Esta información se agrupa en una serie de campos que son el tamaño de las ráfagas (Burst Length), el tipo de ráfaga (Burst Type), la latencia de lectura (CAS Latency) y el modo de operación (Operating Mode).

El siguiente diagrama indica los posibles valores que pueden tomar cada uno de los campos anteriores:

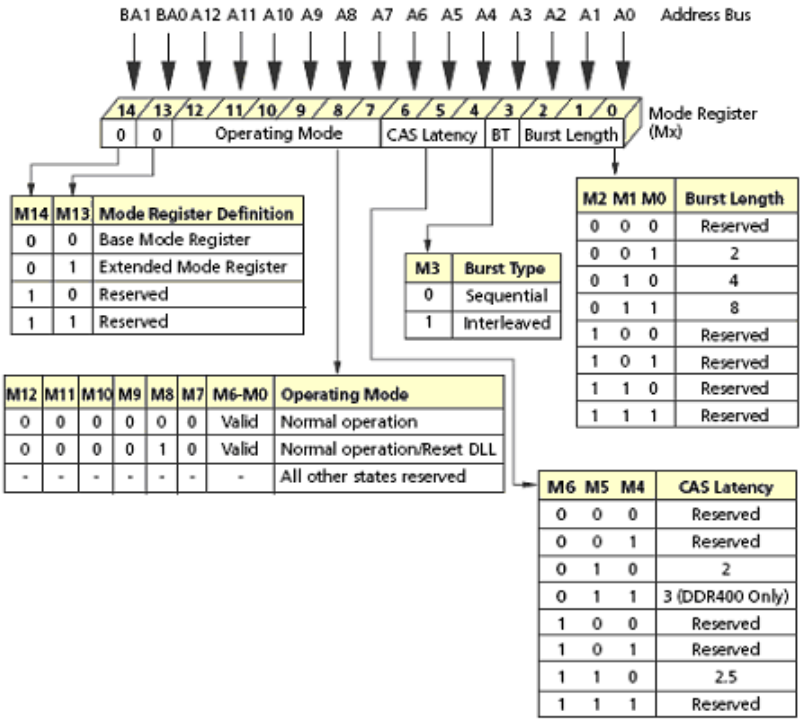


Figura 19: Contenido del Mode Register [R8]

El *mode register* se programa usando el comando LOAD MODE REGISTER (con BA1 = BA0 = 0) y su valor se mantiene hasta que vuelve a ser programado o el dispositivo pierde la alimentación.

El campo *burst type* indica el tipo de los accesos a los elementos de una ráfaga que puede ser secuencial o intercalado.

El campo *burst length* determina el máximo número de columnas a las que se puede acceder durante un comando de lectura o escritura. Debido a esto los *n* últimos bits de la dirección (siendo 2^n el tamaño de la ráfaga) se ignorarían y a partir de la dirección representada por el resto de bits se accedería a tantas columnas como indicara el valor del burst length.

La siguiente tabla muestra el orden de los elementos de una ráfaga en función del tipo de acceso.

Burst Length	Starting Column Address			Order of Accesses Within a Burst	
				Type= Sequential	Type= Interleaved
2	–	–	A0	–	–
	–	–	0	0-1	0-1
	–	–	1	1-0	1-0
4	–	A1	A0	–	–
	–	0	0	0-1-2-3	0-1-2-3
	–	0	1	1-2-3-0	1-0-3-2
	–	1	0	2-3-0-1	2-3-0-1
	–	1	1	3-0-1-2	3-2-1-0
8	A2	A1	A0	–	–
	0	0	0	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7
	0	0	1	1-2-3-4-5-6-7-0	1-0-3-2-5-4-7-6
	0	1	0	2-3-4-5-6-7-0-1	2-3-0-1-6-7-4-5
	0	1	1	3-4-5-6-7-0-1-2	3-2-1-0-7-6-5-4
	1	0	0	4-5-6-7-0-1-2-3	4-5-6-7-0-1-2-3
	1	0	1	5-6-7-0-1-2-3-4	5-4-7-6-1-0-3-2
	1	1	0	6-7-0-1-2-3-4-5	6-7-4-5-2-3-0-1
	1	1	1	7-0-1-2-3-4-5-6	7-6-5-4-3-2-1-0

Figura 20: Orden de los elementos de una ráfaga [R8]

La latencia de lectura, o *CAS Latency*, es el retardo en ciclos de reloj que hay entre el registro de un comando de lectura y está disponible el primer bit del dato.

Por último, el *operating mode* se selecciona mediante el lanzamiento del comando de LOAD MODE REGISTER y puede ser modo de operación normal o modo de operación reset DLL.

Extended Mode register

El *extended mode register* controla funciones adicionales a las del *mode register*. Entre ellas cabe destacar la de la capacitación o descapacitación del DLL.

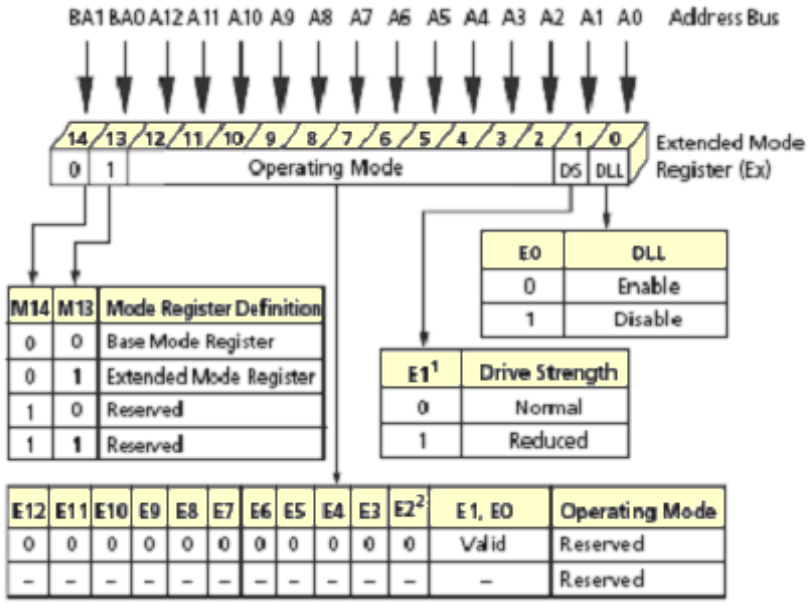


Figura 21: Contenido del Extended Mode Register [R8]

El *extended mode register* se programa usando el comando LOAD MODE REGISTER (con BA1 = 1 y BA0 = 0) y, al igual que el *mode register*, su valor se mantiene hasta que vuelve a ser programado o el dispositivo pierde la alimentación.

Comandos

Como ya se ha comentado, se requieren una serie de comandos para operar sobre la memoria. Estos comandos y su significado son:

- DESELECT: Evita que otros comandos sean ejecutados pero no afecta a los que ya estén en ejecución.

- NO OPERATION (NOP): Se usa para mandar a la memoria la realización de una no operación evitando con ello que durante los ciclos de espera se registren comandos que no se deben ejecutar. Al igual que el comando DESELECT, no afecta a las operaciones que ya estén en ejecución

  Los comandos DESELECT y NO OPERATION son intercambiables.

- LOAD MODE REGISTER: Permite cargar el *mode register* o el *extended mode register* en función de los valores de los bits BA1 y BA0.

- ACTIVE: Activa una fila de un banco particular en el que se vaya a realizar una secuencia de accesos. El banco seleccionado se determina mediante los

bits BA1 y BA0 del *mode register* y para la fila se usan el resto de los bits. La fila seleccionada se mantiene activa hasta que el comando PRECHARGE es lanzado al mismo banco.

- READ: Inicia una operación de lectura sobre una fila activa de un banco particular que está determinado por los bits BA1 y BA0 del *mode register*, de la misma forma que para el comando ACTIVE. La selección de la columna donde debe comenzar el acceso viene dada por los bits de dirección. El valor del bit A10 determina si se usa o no auto-precarga. Si la auto-precarga está seleccionada, la fila sobre la que se esté realizando el acceso es precargada automáticamente al final de cada lectura. Por el contrario, si no lo está, la fila permanece activa para otra secuencia de accesos.

- WRITE: Inicia una operación de escritura sobre una fila activa de un banco. Tanto la selección del banco, la de la columna de inicio del acceso y la de auto-precarga se realiza de la misma forma que en el caso del comando READ. La escritura de los datos que aparecen en el bus de datos DQ sobre la memoria está condicionada por el valor de la máscara DM, de tal forma que solo se escribirán aquellos bytes cuya posición coincida con bits con valor 0 del DM, y en caso contrario serán ignorados.

- PRECHARGE: Desactiva una fila abierta o activada de un banco particular o de todos los bancos. Un banco puede permanecer disponible para una secuencia de accesos durante un tiempo determinado después de lanzar un comando de precarga, excepto en el caso de que esté activada la auto-precarga, para la que los comandos de lectura o escritura de bancos diferentes está permitida siempre y cuando no interrumpan la transferencia de datos del banco actual y no violen ninguna otra restricción temporal. La entrada A10 determina si uno o todos los bancos tienen que ser precargados, y en el caso de que sólo un banco sea precargado, las entradas BA0 y BA1 determinan el banco concreto. En el otro caso estas entradas no son tenidas en cuenta. Una vez el banco haya sido precargado, este se encontrará en un estado idle y deberá ser activado antes de la realización de cualquier comando de lectura o escritura sobre él. El comando PRECHARGE puede ser tratado como una NOP si no hay ninguna fila de cualquier banco activa o si una operación de apertura previa está todavía en el proceso de precarga.

- AUTO-PRECHARGE: Se trata de una característica que permite realizar precargas sobre un mismo banco pero sin requerir ningún comando explicito. Esto se consigue como ya se ha comentado activando el bit A10 junto con un comando READ o WRITE. La precarga del banco o fila que ha sido seleccionada para la operación se realiza automáticamente tras la finalización de ésta. La auto precarga no es persistente por lo que tiene que ser capacitada o discapacitada para cada lectura o escritura individual.

- BURST TERMINATE: Finaliza la lectura de una ráfaga, con auto-precarga desactivada.

- AUTO REFRESH: Este comando se utiliza durante el funcionamiento normal de la memoria. No es persistente por lo que debe ser lanzado cada vez que se

quiere realizar un autorefresco y requiere que todos los bancos de la memoria estén en un estado de espera antes de ser lanzado.

- SELF REFRESH: Este comando puede ser usado para mantener datos en la memoria incluso si se resetea o se apaga. Es inicializado como el comando AUTO REFRESH pero con la diferencia de que la señal CKE está descapacitada.

Para generar estos comandos es necesario usar los valores de las señales de control de la forma en la que aparecen en la siguiente tabla:

Name (Function)	CS#	RAS#	CAS#	WE#	Address
DESELECT (NOP)	H	X	X	X	X
NO OPERATION (NOP)	L	H	H	H	X
ACTIVE (Select bank and activate row)	L	L	H	H	Bank/Row
READ (Select bank and column, and start READ burst)	L	H	L	H	Bank/Col
WRITE (Select bank and column, and start WRITE burst)	L	H	L	L	Bank/Col
BURST TERMINATE	L	H	H	L	X
PRECHARGE (Deactivate row in bank or banks)	L	L	H	L	Code
AUTO REFRESH or SELF REFRESH (Enter self refresh mode)	L	L	L	H	X
LOAD MODE REGISTER	L	L	L	L	Op-Code

Figura 22: Lista de comandos [R8]

2.2.2. Operaciones

Por último veremos con más detalles los pasos necesarios para realizar tanto una lectura como una escritura sobre la memoria.

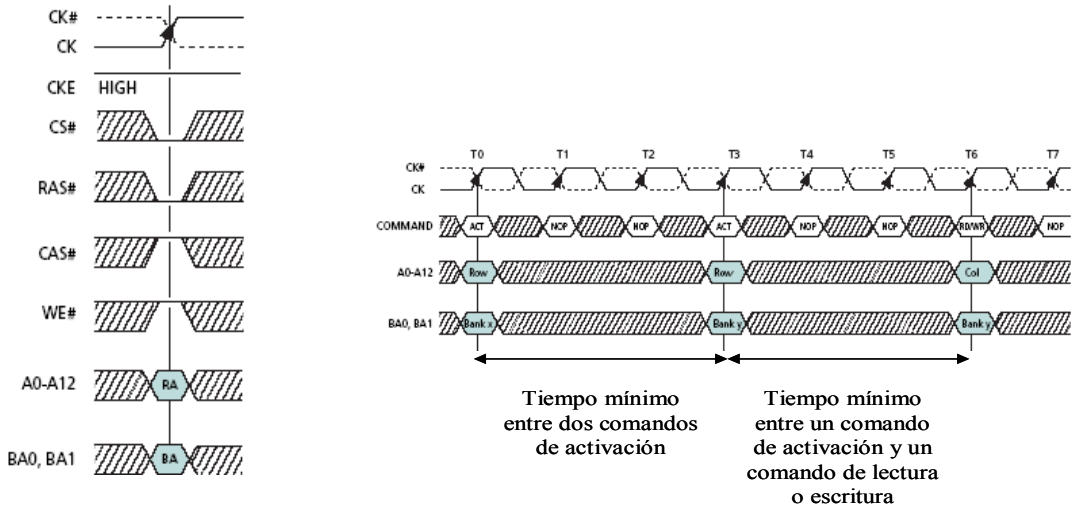
El primer paso para poder utilizar correctamente la memoria es realizar su inicialización de un modo predeterminado. En primer lugar se debe aplicar una serie de voltajes cumpliendo unas determinadas restricciones. Cuando la fuente de alimentación, los voltajes de referencia y el reloj son estables, se requieren 200 µs para poder aplicar cualquier comando ejecutable, tanto de lectura como de escritura. Una vez que estos 200 µs han pasado, la secuencia de inicialización empieza. Ésta se realiza en los siguientes pasos:

- Un comando de DESELECT o NOP es aplicado y el CKE se pone a alta
- Un comando de PRECHARGE es aplicado
- Un comando de LOAD MODE REGISTER es lanzado para cargar el *extended mode registrer* (con BA1 = 0 y BA0 = 1) y capacitar el DLL
- Otro comando de LOAD MODE REGISTER es lanzado para cargar el *mode registrer* (con BA1 = BA0 = 0), resetear el DLL y programar los parámetros de operación
- Un comando de PRECHARGE es aplicado, colocando todos los bancos del dispositivo en un estado de espera o estado IDLE
- Una vez en el estado IDLE, dos ciclos de AUTO REFRESH son realizados.

Cuando se ha completado la inicialización la DDR SDRAM está preparada para realizar cualquier operación.

Previamente a la ejecución de un comando de lectura o escritura es necesario ejecutar un comando de activacion (comando ACTIVE) mediante el cual se selecciona un banco de la memoria y una fila del mismo para poder utilizarla.

En la figura siguiente vemos como se realiza la activación:



RA = Dirección de fila

BA = Dirección de banco

Figura 23: Activación de una fila de un banco de memoria [R8]

Como puede observarse es posible lanzar comandos de activación consecutivos pero cumpliendo un cierto tiempo mínimo entre ellos. Si se intenta lanzar dos comandos de activación a filas diferentes pero de un mimo banco, no sólo habrá que esperar este tiempo, sino que también habrá que esperar a que la fila sea cerrada o desactivada mediante el correspondiente comando de PRECHARGE. Si la activación es de filas de bancos diferentes bastaría con cumplir la restricción del tiempo para poder lanzar dos comandos. Entre el lanzamiento de un comando de activación y el de uno de lectura o escritura también se debe esperar un cierto tiempo.

Lectura

Para realizar una operación de lectura sobre la memoria es necesario lanzar un comando de lectura (comando READ).

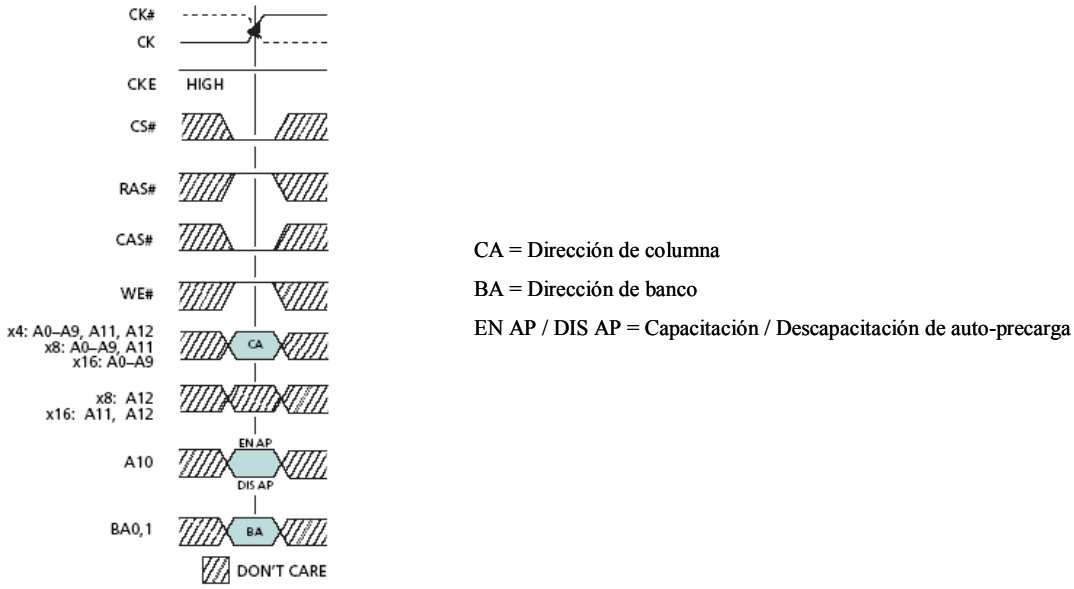


Figura 24: Comando de lectura [R8]

Como se observa en el diagrama anterior tanto el banco como la columna son proporcionados con el comando. También se indica si se desea capacitar o no la auto precarga. Si ésta está capacitada, cada vez que se completa la lectura de una ráfaga la fila a la que se esta accediendo es precargada.

Durante la lectura de cada ráfaga, los datos de salida (DQ) serán válidos una vez que haya transcurrido el CAS latency desde que el comando de lectura se lanzó. A partir de ese momento, en cada flanco positivo o negativo del reloj llegará un nuevo dato hasta que se hayan leído tantos datos como indique el tamaño de las ráfagas. El strobe de los datos (DQS) será transmitido por la memoria a la vez que los datos. Una vez que se ha completado una ráfaga, y asumiendo que ningún otro comando ha sido inicializado, el DQ se pone en alta impedancia. Esto puede verse en la figura que aparece a continuación, donde se muestra la ejecución de una operación de lectura para tres valores diferentes del CAS latency y con tamaño de ráfaga de valor 4 en todos los casos:

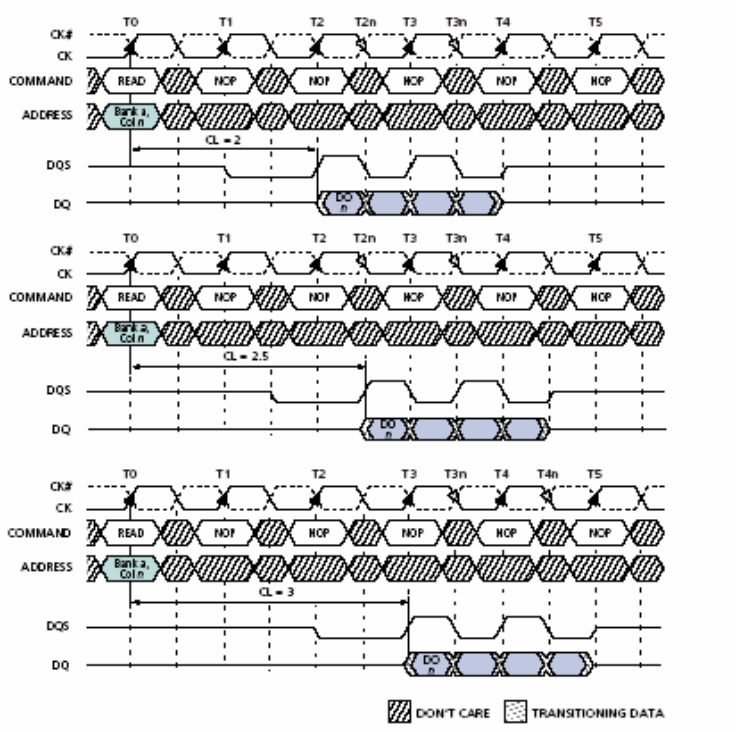


Figura 25: Ejecución de un comando de lectura [R8]

Es posible que los datos de una lectura puedan ser concatenados o truncados con los datos de otras lecturas consecutivos. En cualquiera de los casos puede ser mantenido un flujo continuo de datos de forma que el primer dato de una nueva ráfaga siga al último dato de una ráfaga que se haya completado o de una ráfaga truncada.

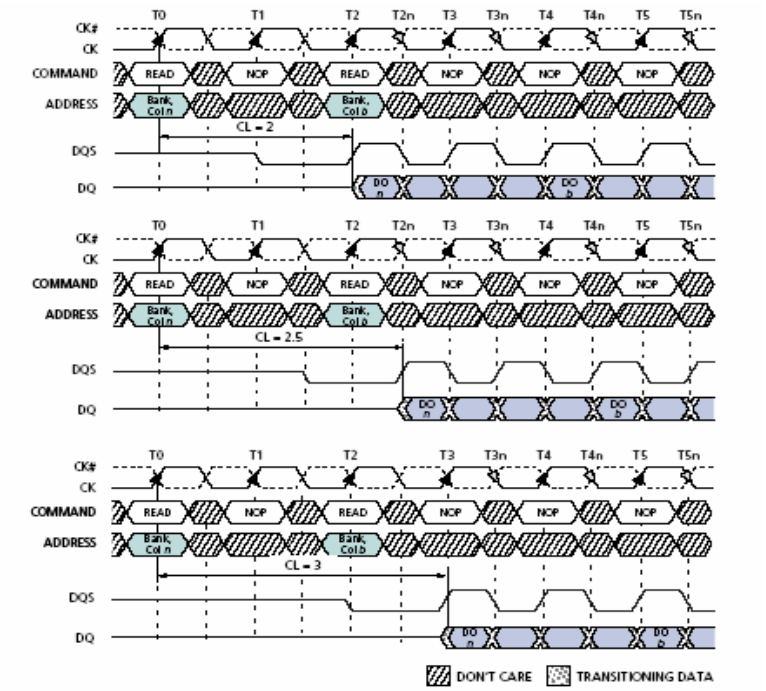
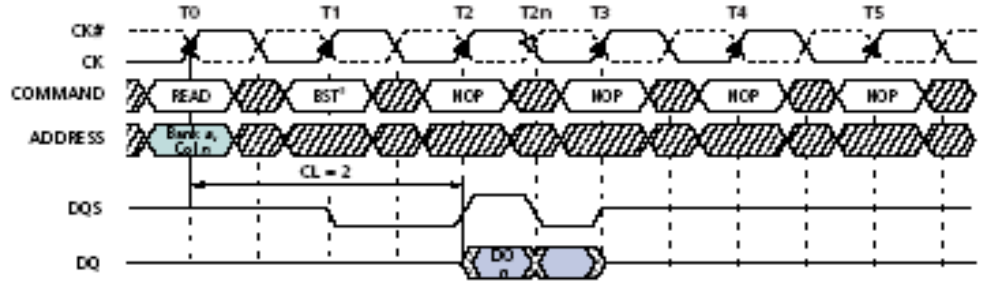


Figura 26: Ejecución de comandos de lectura consecutivos [R8]

También puede ocurrir que tras lanzar un comando de lectura se lance el comando BURST TERMINATE o el comando PRECHARGE. A continuación se muestra un ejemplo de la ejecución de estos comandos tras una operación de lectura, suponiendo tamaño de ráfaga de valor 4 y CAS latency 2.

Comando READ – Comando BURST TERMINATE



Comando READ – Comando PRECHARGE

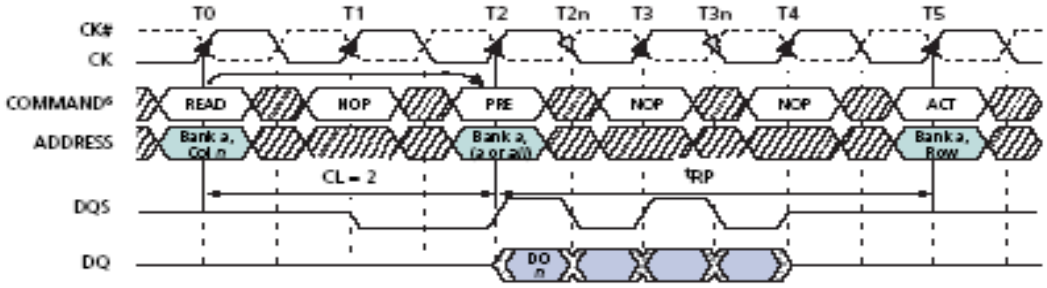


Figura 27: Ejecución de un comando de lectura seguido de un comando BURST TERMINATE y de un comando PRECHARGE [R8]

En cualquiera de estos casos, el lanzamiento de dos comandos consecutivos requiere esperar tantos ciclos como pares de datos se quieran leer.

Por último, existe también la posibilidad de lanzar un comando de escritura seguido de un comando de lectura siempre y cuando los datos de la lectura hayan sido completados o truncados previamente.

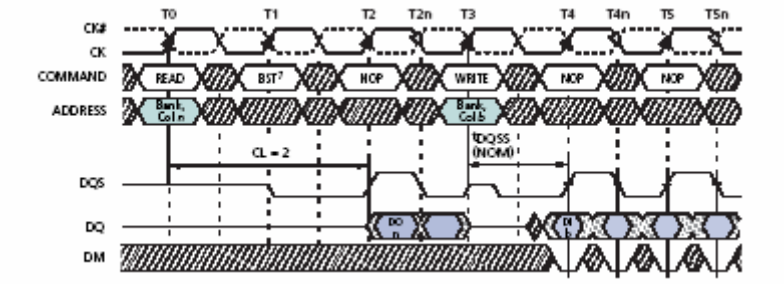
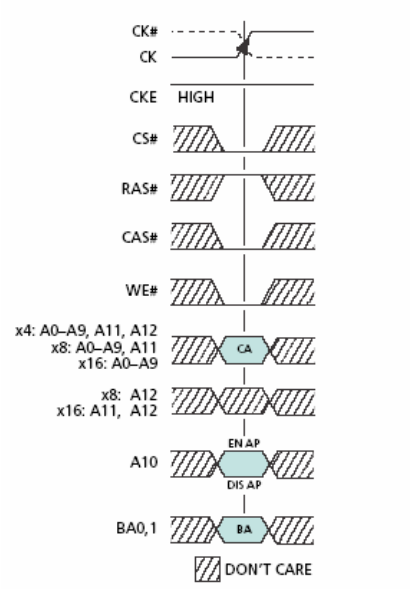


Figura 28: Ejecución de un comando de lectura seguido de un comando de escritura [R8]

Escritura

Para realizar una operación de escritura sobre la memoria es necesario lanzar un comando de escritura (comando WRITE).

Al igual que en el caso de las operaciones de lectura, tanto el banco como la columna son proporcionados con el comando. También se indica si se desea capacitar o no la auto precarga. Si ésta está capacitada, cada vez que se completa la lectura de una ráfaga la fila a la que se esta accediendo es precargada.



CA = Dirección de columna
BA = Dirección de banco
EN AP / DIS AP = Capacitación / Descapacitación de auto-precarga

Figura 29: Comando de escritura [R8]

Durante la escritura de cada ráfaga, el primer dato válido de entrada se registra en el primer flanco positivo del DQS y a partir de ese momento se registran el resto de los datos en los flancos sucesivos.

Una vez que se ha completado la escritura de una ráfaga, y asumiendo que ningún otro comando ha sido inicializado, el DQ se pone en alta impedancia y cualquier entrada de datos adicional es ignorada. Esto puede verse en la figura que aparece a continuación, donde se muestra la ejecución de una operación de escritura con tamaño de ráfaga de valor 4 y tiempo t_{DQS} entre el lanzamiento del comando WRITE y el primer flanco positivo del DQS:

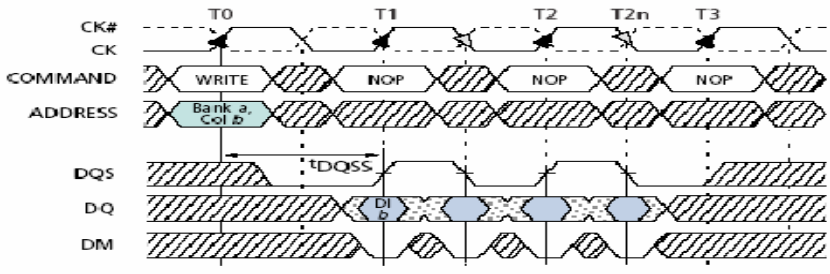


Figura 30: Ejecución de un comando de escritura [R8]

Los datos de una operación de escritura pueden ser concatenados o truncados con los de otra. En cualquiera de los casos, se establece un flujo continuo de datos de entrada, de forma que el primer dato de entrada de una nueva escritura siga al último de una ráfaga que se haya completado o de una ráfaga truncada. En la siguiente figura se puede ver el resultado de la ejecución de dos comandos de escritura consecutivos:

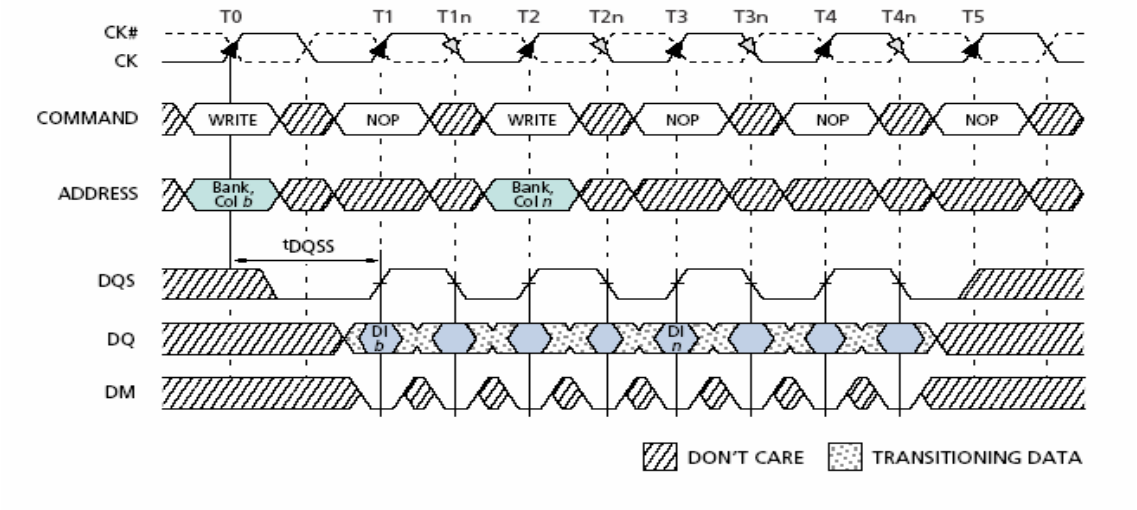
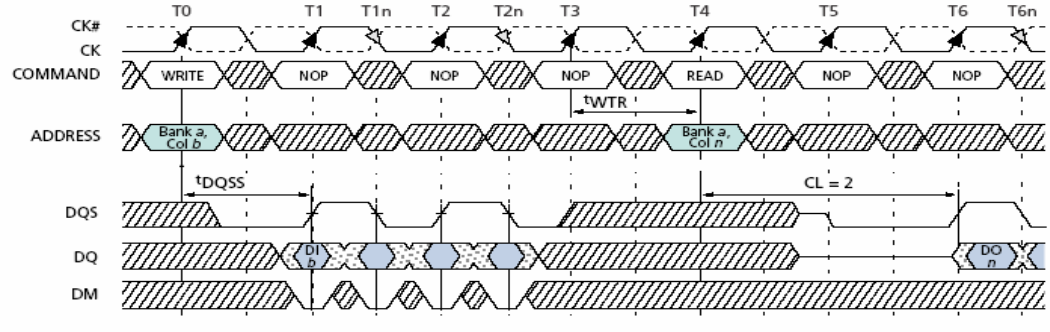


Figura 31: Ejecución de comandos de escritura consecutivos [R8]

Es posible lanzar un comando de lectura después de un comando de escritura. Existen dos posibilidades, o bien la lectura se espera hasta que la escritura haya finalizado, o bien se interrumpe la escritura que se este realizando.

Comando WRITE – Comando READ sin interrupción de escritura



Comando WRITE – Comando READ con interrupción de escritura

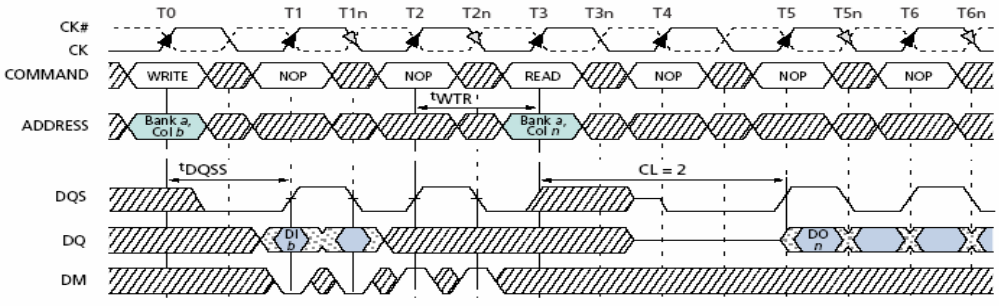
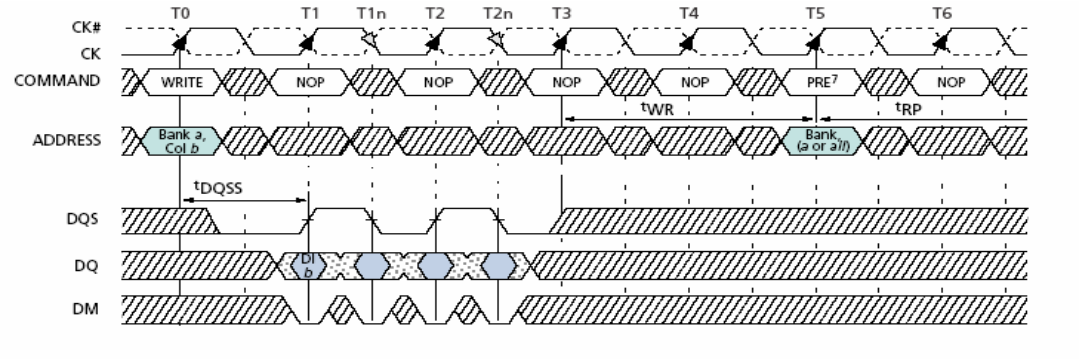


Figura 32: Ejecución de un comando de escritura seguido de un comando de lectura [R8]

Por último, después de un comando de escritura puede también lanzarse el comando PRECHARGE. En este caso ocurre lo mismo que con el comando READ, o bien la precarga se realiza una vez que la escritura haya finalizado, o bien se interrumpe la escritura que se este realizando. A continuación se muestran ejemplos con cada una de estas posibilidades:

Comando WRITE – Comando PRECHARGE sin interrupción de escritura



Comando WRITE – Comando PRECHARGE con interrupción de escritura

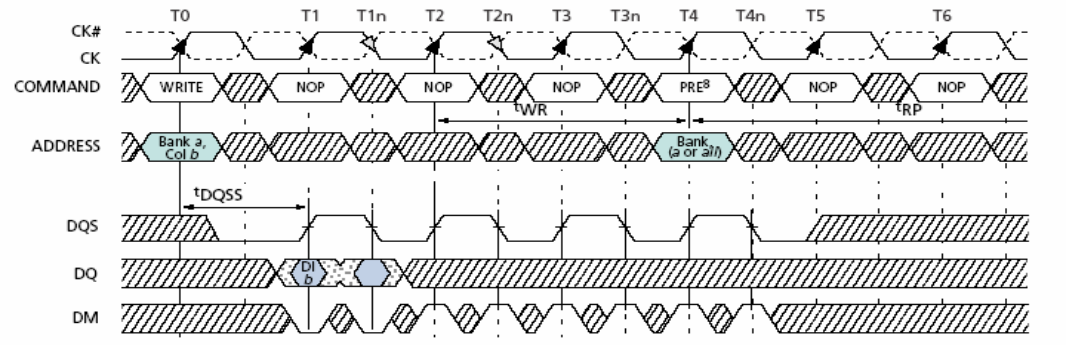


Figura 33: Ejecución de un comando de escritura seguido de un comando de precarga [R8]

2.2.3. Controlador DDR SDRAM

Para poder utilizar una memoria DDR SDRAM es necesario diseñar un controlador que gestione su funcionamiento. El desarrollo de este controlador lo hemos realizado mediante el uso de la aplicación MIG proporcionada por Xilinx y que se explica en el apartado de herramientas software. Esta aplicación genera una interfaz completa que encapsula el control de la memoria en unas cuantas señales de usuario. Sin embargo, este controlador a pesar de simplificar el uso de la memoria, todavía requiere una gestión de las señales de control bastante compleja, ya que para realizar las distintas operaciones permitidas, es necesario seguir unas especificaciones de tiempo que en caso de tener que utilizar el módulo desde distintas partes, provocaría un aumento de la complejidad de los módulos que lo necesitan únicamente para realizar operaciones básicas. Para facilitar el uso del controlador hemos desarrollado una interfaz superior que lo encapsula y que permite trabajar con la memoria DDR como si se tratase de una simple SDRAM.

2.2.3.1. Componentes

El diseño completo de nuestra interfaz junto con el controlador generado por el MIG da lugar a una estructura jerárquica de módulos como la que se muestra a continuación:

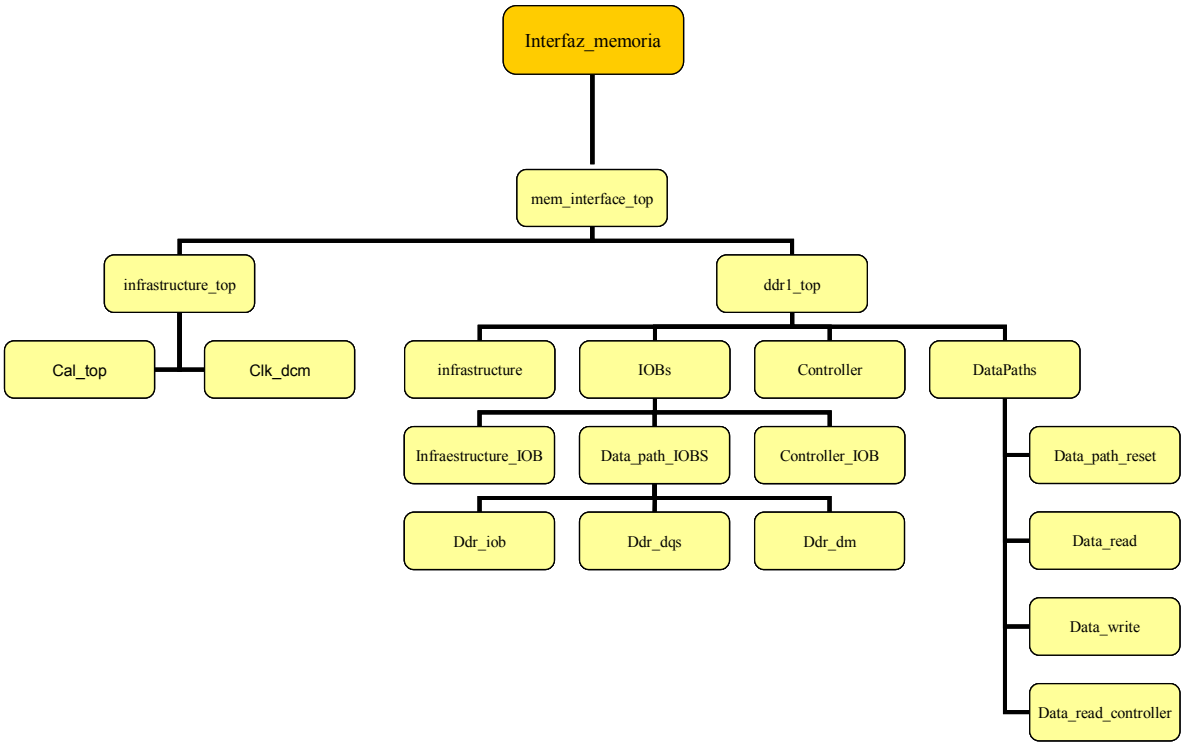


Figura 34: Jerarquía de módulos del controlador de memoria

Vamos a explicar el funcionamiento de este controlador, centrándonos en la parte correspondiente a los módulos generados por el MIG ya que el módulo *interfaz_memoria* generado por nosotros se encuentra explicado en el capítulo de la arquitectura del sistema.

El módulo *infrastructure_top* es el encargado de generar las señales de reloj y de reset utilizadas por el controlador principal de la memoria mediante el uso de un DCM. Además genera una señal que se activa una vez transcurrido 200 μs desde el arranque del sistema.

El módulo *ddr1_top* es el que realmente gestiona el funcionamiento de la memoria ya que el módulo *mem_interface_top* únicamente se encarga de interconectarlo al *infrastructure_top* comentado anteriormente. Está formado por otros cuatro módulos, cuya interconexión se muestra en la siguiente figura:

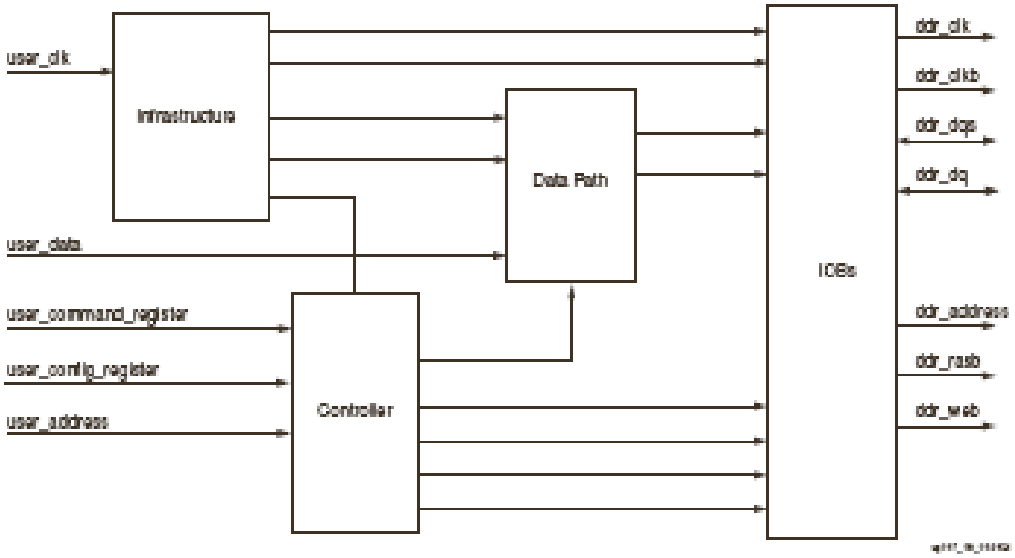


Figura 35: Interconexión de los módulos del controlador [R9]

A continuación explicamos con más detalle cada uno de estos módulos:

Controller

El módulo controlador recibe y decodifica los comandos dados por el usuario y genera los comandos de lectura, escritura e inicialización de la memoria además de señales para otros módulos.

El proceso de inicialización es manejado por el controlador una vez recibida la orden de inicialización por parte del usuario, que en este caso sería nuestro interfaz de memoria.

El controlador basa su funcionamiento en una máquina de estados encargada de realizar los pasos necesarios para cada una de las funcionalidades. Su diagrama de transición de estados es el siguiente:

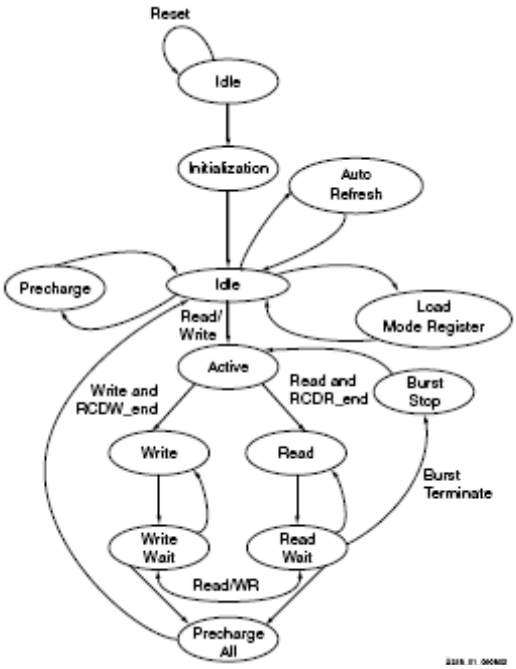


Figura 36: Diagrama de estados del controlador [R10]

Data Path

Este módulo es el encargado de transmitir los datos a la memoria y de recibirlos. Entre sus principales funciones se incluyen almacenar los datos de lecturas, transferir los datos de escritura y capacitar la escritura de los módulos IOB.

Está compuesto de otros cuatro módulos:

- *Data Read Controller*: Se encarga de generar todas las señales de control usadas por el *Data Read*

- *Data Read*: Contiene la ruta de datos para las lecturas

- *Data Write*: Contiene la ruta de datos para las escrituras. Los datos de escritura y las señales de capacitación son propagadas a la DDR SDRAM a través de los flip-flops IOB. La parte de la ruta de datos de las escrituras donde se usan los IOBs se implementan en el módulo *datapath_IOBs*.

- *Data Path Reset*

IOBs

En este módulo se implementan todas las señales de entrada y salida de la FPGA.

2.2.3.2. Forma de uso

El manejo del controlador se realiza sobre la estructura superior de su jerarquía, el módulo *mem_interface_top*, el cual contiene una serie de salidas que se conectan directamente a la memoria y unas entradas que son las que el usuario utilizara para realizar las operaciones sobre la memoria. La dificultad en el uso de este controlador viene dada porque las señales de entrada deben estar sincronizadas cada una con un reloj determinado. Estos relojes son cuatro de la misma frecuencia y fase desplazada 90º entre si.

Vemos ahora un diagrama en el que mostramos las entradas y salidas del controlador:

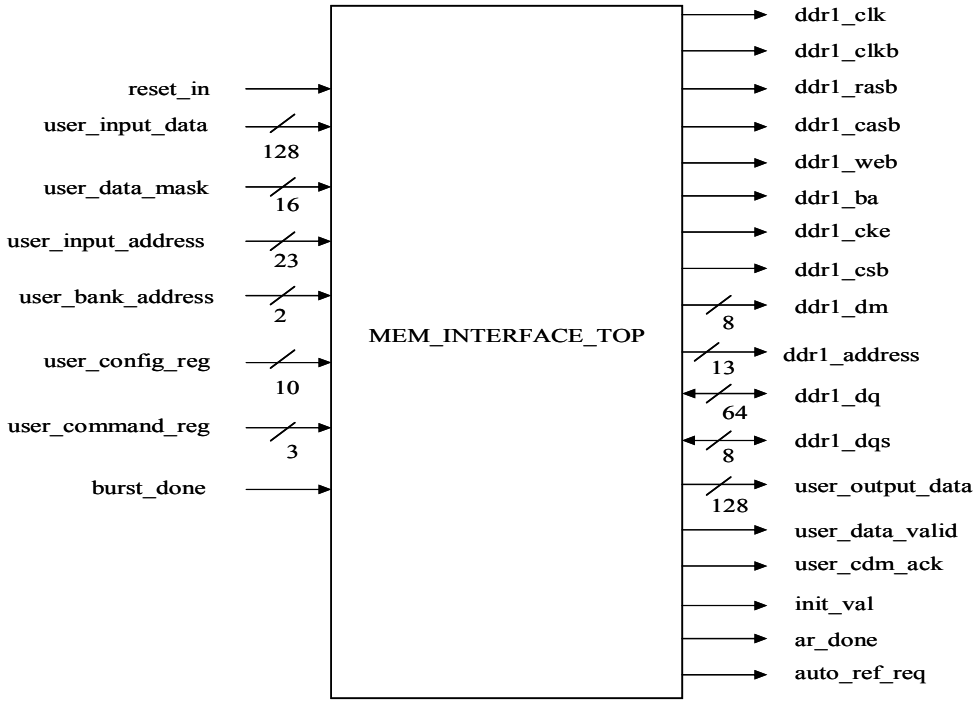


Figura 37: Esquema general del controlador de memoria

Mostramos un listado con las señales en el que describimos su utilización diferenciando si son señales propias del controlador o bien si son las que se comunican con la memoria:

Entrada/Salida de la memoria

Tabla 3: Entrada/Salida del controlador de memoria

Entrada/Salida	Descripción
ddr1_dq	- Bus de datos
ddr1_dqs	- Strobe de los datos

Salidas de la memoria

Tabla 4: Salidas del controlador a la memoria

Salidas	Descripción
ddr1_address	- Direccion
ddr1_rasb	- Comando
ddr1_casb	- Comando
ddr1_web	- Comando
ddr1_ba	- Dirección de banco
ddr1_clk	- Reloj
ddr1_clkb	- Reloj invertido
ddr1_csb	- Selector de chip
ddr1_cke	- Capacitación del reloj
ddr1_dm	- Máscara de datos

Entradas del controlador

Tabla 5: Entradas del controlador de memoria

Entradas	Descripción
reset_in	- Señal de reset del sistema
user_input_data[(2n-1):0]	- Bus de datos de escritura de la memoria - Sólo válido para comandos de escritura - Tamaño del bus = 2n, siendo n el ancho del bus de datos de la memoria - El controlador es el encargado de convertir los datos simples en datos dobles - De los 2n bits, los más significativos se escribirán en el flanco de subida del reloj y los menos significativos en el de bajada
user_data_mask[(2m-1):0]	- Bus de máscara de datos para las operaciones de escritura - Al igual que el *user_input_data* su tamaño es el doble que el del bus de máscara de la memoria - De los 2m bits los más significativos se aplican en el flanco de subida del reloj y los menos significativos en el de bajada
user_input_address[addrwidth-1:0]	- Combinación de la dirección de fila y columna para las lecturas y escrituras de la memoria - Los bits menos significativos corresponden a la columna y los más significativos a la fila
user_bank_address[bankaddrwidth-1:0]	- Dirección de banco de la memoria
user_config_reg[9:0]	- Datos de configuración para la inicialización de la memoria - Se usa para la inicialización del LOAD MODE REGISTER - Tiene el siguiente formato:

<table>
<tr><td></td><td>
<table>
<tr><td>9</td><td>8</td><td>7</td><td>6 5 4</td><td>3</td><td>2 1 0</td></tr>
<tr><td>EMR</td><td>BMR</td><td>BMR/EMR</td><td>CAS Latency</td><td>Burst Type</td><td>Burst Length</td></tr>
</table>
- Los bits 9, 8 y 7 están reservados
- El controlador sólo soporta tipo de ráfaga secuencial
</td></tr>
<tr><td>user_comand_reg[2:0]</td><td>- Comandos de memoria
<table>
<tr><th>user_command_reg[2:0]</th><th>Descripción de comando</th></tr>
<tr><td>000</td><td>NOP</td></tr>
<tr><td>010</td><td>Inicialización de la memoria</td></tr>
<tr><td>100</td><td>WRITE</td></tr>
<tr><td>101</td><td>LOAD MODE REGISTER</td></tr>
<tr><td>110</td><td>READ</td></tr>
<tr><td>Otros</td><td>RESERVED</td></tr>
</table>
</td></tr>
<tr><td>burst_done</td><td>- Señal de fin de transferencia de datos
- Se activa durante dos ciclos de reloj al final de una transferencia de datos</td></tr>
</table>

Salidas del controlador

Tabla 6: Salidas del controlador de memoria

Salidas	Descripción
user_output_data[(2n-1):0]	- Dato leído de la memoria - El controlador convierte el dato DDR leído de la memoria a dato SDR - Tamaño del bus = 2n, siendo n el ancho de bus de datos de la memoria
user_data_valid	- Señal que indica que el *user_output_data* es válido
user_cmd_ack	- Señal de reconocimiento de un comando - La activa el controlador durante la lecturas o escrituras a la memoria - Mientras este activada no se debe lanzar ningún otro comando
init_val	- Señal que activa el controlador para indicar que la inicialización de la memoria ha sido completada
ar_done	- Señal que indica que un comando de auto-refresco ha sido dado a la memoria - La activa el controlador un ciclo después de que el comando haya sido dado y el tiempo T_{rfc} se haya completado. Este tiempo está determinado por el valor de un contador (RFC_COUNTER) que se encuentra en un fichero de parámetros
auto_ref_req	- Señal de petición de auto-refresco - Esta señal se activa cuando el controlador determina que ha pasado un tiempo y es necesario realizar un auto-refresco sobre la memoria

	- Los tiempos en los que el controlador determina si hay que realizar auto-refresco viene dado por el valor de un contador cuyo rango viene dado por el valor del parámetro REF_FREQ_CNT del fichero de parámetros

Para finalizar vamos a hablar de la forma en la que hay que utilizar el controlador para realizar las distintas operaciones sobre la memoria:

Inicialización de la memoria

Como ya se comentó cuando se explicó el funcionamiento de la memoria DDR SDRAM, antes de poder realizar cualquier operación sobre la memoria, es necesario hacer una correcta inicialización de ella de un modo predeterminado. Para ello se utiliza un comando de inicialización, el cual sólo puede ser lanzado cuando todas las señales de reset están desactivadas, lo cual ocurre 200 μs después del arranque del sistema ya que la memoria necesita este tiempo para estabilizarse.

Para lanzar el comando de inicialización es necesario realizar una serie de pasos para configurar el *Load Mode Register*. Una vez que estos pasos se hayan completado el controlador activará la señal de *init_val* indicando así que la memoria ya está lista para ser usada.

La secuencia de pasos que hay que realizar es la siguiente:

(1) Se pone la configuración de datos válida para la inicialización en el *user_config_reg[9:0]* dos ciclos antes de poner el comando de inicialización en el *user_comand_reg[2:0]*
(2) Después de estos dos ciclos, se pone el comando de inicialización en el *user_comand_reg[2:0]* en el flanco de subida del clk180
(3) El controlador indica cuando la inicialización se ha completado activando la señal *init_val* en un flanco de subida del clk180
(4) Una vez que la señal *init_val* está activada, se puede lanzar cualquier comando ejecutable a la memoria

En el siguiente diagrama se puede el cronograma de tiempo resultante de realizar esta secuencia de pasos:

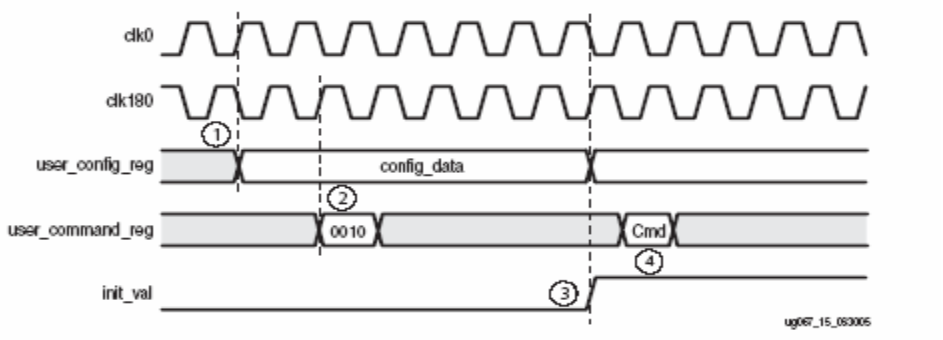


Figura 38: Inicialización de la memoria [R9]

Escritura

Para iniciar una operación de escritura es necesario enviar un comando de escritura al controlador de la memoria. Este comando es activado en el flanco positivo del clk180 colocando en el *user_command_reg[9:0]* la configuración necesaria. Para indicar que un comando de escritura ha sido lanzado el controlador activa la señal *user_cmd_ack* en el flanco positivo del clk180. La primera dirección (fila+columna) se debe activar junto con el comando de escritura y debe mantenerse activa durante 4.5 ciclos después de la activación del *user_cmd_ack*. Las direcciones consecutivas son activadas cada dos ciclos positivos del clk0 a partir del final de la primera dirección. El primer dato de entrada se activa con el primer flanco positivo de clk90 después de activar el *user_cmd_ack*. Para terminar la escritura de una ráfaga, la señal *burst_done* es activada después de la última dirección en *user_input_address* y se mantiene activa durante dos ciclos de reloj. Después de esto el controlador termina la escritura de la ráfaga y emite una precarga a la memoria, tras la cual el *user_cmd_ack* se desactiva. Para lanzar cualquier otro comando hay que esperar a que esta señal esté desactivada.

En la siguiente figura se muestra el diagrama de tiempo de la ejecución de una operación de escritura, para un tamaño de ráfaga de valor 4:

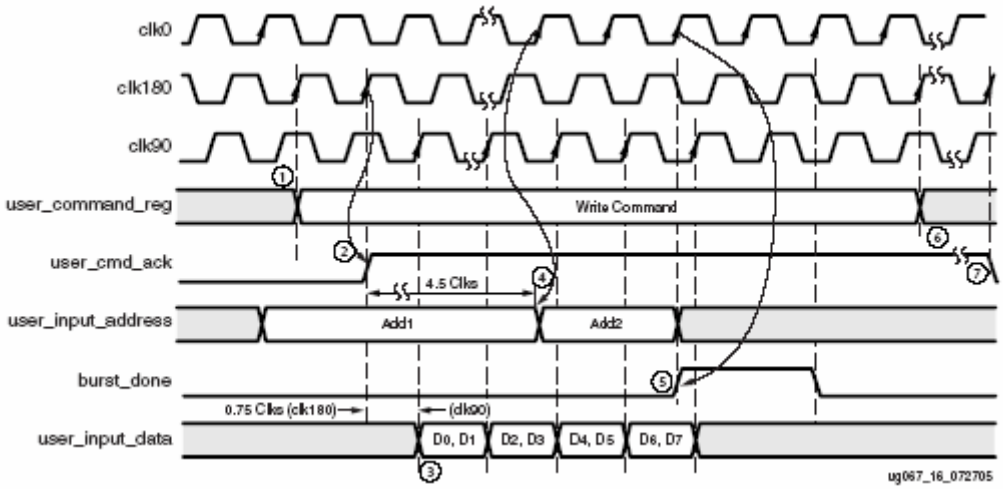


Figura 39: Ejecución de una operación de escritura [R9]

Sobre la figura aparecen marcados los instantes importantes de la ejecución, cuyo significado es el siguiente:

(1) Activación del comando de escritura
(2) Activación del *user_cmd_ack* en el flanco positivo del clk180 para indicar que un comando de escritura ha sido lanzado
(3) Activación de los datos de entrada en el primer flanco positivo de clk90 después de activar el *user_cmd_ack*
(4) Instante hasta el que se mantiene la dirección
(5) Activación del *burst_done* para terminar la escritura de una ráfaga
(6) Desactivación del comando de escritura
(7) Desactivación del user_cmd_ack

Lectura

Para iniciar una operación de lectura el proceso a seguir es muy parecido al de una operación de escritura. En primer lugar es necesario enviar un comando de lectura al controlador de la memoria que es reconocido en el flanco positivo del clk180. Para indicar que un comando de lectura ha sido lanzado el controlador activa la señal *user_cmd_ack* en el flanco positivo del clk180. La primera dirección (fila+columna) se debe activar junto con el comando de lectura y debe mantenerse activa durante 4.5 ciclos después de la activación del *user_cmd_ack.* Las direcciones consecutivas son activadas cada dos ciclos positivos del clk0 a partir del final de la primera dirección. Los datos de una lectura estarán disponibles en el *user_output_data* y serán válidos sólo cuando la señal *user_data_valid* esté activada, y se podrá leer a partir de este momento un dato DDR en cada flanco de subida de clk90. Para terminar la lectura de una ráfaga, la señal *burst_done* es activada después de la última dirección en *user_input_address* y se mantiene activa durante dos ciclos de reloj. Después de esto el controlador termina la lectura de la ráfaga y emite una precarga a la memoria, tras la cual el *user_cmd_ack* se desactiva. Para lanzar cualquier otro comando hay que esperar a que esta señal esté desactivada.

En la siguiente figura se muestra el diagrama de tiempo de la ejecución de una operación de lectura, para un tamaño de ráfaga de valor 4:

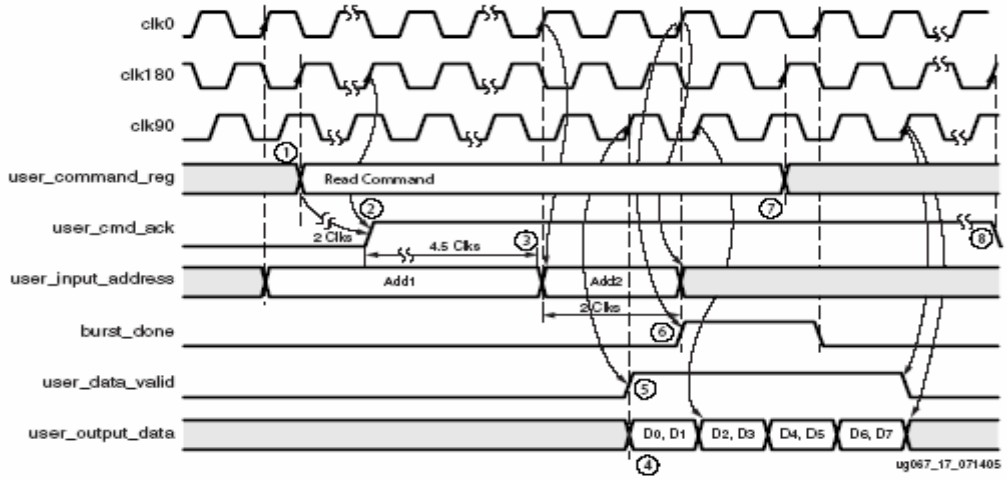


Figura 40: Ejecución de una operación de lectura [R9]

Sobre la figura aparecen marcados los instantes importantes de la ejecución, cuyo significado es el siguiente:

(1) Activación del comando de escritura
(2) Activación del *user_cmd_ack* en el flanco positivo del clk180 para indicar que un comando de escritura ha sido lanzado
(3) Instante hasta el que se mantiene la dirección
(4) Instante en el que los datos leídos de la memoria están disponibles en el *user_output_data*
(5) Instante en el que los datos leídos de la memoria disponibles en el *user_output_data* son válidos

(6) Activación del *burst_done* para terminar la escritura de una ráfaga
(7) Desactivación del comando de escritura
(8) Desactivación del user_cmd_ack

Auto-refresco

El controlador realiza el refresco de la memoria periódicamente. Cada 7.7 μs lanza una petición de auto refresco. Cuando esto ocurre se debe terminar la ejecución de cualquier comando que estuviese en proceso. Para iniciar cualquier otro comando se debe esperar a que el auto refresco haya sido completado, lo cual se indica con la activación de la señal ar_done.

2.3. DCM (Digital Clock Manager)

El uso de componentes hardware que requieren un control complejo, como pueden ser los módulos de memoria, hacen necesario utilizar una gran cantidad de señales que deben estar sincronizadas no siempre por el mismo reloj. Por ello el reloj pasa a ser una clave del diseño y en el caso de no estar correctamente sincronizado podría provocar un mal funcionamiento. Debido a esto es necesaria la utilización de algún tipo de módulo que se encargue de gestionar las señales de reloj. Este módulo es el DCM.

La librería de componentes de Xilinx nos proporciona un módulo de este tipo, cuyo esquema se muestra a continuación:

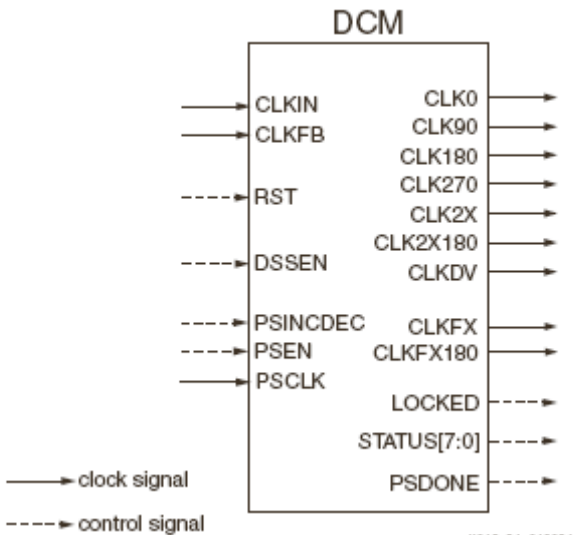


Figura 41: Esquema general de un módulo DCM [R12]

Un módulo DCM es capaz de a partir de un reloj de entrada generar una serie de relojes de salida con distinta frecuencia o desfasados respecto al de la entrada.

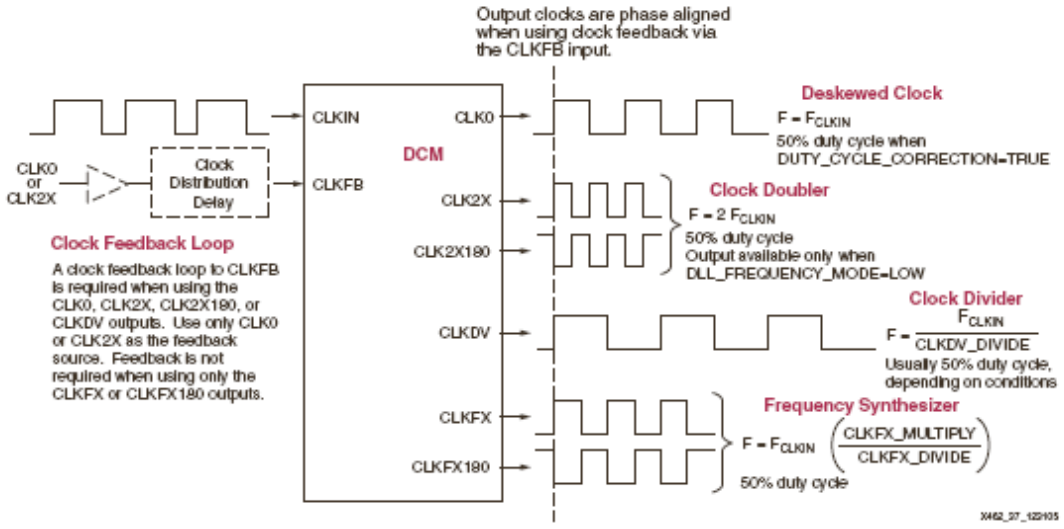


Figura 42: Relojes de salida del DCM [R11]

Entre otras FPGAs, Spartan-3, Virtex-II y Virtex-II Pro poseen DCMs.

Las funciones principales de un DCM son:
- Eliminación de *clock-skew*, para ello usa un reloj realimentado (CLKFB) con el que establece un desfase entre CLKFB y CLK0 de cero grados.
- Síntesis de Frecuencia, el modulo DCM puede generar diferentes frecuencias a partir de una frecuencia de entrada (CLKIN) multiplicándola o dividiéndola por distintos factores.
- Cambio de Fase, todas las señales de salida pueden contener distintas fases con respecto a la de entrada.

El significado de cada una de las entradas y salidas del DCM es el siguiente:

Tabla 7: Entradas de un módulo DCM

Entradas	Descripción
CLKIN	- Reloj de entrada al DCM - Siempre requerido - Su limite de frecuencia está controlado por los atributos DLL_FREQUENCY_MODE y DFS_FREQUENCY_MODE
CLKFB	- Reloj de entrada realimentado al DCM - Requerido a no ser que solo se utilicen CLKFX y CLKFX180 del DFS - Su origen debe ser la salida CLK0 o CLK2X del DCM y el atributo CLK_FEEDBACK debe terner un valor de 1x o 2x respectivamente
RST	- Entrada de reset asíncrona - Resetea la lógica del DCM a su estado de postconfiguración - Provoca que el DCM tenga que adquirir y bloquear la entrada CLKIN
PSEN	- Capacitación del PS
PSINCDEC	- Incrementa o decrementa el PS
PSCLK	- Reloj de entrada del PS

Tabla 8: Salidas de un módulo DCM

Salidas	Descripción
CLK0	- Reloj de la misma frecuencia que CLKIN y sin desplazamiento de fase
CLK90	- Reloj de la misma frecuencia que CLK0 desplazado 90º respecto a CLK0
CLK180	- Reloj de la misma frecuencia que CLK0 desplazado 180º respecto a CLK0
CLK270	- Reloj de la misma frecuencia que CLK0 desplazado 270º respecto a CLK0
CLK2X	- Reloj en fase con CLK0 y del doble de frecuencia
CLK2X180	- Reloj del doble de frecuencia que CLK0 desplazado 180º respecto a CLK2X
CLKDV	- Reloj en fase con CLK y de frecuencia 1/n respecto a la frecuencia de CLK0, siendo n el valor determinado por el atributo CLKDV_DIVIDE
CLKFX	- Reloj sintetizado y controlado por los atributos CLKFS_MULTIPLY y CLKFX_DIVIDE
CLKFX180	- Reloj de la misma frecuencia que CLKFX y desplazado 180 grados

	respecto a CLKFX.
STATUS[0]	- Señal de desbordamiento del PS. Se activa cuando el PS ha alcanzado su valor mínimo o máximo.
STATUS[1]	- Indicador de parada de la entrada CLKIN
STATUS[2]	- Indicador de parada de las salidas CLKFX y CLKFX180
STATUS[7:3]	- Reservados
LOCKED	- Se activa cuando las señales CLKIN y CLKFB están en fase
PSDONE	- Indica que la operación de desplazamiento de fase se ha completado

Debido a que el DCM tiene múltiples modos de funcionamiento existen una serie de atributos que permiten configurarlo. Estos atributos son los siguientes:

Tabla 9: Atributos de un módulo DCM

Atributos	Descripción
DLL_FREQUENCY_MODE	- Especifica el rango de frecuencias permitidas para el reloj de entrada CLKIN y las salidas de la unidad DLL. LOW: - Valor por defecto - El DLL funciona en su modo de frecuencia baja - Todas las salidas del DLL están disponibles HIGH: - El DLL funciona en su modo de frecuencia alta - Las salidas CLK2X y CLK2X180 del DLL no están disponibles
CLKIN_PERIOD	- Especifica en ns el periodo de reloj usado en el pin CLKIN del DCM
CLK_FEEDBACK	- Define la frecuencia del reloj de realimentación
DUTY_CYCLE_CORRECTION	- Capacita o descapacita el reparto del ancho de pulso del 50% de los relojes de salida del DLL
CLKDV_DIVIDE	- Define la frecuencia de el reloj de salida CLKDV - Los valores permitidos son 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8, 9, 10, 11, 12, 13, 14, 15, 16
CLKFX_MULTIPLY	- Define el factor de multiplicación de la frecuencia de los relojes CLKFX y CLKFX180 - Por defecto vale 4 pero puede tomar cualquier valor entero entre 2 y 32 - Se usa junto con CLKFX_DIVIDE
CLKFX_DIVIDE	- Define el factor de division de la frecuencia de los relojes CLKFX y CLKFX180 - Por defecto vale 1 pero puede tomar cualquier valor entero entre 1 y 32 - Se usa junto con CLKFX_DIVIDE
PHASE_SHIFT	- Define el retardo que hay en el flanco de subida entre el reloj CLKIN y los relojes de salida del

	DCM - Es aplicable solo si el atributo CLKOUT_PHASE_SHIFT tiene como valor FIXED o VARIABLE - Se especifica como un entero que representa una fracción del periodo de reloj. Por defecto toma el valor 0 aunque puede estar comprendido entre -255 y 255
DESKEW_ADJUST	- Controla el retardo entre el pin del reloj de entrada de la FPGA y las salidas de reloj del DCM

El funcionamiento del DCM es el siguiente: toma un reloj de entrada de referencia en el puerto CLKIN y saca una copia retardada de este reloj es su puerto CLK0 (o CLK2X). Esta salida es generalmente usada en un árbol de distribución de reloj (BUFG o buffer de reloj global). Una salida del árbol de distribución se usa en la entrada de realimentacion (CLKFB) del DCM. El retardo entre los puertos CLKIN y CLK0 se ajusta automáticamente por el lazo de realimentación hasta que las entradas CLKIN y CLKFB están en fase. Una vez que la diferencia de fase entre estas señales es mínima se dice que el DCM está bloqueado, lo que quiere decir que las salidas del árbol de distribución están en fase con la señal de entrada de reloj. El retardo de propagación a través de dicho árbol se cancela ya que el retardo de propagación total entre el puerto CLKIN del DCM y la salida del árbol es exactamente de un ciclo o un múltiplo.

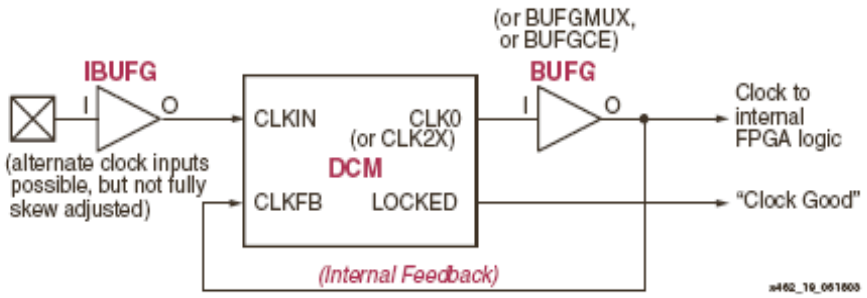


Figura 43: Realimentación de relojes en un DCM [R11]

Un DCM está formado por cuatro componentes principales:

- Delay Locked Loop (DLL)
- Digital Frequency Synthesizer (DFS)
- Phase Shifter (PS)
- Lógica de Estado

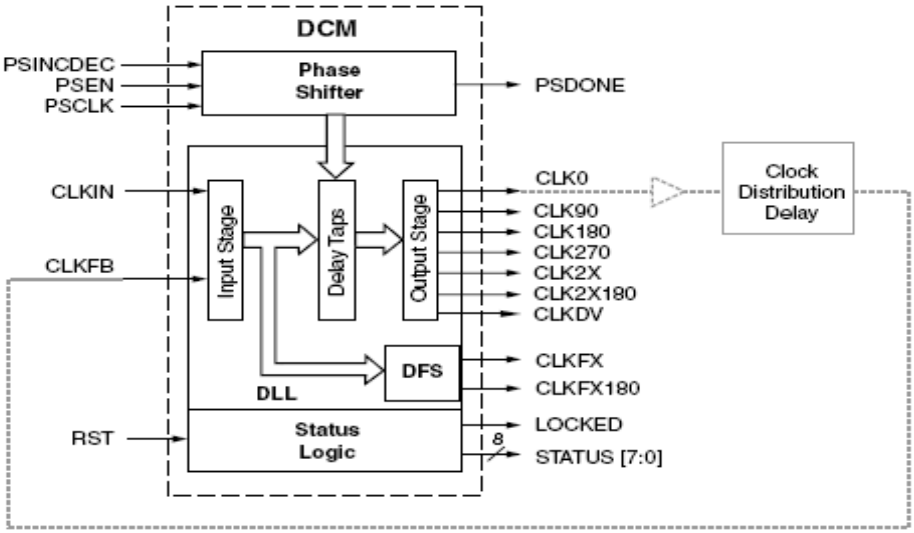


Figura 44: Componentes de un DCM [R11]

A continuación explicamos cada uno de los componentes que forman el DCM:

- DLL:

El DLL se encarga de eliminar el retardo de los relojes externos. Tiene como entradas CLKIN y CLKFB y como salidas CLK0, CLK90, CLK180, CLK270, CLK2X Y CLK2X180.

El DCM soporta dos frecuencias para el DLL. Por defecto, el atributo DLL_FREQUENCY_MODE está en baja por lo que la frecuencia de la señal de reloj de entrada CLKIN debe ser la menor comprendida en el rango de frecuencias determinado por los atributos DLL_CLKIN_MIN_LF y DLL_CLKIN_MAX_LF. En este caso, las salidas CLK0, CLK90, CLK180, CK270, CLK2X, CLKDV y CLK2X180 están disponibles. Por el contrario, cuando el atributo DLL_FREQUENCY_MODE está en alta, la frecuencia de la señal CLKIN debe ser la mayor comprendida entre DLL_CLKIN_MIN_HF y DLL_CLKIN_MAX_HF y solo las salidas CLK0, CLK180 y CLKDV están disponibles.

El reparto del ancho de pulso de la salida CLK0 es 50-50, a menos que el atributo DUTY_CYCLE_CORRECTION este a falso, en cuyo caso es el mismo que el de la entrada CLKIN. El de las salidas desfasadas CLK90, CLK180 y CLK270 es el de de la salida CLK0. Para las salidas CLK2X, CLK2X180 y CLKDV el reparto es

también 50-50 a no ser que el atributo CLKDV_DIVIDE no sea un entero y el DLL_FRECUENCY_MODE este en alto. La frecuencia de la salida CLKDV está determinada por el valor asignado al atributo CLKDV_DIVIDE.

- DFS:

El DFS proporciona un amplio y flexible rango de frecuencias basadas en los dos valores enteros definidos por el usuario que son los atributos CLKFX_MULTIPLY y CLKFX_DIVIDE. Genera las salidas CLKFX y CLKFX180 que se pueden usar simultáneamente. El reparto del ancho de pulso de ambas es de 50-50.

La frecuencia de salida se deriva de la entrada de reloj CLKIN mediante multiplicaciones y divisiones de frecuencia simultáneamente.

El DLL puede ser utilizado o no. Si se utiliza, CLKFX y CLKIN están en fase cada CLKFX_MULTIPLY ciclos de CLKFX y cada CLKFX_DIVIDE ciclos de CLKIN cuando se realimenta la entrada CLKFB del DDL. Por el contrario, si el DLL no se usa no existe relación de fase entre la entrada CLKIN y las salidas del DFS.

La frecuencia de CLKFX se calcula con la siguiente ecuación:

$$Frecuencia_{CLKFX} = CLKFX\_MULTIPLY\_value / CLKFX\_DIVIDE\_value * Frecuencia_{CLKIN}$$

- PS:

Es la parte encargada de controlar la relación de fases entre las salidas del DCM y la entrada CLKIN. Para ello cambia la fase de los nueve relojes de salida por una fracción fija del período de reloj de la entrada. El valor de esta fracción se establece en tiempo de diseño.

Las entradas del PS son PSINCDEC, PSEN y PSCLK y las salidas son PSDONE y la STATUS[0].

- Lógica de estado:

La lógica de estado indica el estado actual del DCM mediante las señales de salida de LOCKED y STATUS [0-2]. La señal de LOCKED indica cuando las salidas de reloj del DCM están en fase con la entrada CLKIN. La señal STATUS indica el estado del DLL y las operaciones del PS.

La entrada RST resetea el DCM y lo devuelve a su estado de post-configuración. Asimismo, el reset fuerza al DCM a readquirir y bloquear la entrada CLKIN.

Capítulo 3: Herramientas software

Para poder llevar a cabo el proyecto, a pesar de ser de tipo hardware, ha sido imprescindible el uso de herramientas software tanto para el desarrollo y diseño de la arquitectura del sistema, la simulación de los componentes y verificación de su comportamiento, así como para cargar los mapas de bits en la FPGA para probar el funcionamiento real sobre el sistema.

Dentro de las herramientas de desarrollo nos hemos basado en los Entornos Integrados de Software (ISE) de Xilinx que ofrecen multitud de herramientas de edición, diseño… así como la posibilidad de integración con simuladores y otro tipo de software.

Para la simulación optamos por el uso de ModelSim ya que a parte de integrarse con el ISE, es muy sencillo de usar y tiene múltiples funciones que proporcionan una gran potencia.

El proceso de carga de los mapas de bits sobre la FPGA lo hemos realizado utilizando la herramienta iMPACT que está incluida dentro del ISE. Muestra una visión gráfica de los dispositivos conectados y de una forma rápida permite cargar los archivos binarios sobre los dispositivos.

Por último describiremos el funcionamiento de la aplicación MIG de Xilinx que permite la generación de interfaces de memoria y que hemos utilizado para nuestro controlador de memoria DDR.

Vamos a entrar un poco más en detalle de las herramientas utilizadas mostrando y explicando algunas de las funcionalidades que hemos utilzado.

3.1. Xilinx ISE

El entorno de desarrollo Xilinx ISE (Integrated Software Environment) es una herramienta de diseño de circuitos digitales. Permite la creación, verificación, simulación y síntesis de un proyecto basado en un CPLD o una FPGA.

Se divide en dos secciones principales:

Project Navigator: Editor

ModelSim: Simulador

El proceso de diseño, síntesis e implementación de circuitos digitales en dispositivos de lógica programable incluye los siguientes pasos:

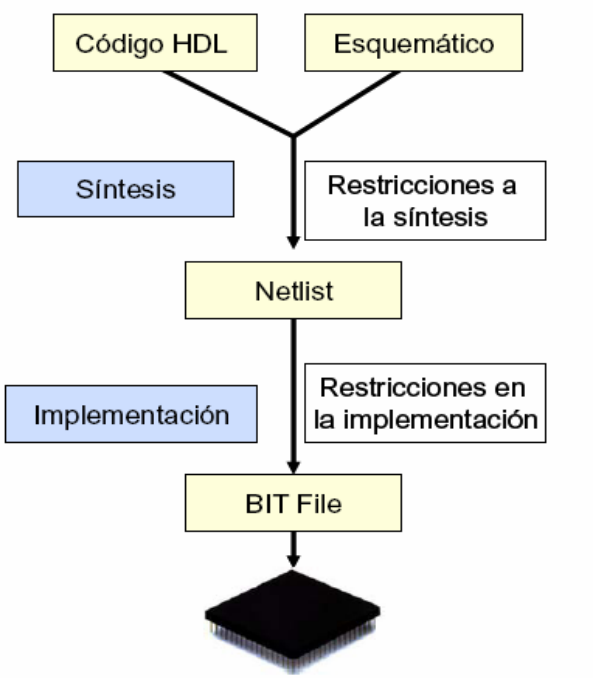


Figura 45: Pasos del proceso de diseño de circuitos digitales

El ISE de Xilinx nos proporciona las herramientas necesarias para poder ejecutar cada uno de estos pasos.

3.1.1. Project Navigator

Es el editor donde se realizará el diseño del circuito, bien mediante un esquemático, o bien utilizando un lenguaje de descripción de hardware (en nuestro caso VHDL).

Una vez iniciado el entorno de desarrollo, aparece la siguiente interfaz:

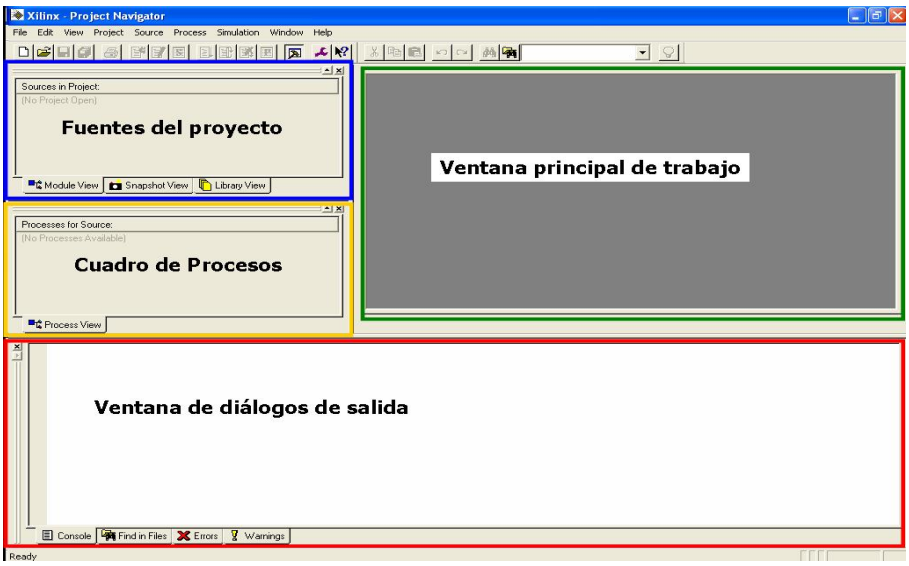


Figura 46: Ventana principal del Project Navigator

Como puede observarse está dividida en cuatro secciones:

- Fuentes del proyecto ("Sources in Project"): Contiene todos los elementos que forman parte del proyecto ordenados de manera jerárquica.
- Cuadro de procesos ("Processes for Current Source"): Muestra todos los procesos disponibles para un determinado elemento del proyecto.
- Ventana de diálogos de salida ("Console"): Despliega los mensajes de estado, errores y advertencias.
- Ventana principal de trabajo: Ventana que permite desplegar diferentes tipos de documentos como archivos de texto ASCII (los que contienen los programas), formas de onda para simulación, resultados en html o texto, entre otros.

Creación de un proyecto

Para crear un proyecto nuevo los pasos a seguir son:

(1) Seleccionar File → New Project. Aparece el siguiente cuadro de diálogo en el que habrá que especificar el nombre del proyecto, el directorio donde se va a guardar y el tipo de proyecto.

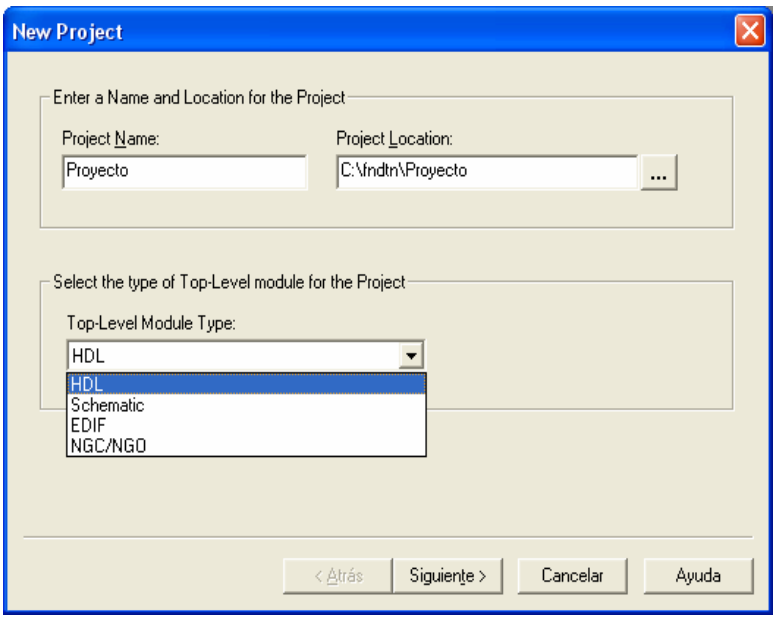


Figura 47: Creación de nuevo proyecto (paso 1)

(2) El siguiente paso es seleccionar el tipo de dispositivo a utilizar

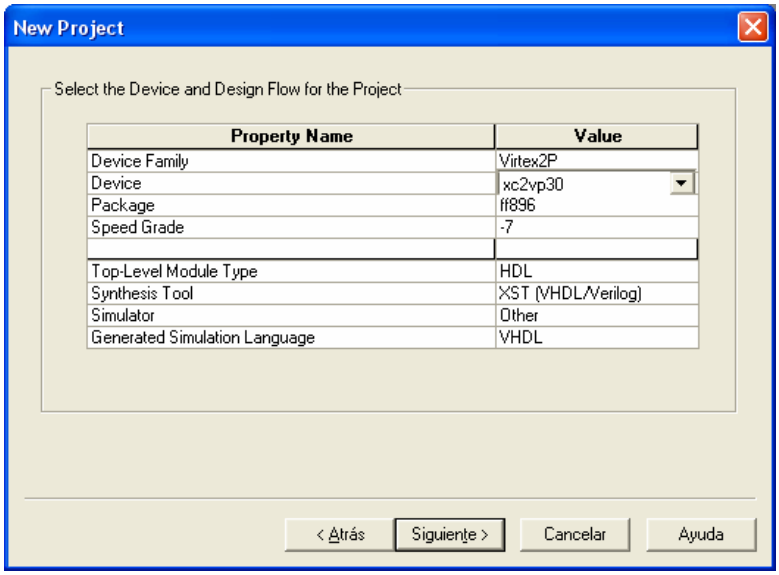


Figura 48: Creación de nuevo proyecto (paso 2)

(3) A continuación se definen los ficheros fuentes para el diseño. Se selecciona Project → New Source. Lo primero que hay que hacer es definir el fichero fuente del nuevo módulo y a continuación el nombre de la entidad y arquitectura de dicho módulo junto con sus puertos de entrada y salida

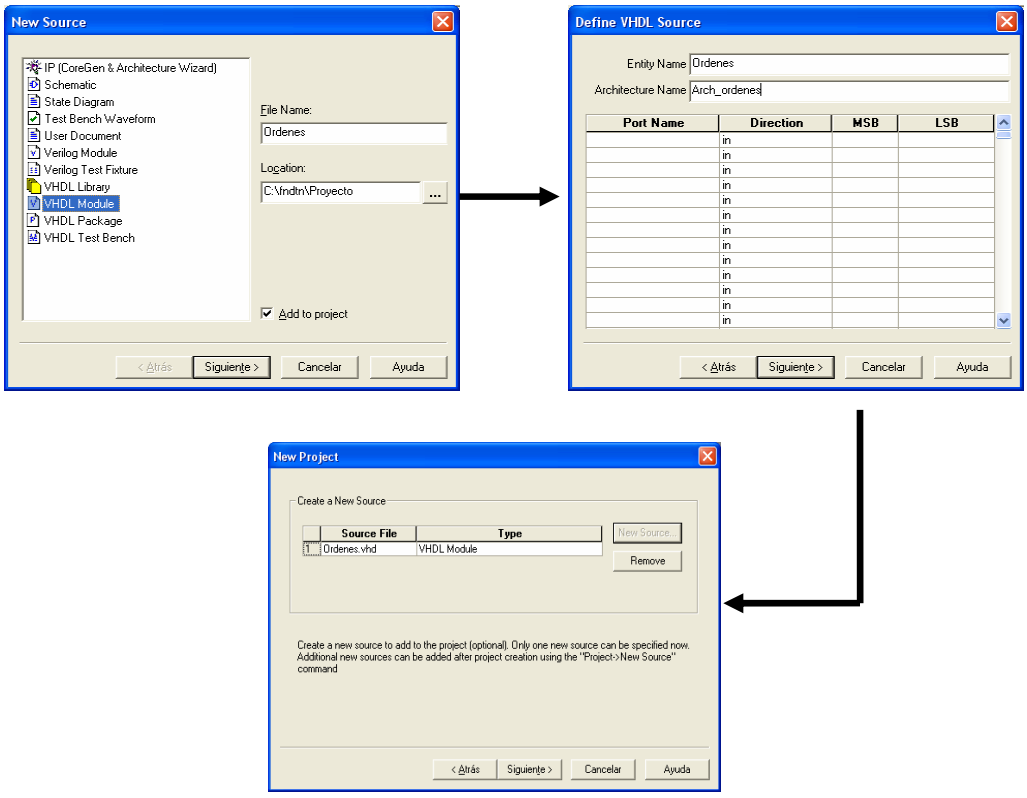


Figura 49: Creación de nuevo proyecto (paso 3)

También se pueden añadir ficheros ya existentes.

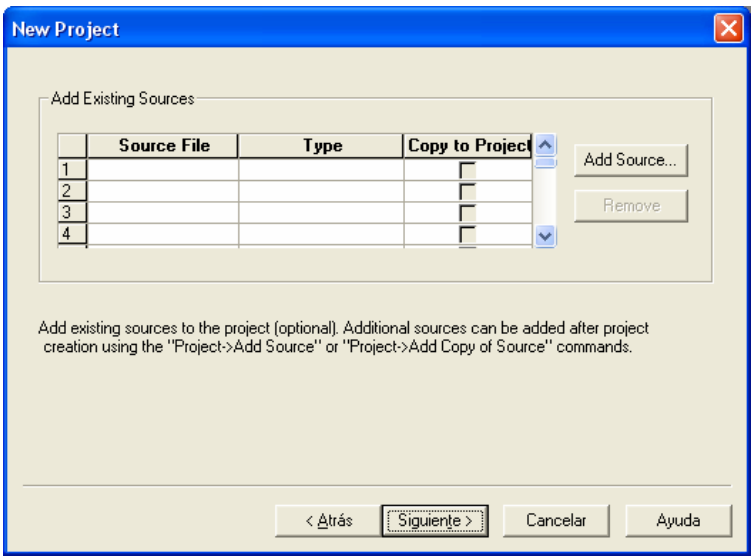


Figura 50: Creación de nuevo proyecto (paso 4)

Una vez que se han creado o añadido los ficheros fuente necesarios se muestra la siguiente ventana indicando toda la información del proyecto creado.

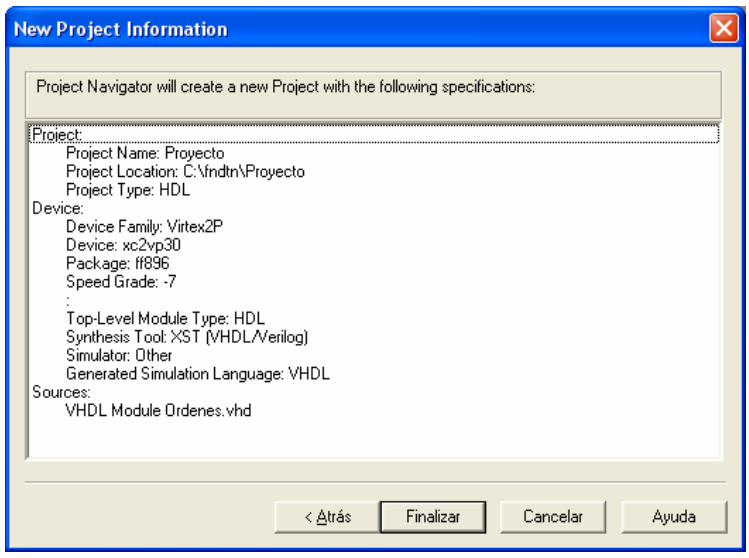


Figura 51: Creación de nuevo proyecto (paso 5)

Síntesis

El siguiente paso en el diseño de un circuito digital es el proceso de síntesis, mediante el cual se obtiene una descripción física a nivel circuital a partir de una descripción conductual lo más abstracta posible.

Para realizar el proceso de síntesis mediante el ISE hay que seleccionar la opción *Synthesize-XST* de la lista de procesos disponibles:

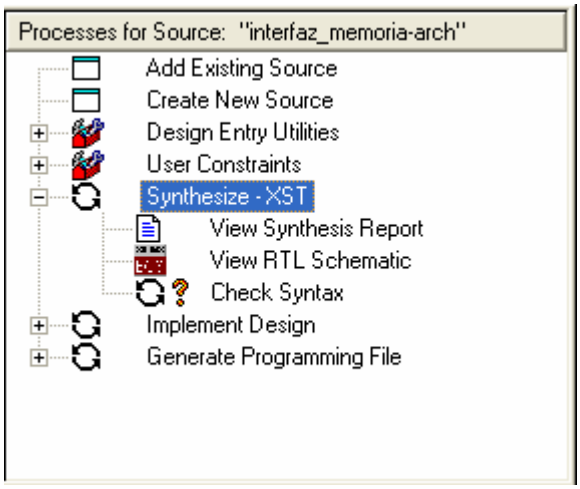


Figura 52: Proceso de síntesis

Implementación

A continuación, en el diseño hay que realizar el proceso de implementación, el cual está dividido en tres fases:

- Translate: Mezcla los netlists de todos los componentes del diseño con las restricciones contenidas en el fichero .ucf y genera como salida un fichero NGD (Native Generic Database), que describe el diseño lógico reducido a primitivas de Xilinx. El fichero ucf es un fichero de texto que especifica las restricciones del diseño lógico. Si no se incluye en el proyecto, no habrá restricciones y por lo tanto la herramienta ISE tomará las que crea convenientes.

- Map: Crea un fichero NCD (Native Circuit Description), el cual representa la descripción física del circuito de entrada pero aplicada al dispositivo especifico sobre el que se desea trabajar.

- Place & Route: Usa el fichero NCD generado en la fase Map para colocar y rutar el diseño.

Para llevar a cabo la implementación hay que seleccionar la opción *Implement Design* de la lista de procesos disponibles:

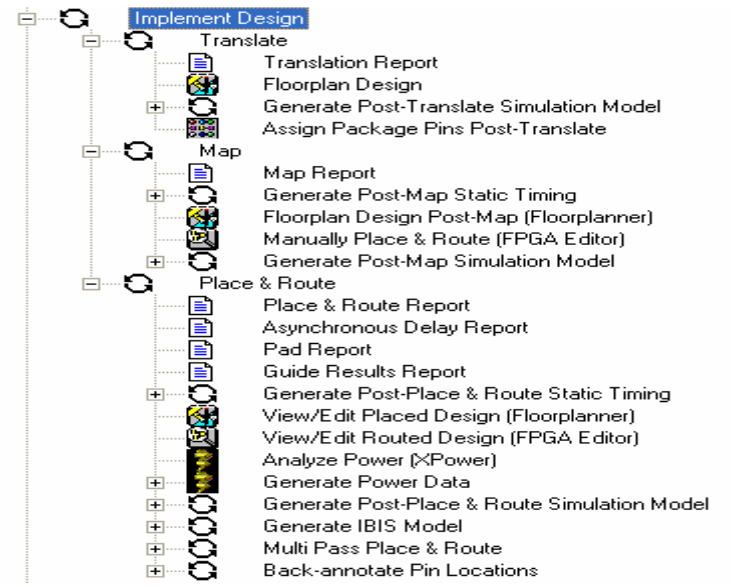


Figura 53: Proceso de implementación

Generación fichero .bit

Por último, una vez que el diseño ha sido rutado en la FPGA, se realiza el proceso de generación del bitstream o fichero .bit, el cual contiene la información de localización en el dispositivo, esto es la colocación de los CLBs, los IOBs, los TBUFs, pines y elementos de rutado.

Para llevar a cabo la este paso hay que seleccionar la opción *Generate Programming File* de la lista de procesos disponibles:

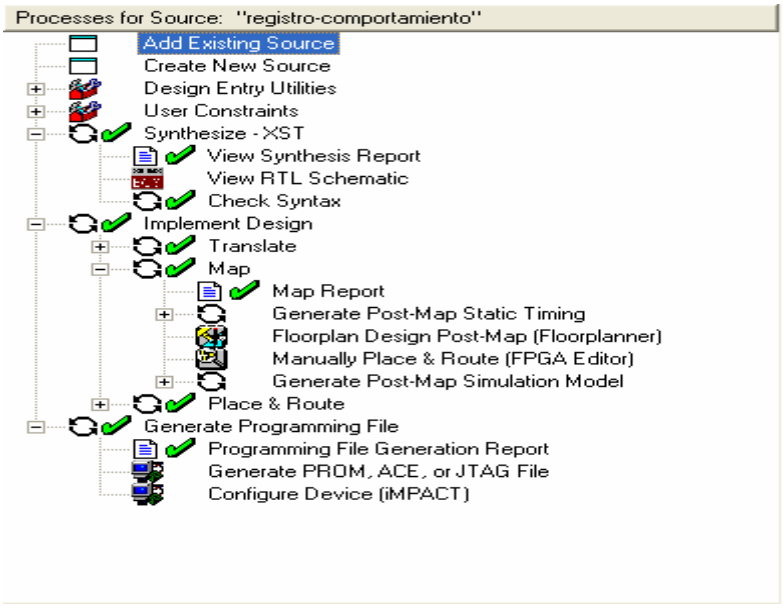


Figura 54: Proceso de generación del .bit

3.1.2. ModelSim

Es la herramienta que permite realizar la simulación del funcionamiento del circuito y de este modo comprobar si funciona según las especificaciones establecidas.

Entre las distintas funcionalidades que ofrece la aplicación nosotros la hemos utilizado para verificar si el comportamiento de nuestros circuitos era el esperado.

Para realizar estas comprobaciones es necesario seguir los siguientes pasos:

(1) Compilar el fichero fuente VHDL que contenga el comportamiento del circuito que se desee simular

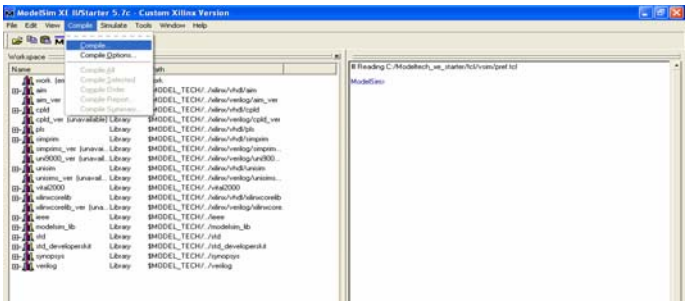


Figura 55: Compilación de un fichero fuente VHDL

(2) Una vez compilado el fichero fuente ya se puede realizar su simulación. Para ello se selecciona el componente y se elige la opción simular

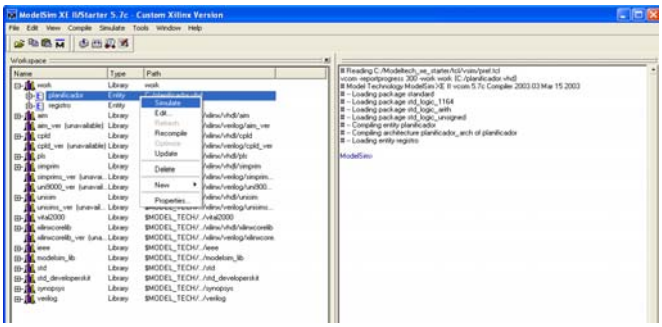


Figura 56: Simulación de un fichero VHDL

(3) Tras esto se abrirá una nueva ventana en la que se muestra el árbol de componentes incluidos dentro del fichero

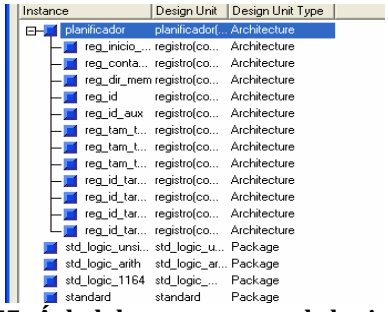


Figura 57: Árbol de componentes de la simulación

(4) A partir de este momento habría que asignar a cada señal los valores deseados e ir ejecutando ciclos de simulación. Durante la ejecución es posible modificar los valores de las señales. Existen varias posibilidades para realizar esto y en nuestro caso lo hemos hecho utilizando un fichero de script, que contiene toda la asignación de señales.

(5) Tras ejecutar el script obtenemos como resultado el diagrama temporal de la simulación

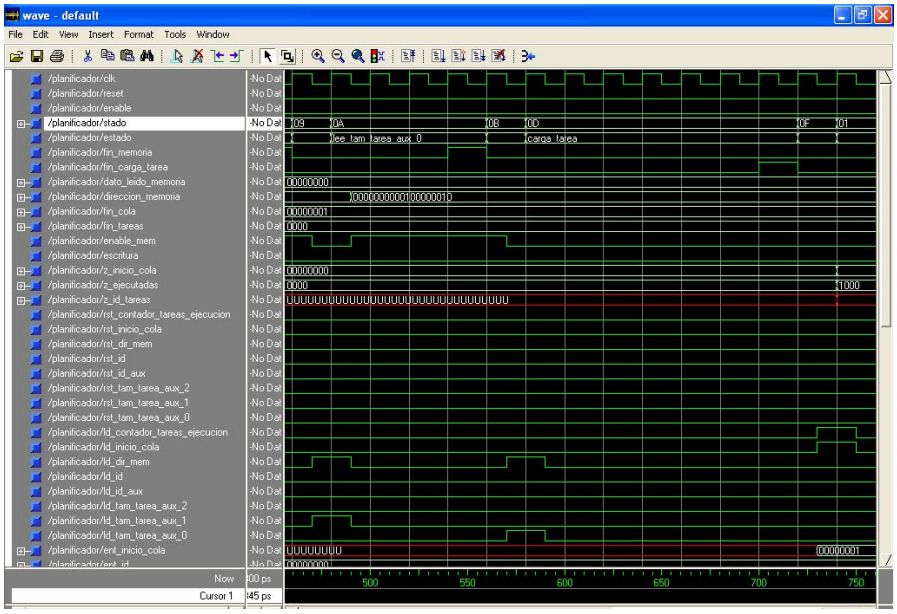


Figura 58: Resultado de la simulación

3.2. iMPACT

La herramienta iMPACT permite realizar la comunicación entre un ordenador y las placas de prototipado.

Una vez abierto, detecta y muestra los dispositivos conectados.

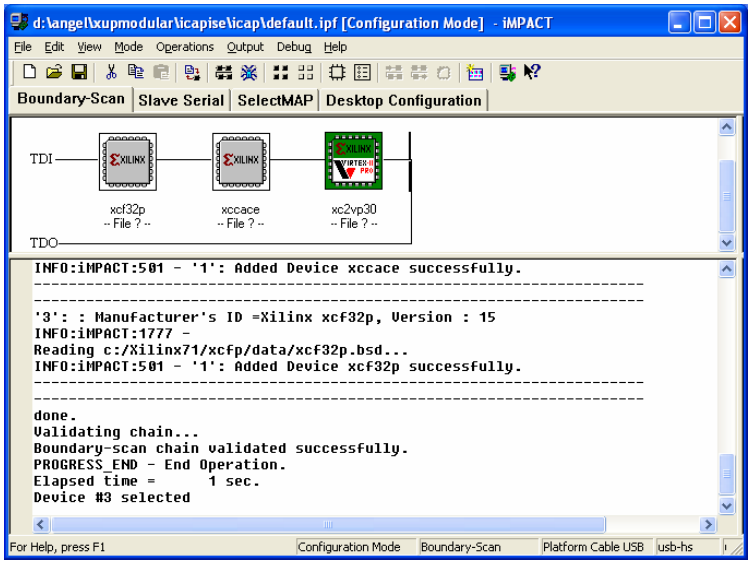


Figura 59: Ventana principal de la aplicación iMPACT

La herramienta permite realizar diferentes acciones sobre cada uno de los dispositivos detectados. Estas pueden ser cargar un mapa de bits, verificar su funcionamiento, etc.

En nuestro caso, solamente hemos utilizado la opción de cargar el mapa de bits. Para ello hay que seleccionar el dispositivo sobre el que se desee trabajar y se le asigna el mapa de bits correspondiente del circuito que se quiere cargar. Una vez hecho esto, se selecciona la opción *program* para terminar el proceso de carga sobre el dispositivo.

3.3. MIG 007

El MIG 007 (Memory Interface Generator) es una aplicación proporcionada por Xilinx que nos permite generar interfaces de memoria tanto para memorias DDR SDRAM en el caso de las placas Xilinx Spartan 3, Virtex II y Virtex II Pro, como para DDR2 SDRAM en el caso de la Virtex II y Virtex II Pro. [R9]

Para ello la aplicación toma como entrada una serie de datos proporcionados por el usuario a través de una interfaz gráfica (GUI) y con ellos genera los archivos RTL (en Verilog o VHDL), SDC, UCF y los ficheros de documentos.

Tiene tres modos de operación diferentes: modo generador, modo editor de pines y modo placa.

Modo generador

Es el modo por defecto. Coloca los pines en los bancos seleccionados para datos, direcciones y control.

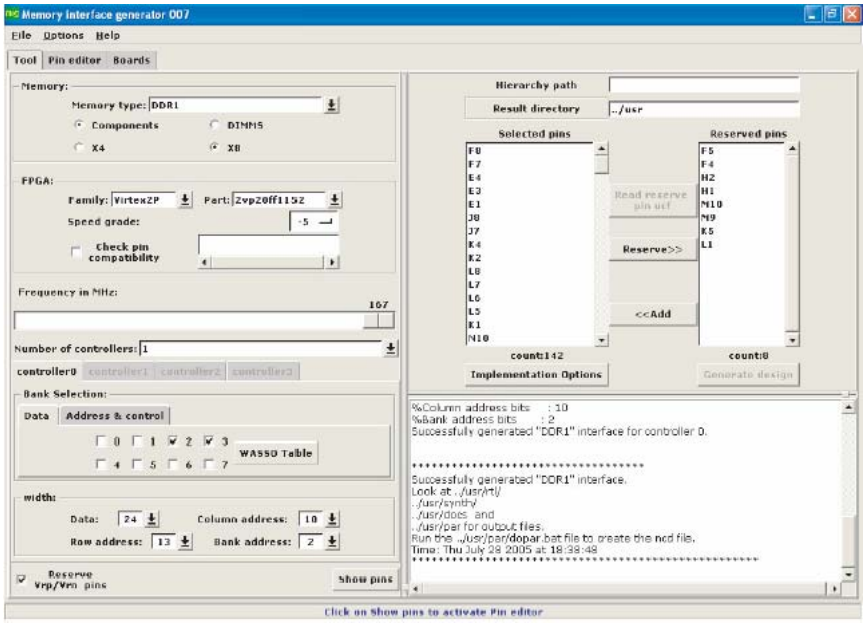


Figura 60: Ventana principal de la aplicación MIG

Mediante esta interfaz se permite al usuario seleccionar los siguientes parámetros:

- Tipo de memoria que puede ser DDR1, DDR2 diferencial y DDR2 no diferencial
- Components / DIMMS
- X4 / X8 para indicar el número de bits de datos por strobe
- Familia de la FPGA: Spartan 3, Virtex II o Virtex II Pro
- Número de dispositivo de la FPGA
- Grado de velocidad de la FPGA (-4 y -5 para Spartan 3, -4, -5 y -6 para Virtex II y -4, -5, -6 y -7 para Virtex II Pro)

- Compatibilidad de pines (opcional)
- Frecuencia del interfaz que puede ser un valor comprendido entre 100 y 230 MHz
- Número de controladores
- Bancos de datos y señales de strobe de datos
- Bancos para direcciones y señales de control
- WASSO (weighted average simultaneous switching output)
- Ancho de datos
- Ancho de direcciones de fila
- Ancho de direcciones de columna
- Ancho de direcciones de banco
- Reserva y uso de los pines V_{RP} y V_{RN}
- Ruta dentro de la jerarquía
- Directorio destino
- UCF de pines reservados
- Opciones de implementación como son usar un DCM o añadir un test de prueba

Modo editor de pines

En este modo, la aplicación permite elegir al usuario los pines en los que desee colocar los datos, strobes, direcciones y señales de control. Se usa después de seleccionar todos los parámetros anteriores.

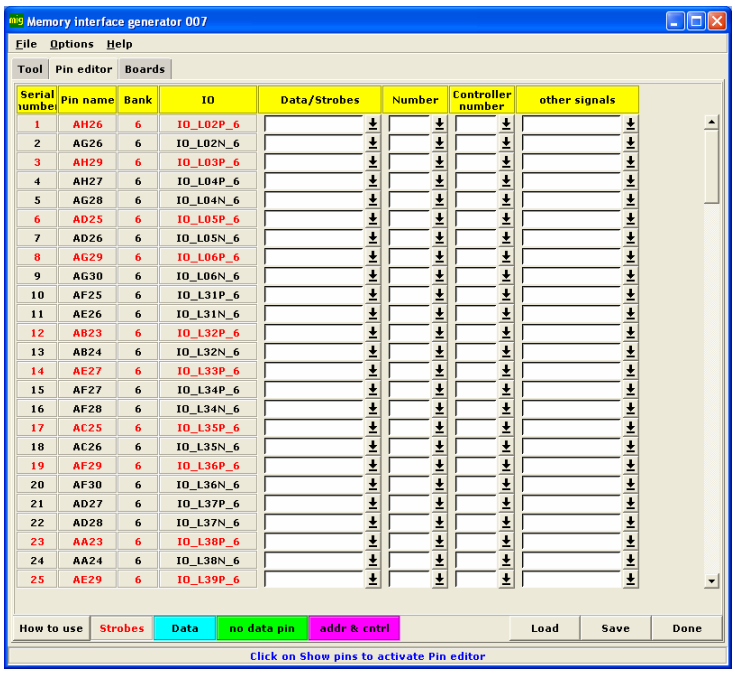


Figura 61: Modo editor de pines

Modo placa

Permite al usuario seleccionar el tipo de placa para la cual desee generar el interfaz. Se puede elegir entre tres tipos de placas: SL361, ML361 y ML367, cada una de 166

MHz, 167 MHz y 200MHz respectivamente y genera como salida el UCF y el fichero de mapa de bits.

Forma de uso

Los pasos que hay que seguir para generar el interfaz de memoria son:

(1) Especificar todos los parámetros del diseño mediante el modo generador
(2) Pulsar el botón Show pins para activar el Pin editor
(3) Pulsar el botón Generate Design para generar el diseño y los ficheros de salida

Una vez realizados estos pasos todos los ficheros generados se guardan en el directorio de salida seleccionado. Si hay algún error en la selección de los parámetros se indica mediante un mensaje de error.

Para generar el controlador de la memoria que hemos utilizado en el proyecto seleccionamos los siguientes parámetros en el *modo generador*:

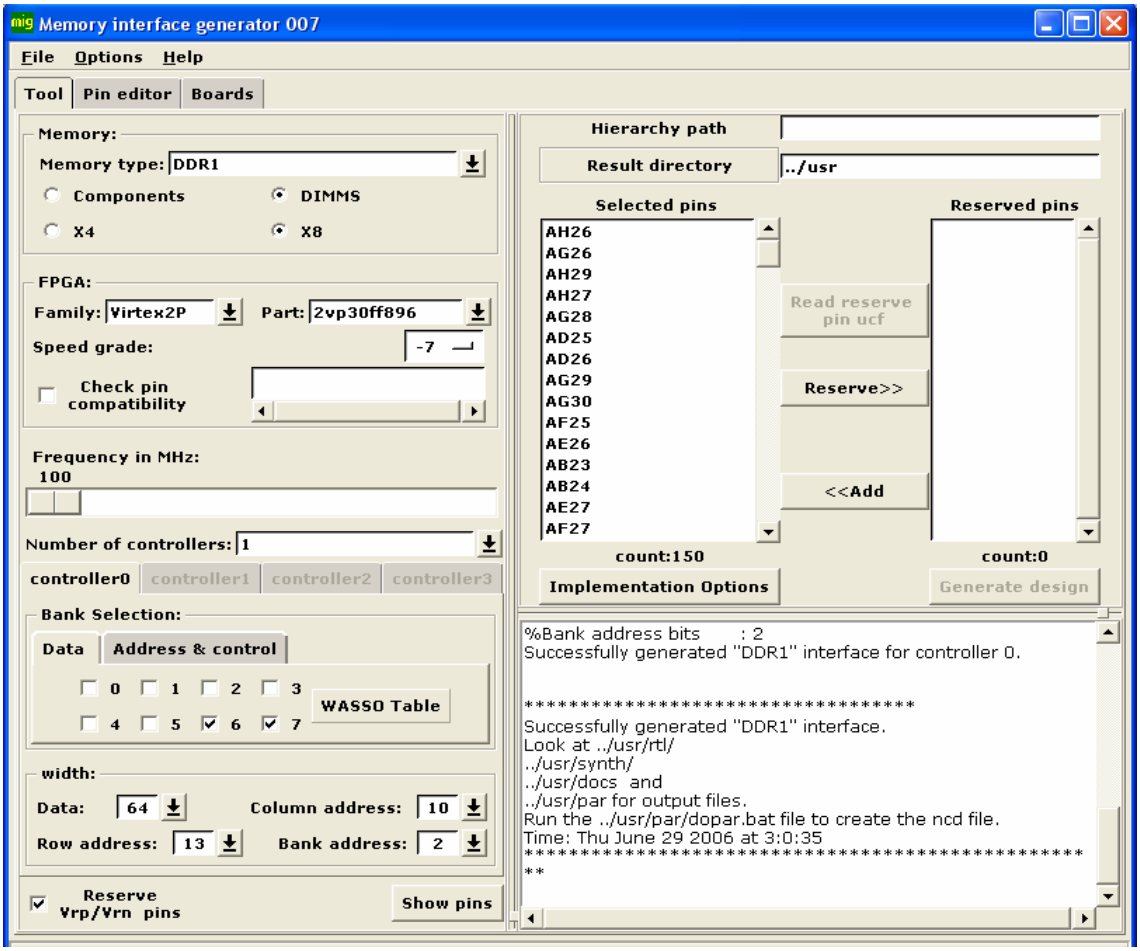


Figura 62: Ejemplo de uso del MIG (paso 1)

Antes de pulsar el botón Generate design para que nos genere el diseño debemos seleccionar mediante el modo *pin editor* la colocación de los pines propios de la FPGA sobre la que vamos a trabajar ya que por defecto la aplicación puede no hacerlo correctamente.

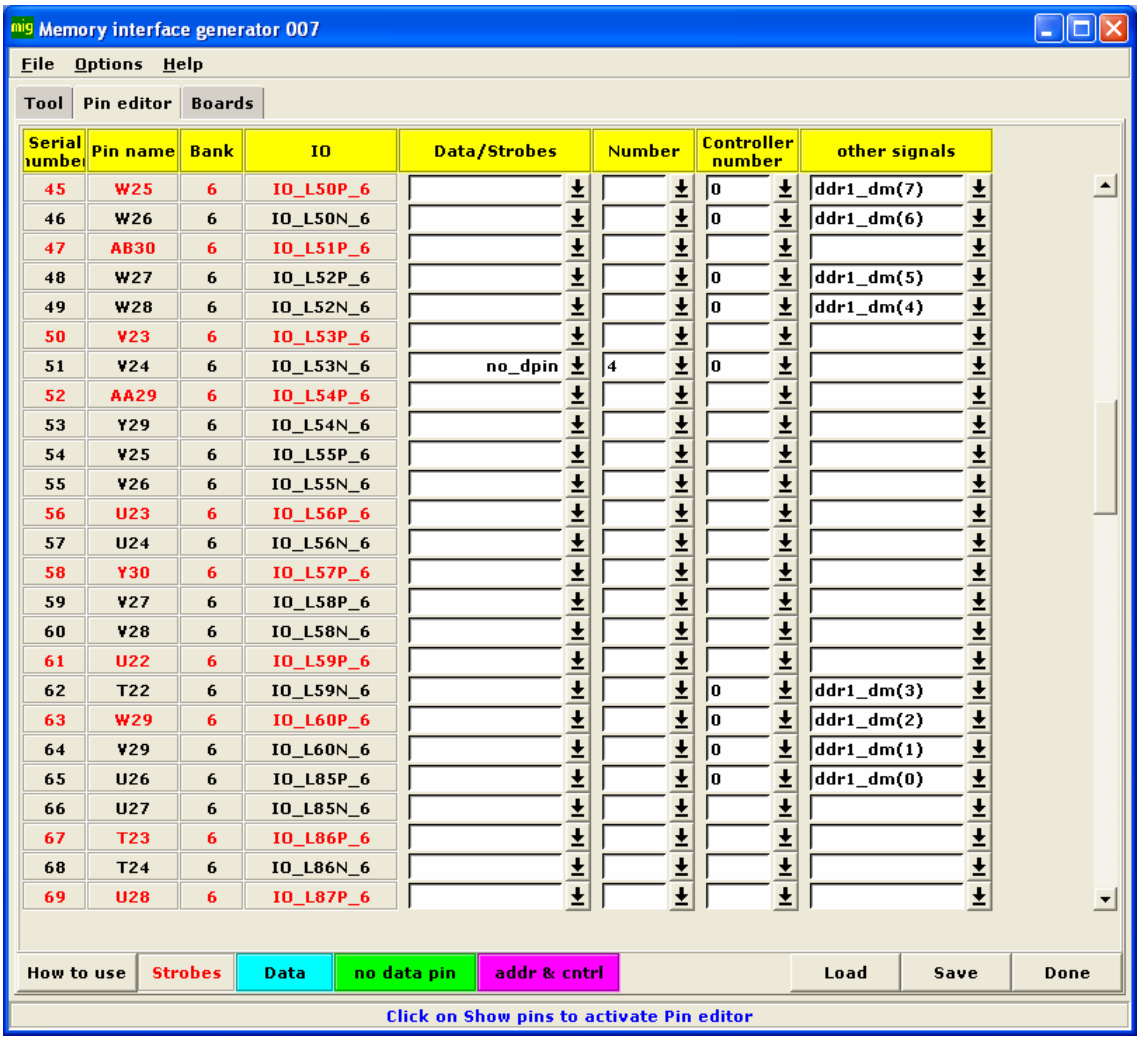


Figura 63: Ejemplo de uso del MIG (paso 2)

Por último volvemos al *modo generador* y ya podemos pulsar el botón Generate Design para generar el diseño final.

Capítulo 4: Desarrollo del proyecto

Parte 1

Desde que comenzamos por la idea inicial hasta que conseguimos el diseño final hemos pasado por diversas etapas. Unas veces era debido a que terminábamos una parte de la funcionalidad y teníamos que empezar con otra nueva que ampliara el prototipo, y otras veces porque surgían problemas que nos hacían volver atrás o cambiar la arquitectura del sistema para adaptarse a una nueva especificación funcional del sistema.

A continuación vamos a explicar todo el proceso que hemos seguido desde que comenzó el proyecto hasta el punto en el que quedó al final.

En primer lugar tuvimos una fase de aprendizaje. En esta fase tuvimos que refrescar los conocimientos que teníamos de VHDL ya que era el lenguaje de descripción que teníamos que utilizar para diseñar el sistema. Para ello implementamos un juego que funcionaba sobre una FPGA Spartan2 montada en una placa de prototipado XSA-100.

Una vez hecho esto y habiendo aprendido algunas características nuevas del lenguaje comenzamos realmente a diseñar el prototipo del sistema.

Debido a que una de las partes básicas era permitir que el usuario se pudiera comunicar con el sistema empezamos a diseñar un prototipo que permitía recoger los caracteres pulsados en un teclado PS/2 y los mostraba en un monitor VGA.

Para la creación de este primer prototipo tuvimos además que aprender a utilizar el entorno de desarrollo sobre el que íbamos a trabajar. Este entorno es el ISE 6 de Xilinx del que hablamos en el apartado de herramientas software.

También tuvimos nuestro primer contacto sobre la placa de prototipado que íbamos a utilizar durante el proyecto. La placa es la *MicroBlaze and Multimedia Development Board* de Xilinx que contiene una FPGA Virtex II de la que hablamos en el apartado de entorno de trabajo.

En el desarrollo de este interfaz de comunicaciones tan simple no tuvimos demasiados problemas. Los más destacables fueron todos aquellos que tenían que ver con la utilización del entorno de desarrollo y la secuencia de pasos que había que seguir para la generación del mapa de bits del diseño.

Una vez que obtuvimos este primer mapa de bits, lo cargamos utilizando la herramienta *iMPACT* (explicada en el apartado de herramientas software) y comprobamos que nuestro prototipo funcionaba.

Después de conseguir esta interacción entre el usuario y el sistema la aplicamos para mejorar nuestro juego diseñado para Spartan2. Cambiamos la implementación del juego para que funcionara en la Virtex II y además para que la entrada/salida no fuese por medio de switches y leds sino que se realizara utilizando el teclado y la pantalla que acabábamos de aprender a utilizar.

Una vez hecho esto, en lo que no encontramos mayor dificultad, comenzamos realmente con el desarrollo del sistema objetivo de nuestro proyecto.

Parte 2

Siguiendo la especificación del proyecto se puede obtener la idea clave de su comportamiento que es que en un instante de tiempo el usuario mediante la emisión de una orden transmitida utilizando el teclado indica al sistema que debe tomar una tarea almacenada en memoria y cargarla sobre la FPGA. Esta idea nos da una serie de pasos que debemos seguir a partir de ahora para continuar con nuestro desarrollo.

Como la parte de utilización del teclado de forma básica ya la teníamos controlada, lo siguiente que debíamos hacer es que nuestro sistema fuese capaz de comprender órdenes concretas escritas por el usuario y descartar aquellas que no tuvieran un significado válido.

Por otra parte, también vimos que sería importante que nuestro sistema fuese capaz de comunicarse con un nuevo dispositivo, la memoria RAM. A la descripción del funcionamiento de este tipo de memoria le dedicamos un apartado del capítulo referente a la memoria RAM.

Con estas dos cosas que teníamos pendientes optamos por comenzar el desarrollo de nuestro sistema con un diseño que permitiera al usuario leer y escribir en la memoria en función de una orden escrita por teclado.

Las funciones que al final implementamos fueron leer una palabra de un byte contenida en una dirección memoria, cargar una palabra de un byte en una dirección de memoria y cargar *n* palabras de forma consecutiva a partir de una dirección de memoria.

El formato de las órdenes que el usuario podría introducir al sistema era el siguiente:

- Para leer un byte de una dirección de memoria el usuario debe escribir:
    - *LEER DIRECCION*
        - Donde *DIRECCION* es una secuencia de 8 dígitos hexadecimales que representan una dirección de 32 bits.
- Para cargar un byte a una dirección de memoria el usario debe escribir:
    - *CARGAR DIRECCION DATO*
        - Donde *DATO* es el dato que se escribirá de un byte formado por 2 dígitos hexadecimales.
        - *DIRECCION* es la direccion formada por 8 dígitos hexadecimales en la que se escribe el dato.

- Para cargar *n* bytes consecutivos a partir de una dirección de memoria el usuario debe escribir:

    - *CARGARN DIRECCION NUMERO_DATOS DATO1 DATO2 …DATOn*

        - Donde *NUMERO_DATOS* es el número total de datos que el usuario quiere cargar en la memoria. Es un valor de un byte formado por 2 dígitos hexadecimales.

        - *DATO1 DATO2 … DATOn* son los *n* datos que se escribirán en memoria. El valor de *n* viene dado por el valor establecido en *NUMERO_DATOS*. Cada dato va separado del resto por un espacio y es de un byte representado por 2 dígitos hexadecimales.

        - *DIRECCION* es la dirección a partir de la cual se escriben los *n* datos de manera consecutiva.

Creamos una serie de módulos para dar funcionalidad a esta parte. En primer lugar hicimos un módulo que reconociera el tipo de orden que se ha introducido a partir de un cierto momento. Este módulo va comprobando cada una de las pulsaciones realizadas y cuando el usuario pulsa la tecla espacio querrá decir que ya ha terminado de introducir la orden y pasa a introducir los datos propios de la orden. El módulo tiene tres señales de salida, una que se pone a 1 si se ha reconocido una orden válida, otra que se pone a 1 si se ha encontrado un error durante el reconocimiento de la orden, es decir, que la orden no es válida y la tercera que indica mediante 8 bits un valor codificado que representa la orden reconocida si todo ha ido bien.

El segundo módulo que implementamos se encargaba de interpretar los datos propios de cada orden y devolverlos. El funcionamiento básicamente consiste en reconocer a partir de un cierto momento el valor de un byte a partir de dos pulsaciones de teclado de caracteres hexadecimales. Este módulo también tiene tres salidas que indican si se ha producido un error que, en el caso de ocurrir, pondría a 1 la señal de error; otra que se pone a 1 si se ha reconocido un byte correctamente a partir de dos caracteres hexadecimales y otra que devuelve el valor del byte representado por los dos caracteres.

Una vez realizados estos dos módulos y comprobando en la simulación que su comportamiento era el esperado realizamos un tercer módulo que se encargaba de comunicarlos entre si. Este módulo o unidad de control se encargaba de que el proceso que se siguiera para reconocer cada una de las líneas de orden introducidas por el usuario fuese el siguiente:

- En primer lugar el usuario deberá introducir la orden que quiere ejecutar, la unidad de control resetea el módulo encargado de reconocer la orden y espera a que éste finalice bien sea por el reconocimiento de una orden correcta como porque se haya producido un error.

- Si la orden se reconoce, la unidad de control capacita al módulo que se encarga de leer los datos de la orden. Este módulo irá detectando cada uno de los bytes

cada dos pulsaciones y estos datos se irán almacenando de manera consecutiva en un banco de registros.

- Una vez que el usuario finalice de introducir los datos que necesite la orden pulsará la tecla intro y si todo ha ido bien el proceso finalizará con éxito.

Con estos módulos que acabamos de explicar teníamos un sistema que era capaz de reconocer una orden, sus datos y almacenar todo esto en registros, de forma que en el paso siguiente que teníamos que implementar, pudiéramos utilizarlos para llevar a cabo la ejecución de la orden concreta.

Antes de seguir intentamos probar nuestro prototipo y comprobamos que tras cargarlo en la FPGA su comportamiento no era del todo el esperado, tuvimos algunos problemas para que el controlador que se encarga de activar los módulos lo hiciera de la forma correcta. Una vez conseguido esto continuamos teniendo algunos problemas con cada uno de los módulos por separado ya que cuando uno terminaba su función y comenzaba el otro, el anterior detectaba que se había producido un error a pesar de que él ya no debería estar ejecutándose. Y también al revés, el módulo encargado de reconocer los datos de la orden comenzaba a funcionar antes de tiempo y activaba su señal de error cuando el usuario pulsaba una tecla que no correspondía con un carácter hexadecimal pero que no estaba mal pulsada ya que éste carácter realmente pertenecía a la cadena de la orden. Tras algunas modificaciones sobre el diseño original de los módulos conseguimos solucionar todos estos pequeños problemas, provocados en su mayor parte por la inexperiencia que teníamos en este punto en el desarrollo de sistemas hardware.

Comenzamos con la creación de un nuevo módulo de nuestro sistema que fuese capaz de decidir, a partir de los datos que habían recopilado los dos módulos anteriores, que es lo que había que hacer y activara las señales que permitieran realizarlo.

Básicamente su función iba a consistir en leer y escribir de memoria por lo que llegados a este punto tuvimos que aprender a controlar la memoria SDRAM de la placa.

Para controlar la memoria que tiene la placa incorporada hicimos uso de un interfaz de Xilinx que ya estaba implementado. Este interfaz se encargaba de asignar valores a las señales que controlaban la memoria a partir de unas señales que entraban en él como activación de lectura, capacitación, etc. El diseño de la memoria así como su uso por medio del interfaz de Xilinx queda explicado en el capítulo de la memoria RAM.

Una vez comprendido el uso del interfaz, continuamos diseñando nuestro módulo encargado de ejecutar las órdenes introducidas por el usuario. El módulo hacía uso de la señal del tipo de orden proporcionada por el módulo que se encargaba de reconocerla, el banco de registros en el que se encuentran los datos de la orden y a partir de todo ello, genera las señales necesarias del interfaz de memoria para que se produzca una lectura y otras para almacenar el dato leído en un registro. Lo mismo ocurre en el caso de que el usuario desee hacer una escritura en memoria.

Tuvimos que modificar nuestro controlador general para que tras el reconocimiento de una orden concreta y de sus datos, pasara a ejecutar dicha orden utilizando el nuevo módulo creado.

Con todo esto nuestro prototipo era capaz de realizar la funcionalidad que nos habíamos propuesto en un principio. Generamos el mapa de bits y probamos a cargarlo en la FPGA. Como solía ocurrirnos, el prototipo no funcionó a la primera por lo que tuvimos que depurarlo hasta que conseguimos que hiciera lo que realmente tenía que hacer. Los problemas que tuvimos fueron principalmente en la parte que activa las señales del interfaz de memoria. Había que cumplir unos requisitos de tiempo de espera y de mantenimiento de las señales en las líneas de entrada y alguno de estos requisitos no lo cumplíamos del todo bien. Tras cambiar algunas de las señales y realizar una serie de pruebas más, al final conseguimos que el sistema fuese capaz de leer y escribir de memoria siguiendo las indicaciones que el usuario introdujese por teclado.

Parte 3

Ahora era el momento de seguir añadiéndole más funcionalidad a nuestro prototipo para conseguir que se aproximara cada vez más al diseño final.

Una vez realizada la parte de leer y escribir en memoria optamos por continuar nuestro desarrollo creando un planificador de tareas.

Este planificador de tareas sería un módulo con tres funcionalidades básicas:

- La primera es que cuando el usuario lo indicase, tomaría una tarea de memoria y la añadiría a una cola de ejecución antes de ser cargada en la FPGA.

- La segunda consiste en detectar si hay tareas en cola y en el caso de estar libre uno de los cuatro huecos disponibles en la FPGA para cargar tareas, tomar la tarea de la cola y cargarla en la FPGA para que comience su ejecución.

- La tercera permite liberar huecos de la FPGA ocupados por tareas que ya hubieran finalizado.

Diseñamos el planificador sobre papel, siguiendo la metodología de trabajo que de momento nos había ido bien, y después lo implementamos en VHDL.

El planificador no estaría completo del todo ya que dejamos en el aire la parte correspondiente a la carga de la tarea en el hueco de la FPGA y la de la liberación de las tareas finalizadas, ya esa parte requería poder realizar una reconfiguración parcial sobre la FPGA Virtex II que hasta ese momento no se había conseguido realizar. A pesar de esto el prototipo tenía implementada toda la parte correspondiente a la gestión de la cola de tareas y además existían una serie de señales que indicaban cuando una de las cuatro tareas que puede haber cargada sobre la FPGA había finalizado. Estas señales que en un futuro saldrían de las tareas y llegarían al planificador, ahora las tendríamos forzadas al valor que nosotros quisiésemos. En el caso de estar como que las tareas no han finalizado, el planificador cargaría las cuatro primeras tareas en orden de lanzamiento, actualizaría sus registros con la información de que los cuatro huecos están ocupados y

después, el resto de tareas lanzadas, se añadirían a la cola y no serían cargadas nunca ya que las señales de finalización permanecían fijas.

La parte correspondiente a la liberación del espacio dejado por una tarea finalizada también estaba implementada a pesar de no poder se usada. Se podía comprobar como si la señal que indica el fin de las tareas se encuentra siempre activa, se podrían lanzar todas las tareas que se quisieran sin límite ya que siempre habría un hueco libre al estar la señal de fin activada. El módulo encargado de la liberación lo que haría sería que nada más llegar una tarea, como se encontraba su línea de finalización activa, la eliminaría de los registros que almacenan la información de las tareas cargadas y permitiría así cargar nuevas tareas. Lo que realmente no hacía el módulo es borrar realmente la tarea de la FPGA ya que nunca fue cargada. De todas formas, esta parte no sería necesario implementarla ya que las tareas básicamente se sobrescribirían unas con otras.

Tras implementar toda la parte del planificador, el siguiente paso era comprobar que realmente funcionaba. Para ello lo añadimos sobre el prototipo anterior y lo integramos con el resto de módulos creando señales nuevas que hicieran funcionar al planificador cuando tuviera que hacerlo.

En primer lugar, para poder lanzar tareas lo que necesitábamos era una orden nueva que indicara al sistema de la nueva acción que tenía que realizar. La sintaxis de dicha orden es:

- *LANZAR IDENTIFICADOR_TAREA*

    - Donde *IDENTIFICADOR_TAREA* es un número de 8 bits representado por dos caracteres hexadecimales que sirve para distinguir las tareas que están almacenadas en memoria y cargar sólo aquella que desee el usuario.

Ya que habíamos de introducir una orden nueva, debíamos modificar nuestro módulo encargado de interpretar las órdenes para que soportara la nueva orden.

Para la parte correspondiente al almacenamiento de los datos de la orden, es decir, el identificador de la tarea, no fue necesario modificar el módulo lector de los datos de las órdenes ya que al tratarse de un dato hexadecimal podía ser almacenado en el banco de registros de la misma forma que el resto de los datos.

El último módulo que tuvimos que modificar fue el encargado de ejecutar las órdenes. Dicho módulo ya no sólo iba a permitir leer y cargar de memoria sino que tendría que ser capaz de informar al módulo nuevo del planificador del lanzamiento de una nueva tarea en el caso de que la orden introducida por el usuario fuese de ese tipo.

Una vez modificado todo esto, generamos el mapa de bits y tras cargarlo en la FPGA comprobamos para sorpresa nuestra que funcionó casi a la primera, salvo por algún fallo debido a despistes con el nombre de las señales no tuvimos mayor problema.

Parte 4

Terminado el planificador ya teníamos un sistema capaz de llevar tareas a petición del usuario desde la memoria a la FPGA. Nos encontrábamos ahora con un nuevo paso que debíamos de dar para ir avanzando hasta nuestro diseño final, este paso era pre-cargar las tareas en la memoria desde un dispositivo de almacenamiento no volátil. Esta pre-carga es necesaria ya que de alguna forma habría que llevar las tareas a la memoria SDRAM antes de que el usuario pueda decidir cuál quiere lanzar.

Para este punto nos planteamos varias alternativas. Una de estas era cargar las tareas desde un ordenador normal por medio de una conexión ethernet con la placa. Este método era viable pero no tenía sentido tener que depender de una conexión con un ordenador para poder ejecutar el sistema.

La segunda opción era utilizar la memoria compact flash incorporada en la placa. De esta forma podrían almacenarse las tareas dentro de una tarjeta de memoria y ser llevadas a la memoria en el arranque del sistema sin necesidad de ningún tipo de intervención por parte del usuario.

Decidimos implementar esta opción y comenzamos a documentarnos sobre la estructura y el funcionamiento del controlador del lector de tarjetas. **[R13]**

Realizamos un primer prototipo siguiendo la misma idea que en el caso de la memoria SDRAM, es decir, creamos un diseño que permitía al usuario leer y escribir bytes de la tarjeta compact flash. Para ello además tuvimos que realizar modificaciones sobre los módulos que se encargan de interpretar y ejecutar las órdenes para soportar las instrucciones que permiten al usuario utilizar la compact flash.

El prototipo no funcionó y seguimos investigando los posibles problemas que pudiéramos tener y las cosas que no habíamos tenido en cuenta.

Además modificamos la parte que se encargaba de la comunicación con la memoria SDRAM. Creamos un nuevo módulo que encapsulara el uso de la interfaz por medio de unas pocas señales y dejando de forma transparente la gestión de tiempos de espera. El resto de módulos que usaran este controlador únicamente deberían poner los valores deseados en las entradas y esperar la señal de fin proporcionada por el nuevo controlador.

En este punto del desarrollo se nos comunicó que la reconfiguración parcial de la FPGA no funcionaba correctamente sobre la Virtex II que estábamos utilizando, y que tendríamos que utilizar una nueva FPGA, la Virtex II Pro.

Parte 5

Este cambio de placa que en un principio no debía suponer un trabajo excesivo, supuso que el proyecto no pudiera llegar a completar toda la funcionalidad prevista inicialmente debido a nuevos problemas que encontramos en la nueva placa.

A partir de ahora, además de seguir investigando sobre el desarrollo del módulo controlador de la compact flash tuvimos que migrar todo el diseño que funcionaba correctamente en la Virtex II a la nueva Virtex II Pro.

La parte correspondiente al control de teclado no supuso ningún cambio importante. Por el contrario, el controlador de la VGA a pesar de que servía el mismo, nos dio problemas con la frecuencia de refresco. Este problema al principio no sabíamos realmente a que se debía y después llegamos a la conclusión de que teníamos que cambiar un parámetro de la configuración del controlador para soportar una nueva frecuencia de refresco.

Tuvimos que cambiar también un módulo *CLOCK_GEN* que nos proporcionaba una serie de relojes con distintas frecuencias a partir de los relojes de entrada generados por la placa. En este cambio no se produjo ningún tipo de problema.

Además de estos cambios, también tuvimos que cambiar la conexión de las salidas del sistema con los dispositivos ya que al cambiar de placa, la numeración de los pines de entrada/salida también cambió.

En el resto de módulos no tuvimos que hacer ninguna modificación casi en ninguno ya que el cambio de placa afectaba únicamente a los módulos controladores de dispositivos externos. En el único módulo que modificamos fue el controlador de memoria, lo dejamos de tal forma que permitía hacer uso de las operaciones de memoria aunque realmente ni los datos se escribían ni se leían de la memoria. Esto lo hicimos así ya que el módulo controlador de Xilinx tenía que cambiar principalmente porque la memoria que íbamos a utilizar ya no era SDRAM sino DDR.

Desde aquí hasta el final del proyecto el desarrollo iba a estar plagado de problemas que fueron ocasionados casi en su totalidad por el nuevo tipo de memoria RAM que teníamos que utilizar.

Parte 6

Dejamos a un lado el desarrollo del módulo controlador de compact flash y nos dedicamos principalmente a la implementación del nuevo controlador de memoria DDR, ya que si no éramos capaces de realizar operaciones de lectura y escritura sobre la memoria RAM de nada serviría hacer un módulo que leyera de una memoria externa si luego no se podía escribir en la memoria lo que leyésemos.

Debido a que mantuvimos la misma estructura que la del controlador que diseñamos para la SDRAM, no tuvimos que cambiar ningún tipo de señales de comunicación con el resto de módulos, únicamente las que salían de la FPGA hacia la memoria.

El funcionamiento de esta memoria DDR no tiene mucho que ver con el de la SDRAM. El proceso de lectura ya no consiste en establecer la dirección y esperar unos ciclos de latencia hasta obtener el dato, de la misma forma que en el caso de la escritura. Esta memoria al ser mucho más compleja requería diseñar un controlador mucho más complejo. Este controlador tenía que ser capaz de ajustarse a todas las restricciones temporales que tiene la memoria, fases de inicialización, periodos de autorrefresco y

otra serie detalles que hacen que si no se realizan correctamente, ajustándose a la especificación, la memoria no funcione.

Empezamos a leernos todo tipo de documentación sobre el funcionamiento de las memorias DDR, lo que se puede y lo que no se puede hacer, los pasos necesarios para realizar las operaciones permitidas, etc.

Si cuando estuvimos diseñando el controlador de compact flash ya nos pareció un trabajo largo y complicado, ni nos imaginábamos lo que iba a suponer diseñar un controlador para una memoria de estas características.

Parte 7

Motivados en primer lugar por la falta de tiempo, y en segundo por la gran dificultad que suponía desarrollar un interfaz completo que nos permitiera controlar la memoria DDR, nos decidimos por buscar un poco de ayuda y ver si encontrábamos algún diseño ya existente que pudiéramos utilizar como ya habíamos hecho otras veces.

Encontramos una aplicación proporcionada por Xilinx que permitía generar controladores de memoria DDR para muchos tipos de memoria, el MIG, explicado en el capitulo de herramientas software. Comenzamos a estudiar la posibilidad de utilizar dicho generador leyéndonos la documentación que incluía y vimos que podía ser la solución que estábamos buscando.

El funcionamiento de la aplicación es bastante sencillo, únicamente hay que especificarle el tipo de memoria y algunas de sus características como son su ancho de palabra, frecuencia de reloj, tipo de FPGA sobre la que se va a utilizar, bancos en los que se desea colocar dentro de la FPGA, etc. El programa genera una serie de archivos en VHDL que a simple vista habría que colocar en el proyecto y comenzar a utilizar. Además genera un archivo UCF con las restricciones de localización de los módulos creados para cumplir una serie de restricciones temporales exigidas por el diseño.

De esta forma generamos los archivos del controlador de la memoria que teníamos que utilizar en la nueva placa.

Comenzamos a adaptar nuestro diseño al nuevo controlador, para ello modificamos por completo el módulo que encapsulaba la interfaz de memoria. Las operaciones ya no consistían en poner en las líneas de control de la RAM los valores deseados y tras esperar un tiempo de latencia obtener el resultado, la complejidad del controlador exigía un diseño en el cual había que tener en cuenta exactamente una serie de ciclos de reloj, sincronizar señales con hasta cuatro relojes desfasados, etc.

Para la implementación de nuestro interfaz de memoria hicimos uso de la documentación que se incluía con la aplicación MIG. En ella se explicaba el comportamiento que se le tenían que dar a las señales de entrada del controlador para que éste pudiese generar las señales de control de la memoria correctamente en cada tipo de operación. Esta documentación además incluía una serie de simulaciones que nos permitieron una vez creado nuestro interfaz, comprobar si lo que nosotros habíamos generado coincidía en simulación con lo que venía especificado en la documentación.

El interfaz lo creamos siguiendo nuestra metodología de trabajo que seguía dándonos buenos resultados puesto que en simulación el controlador funcionaba correctamente. A pesar de esto, la tarea de implementarlo no fue sencilla, ya que el diseño se complicaba un poco al exigir sincronizar algunas señales con hasta cuatro relojes. Tras varias simulaciones con sus correspondientes correcciones de código conseguimos un diseño que cumplía fielmente la especificación.

Una vez que nuestro interfaz generaba correctamente las señales de control del módulo generado por el MIG nos dedicamos a incluir dicho módulo al diseño y conectarlo con nuestro interfaz. De toda la jerarquía de módulos que generaba el programa tomamos como raíz uno que tenía como nombre *ddr1_top*. Supusimos que este sería la entidad principal de la jerarquía ya que había otras que se utilizaban como banco de pruebas.

Aquí comenzaron nuevamente nuestros problemas. En primer lugar, las señales de salida hacia el módulo de memoria DDR que proporcionaba el controlador no coincidían con las que aparecían en la documentación de la Virtex II Pro. Algunas señales directamente no existían y otras eran réplicas de algunas que sólo existían una vez, como lo en el caso de la señal *clock enable* (*CKE*) y la de *chip select* (*CSB*) que en el diseño cada una era una única salida y en las señales de la especificación aparecían dos. Para estas señales optamos sencillamente por la solución de duplicar las que salían del controlador. Haciendo esto aparecieron nuevos problemas ya que el controlador, internamente, asignaba a las señales un buffer de salida por lo que si las queríamos duplicar después se producía un error. Para solucionar esto tuvimos que modificar el código del controlador para sacar la señal desde antes que se le asignara el buffer, añadiendo la dificultad de tener que adentrarnos en el código generado por el MIG, comprender un poco el uso de las señales y saber exactamente dónde teníamos que modificar.

También había una serie de señales que entraban en el módulo controlador pero que nosotros en nuestro diseño anterior, al no existir, nunca le dábamos un valor. Optamos por asignarles un valor fijo predeterminado. Estas señales eran el bus de selección de banco de la memoria que se utiliza, le asignamos un valor de 2. Otra es la señal de máscara de datos que la forzamos con un valor de cero para todos los bytes. Había una señal de espera, otra de retardo y algunas de reset que no estábamos seguros del valor que debían tomar así que les dimos el valor cero. De igual forma hicimos con otras señales que nunca se utilizaban.

Cuando ya teníamos asignadas todas las señales del controlador con las señales reales, añadimos al archivo UCF de nuestro proyecto las restricciones contenidas en el archivo UCF generado por el MIG e intentamos generar el mapa de bits. Descubrimos varias cosas, una es que no sintetizaba debido a numerosos errores entre los que destacaba uno que indicaba que la herramienta había encontrado durante el proceso una situación de síntesis que no había podido resolver. Este problema lo conseguimos solventar actualizando la versión del ISE por la 7.1. Otro problema fue que la traducción nos devolvía numerosos errores en las restricciones que habíamos añadido en el UCF. Para este último error lo que hicimos fue eliminar dichas restricciones, dejándolas para cuando el diseño al menos sintetizara.

El diseño finalmente sintetizó y conseguimos generar el mapa de bits pero al cargarlo en la FPGA nos dimos cuenta de que no funcionaba.

Estudiando un poco más la documentación comprobamos que algunas de las señales que habíamos fijado manualmente no tenían siempre el mismo valor sino que podían cambiar a lo largo del tiempo.

Comprobamos que habíamos tomado como entidad principal una que no era. Volvimos a generar el diseño con el MIG desactivando la opción de generar banco de pruebas y nos quedamos con la entidad principal *mem_interface_top*.

Cuando añadimos la nueva entidad principal al proyecto tuvimos que rehacer los pasos anteriores de añadir las señales que faltaban (cke y csb) y asignar un valor por defecto a algunas señales. Comprobamos que muchas de las señales que antes no sabíamos para que se utilizaban ya no eran entradas de la entidad sino que se generaban internamente.

Esto no nos quitó problemas. Teníamos los mismos problemas que antes pero que se resolvían de la misma manera. Además se producía un nuevo error debido a los relojes con los que alimentábamos el módulo controlador. En ningún sitio especificaba exactamente como debían ser los relojes por lo que le asignamos el reloj del sistema que al parecer no funcionaba correctamente.

El problema de los relojes nos hizo perder bastante tiempo, y nos daría problemas hasta el final, ya que como hemos dicho, no sabíamos bien que relojes son los que había que utilizar y además los errores que nos daba tampoco sabíamos muy bien a que es a lo que se referían. Tras muchos quebraderos de cabeza seguimos profundizando más en la comprensión del código generado por el MIG viendo por qué nos producía un error la señal de reloj. Una de las posibles causas era que el controlador internamente le añadía un buffer a la señal de reloj, por lo que si utilizábamos cualquiera de las que entraban a la FPGA, como también tenían asignado un buffer, se produciría un error. Pensamos en quitar el buffer y utilizar la señal que provenía del exterior directamente, al hacer esto así, el proceso de generación del mapa de bits nos devolvía nuevos errores, algunos en partes del diseño que si que sabíamos que funcionaban y otros como resultado de que la herramienta eliminaba parte del diseño durante el proceso de síntesis. Al eliminar parte del diseño algunas señales que salían hacia otros componentes de nuestro sistema también se eliminaban provocando nuevos errores y así sucesivamente. No veíamos una solución clara al problema.

Parte 8

No sabíamos por donde seguir, cada error que intentábamos solucionar cambiando algo provocaba la aparición de nuevos errores y no conseguíamos generar el mapa de bits del sistema para poderlo probar.

Optamos por una solución drástica pero que al final nos dio buen resultado y nos permitió seguir avanzando con el desarrollo de nuestro proyecto. Como la fuente de los errores ocasionados era el reloj, empezamos a estudiar por dónde se propagaba el reloj. Vimos que dentro del módulo *infrastructure_top* del controlador se encontraba un DCM que generaba las señales de reloj que se utilizaban en el resto del controlador. Las señales que se utilizaban eran cuatro relojes desfasados 90 grados cada uno a partir del reloj de entrada de 100 MHz. Nuestra solución fue que si no eramos capaces de

alimentar al DCM para que generase los relojes, podríamos obtener los relojes de otro sitio y alimentar a todo el controlador con ellos. Los relojes que decidimos utilizar fueron los que generaba el módulo *CLOCK_GEN*. También tuvimos que propagar la señal de *locked* que indica que los relojes proporcionados por el DCM están sincronizados. Eliminamos toda la parte de la generación de relojes del controlador y cambiamos todos los módulos que fueron necesarios para permitir la llegada de las señales de reloj y la señal de locked del exterior.

Para nuestra sorpresa, el sistema sintetizó y conseguimos generar el mapa de bits. Tras cargarlo en la FPGA nos quedamos aún más sorprendidos de que al ordenar al sistema que leyese de memoria el resultado que nos devolvía no eran todo ceros como solía suceder sino que era algo, posiblemente basura. Además nos quedamos aún más sorprendidos cuando realizamos una operación de carga, se completó y procedimos a leer lo que acabábamos de escribir y… funcionó, o al menos eso parecía.

Decimos que eso parecía ya que en una primera prueba al haber mantenido toda la estructura de la memoria anterior únicamente leíamos o escribíamos un byte. Cuando visualizábamos el resultado en pantalla sólo mostrábamos el byte leído por lo que ignorábamos que ocurría con los otros 31, ya que en cada ejecución podemos leer o escribir una ráfaga de cuatro palabras de 64 bits.

Cambiamos nuestra visualización de los datos para mostrar ahora las dos palabras de 128 bits que leíamos cada vez. Comprobamos que a pesar de que el sistema hacía lecturas y escrituras, los datos obtenidos tras la lectura no eran correctos en todos los casos, es decir que de los 256 bits, a veces alguno no se leía correctamente. Esta lectura incorrecta podría ser bien por que la propia lectura estuviera mal o porque el proceso de escritura alterara los datos.

Realizamos una serie de pruebas y comprobamos que era la lectura la que estaba mal ya que los datos que unas veces se leían mal, otras veces se leían bien por lo que no podrían estar mal almacenados en memoria. Además comprobamos que los datos que se leían de manera incorrecta no lo eran por completo sino que solamente alguno de sus bits se veía alterado en el proceso.

Continuamos con nuestro proceso de investigación para averiguar que era lo que hacía que no funcionase bien la lectura. Esta investigación nos llevaría al final del proyecto.

Entre las cosas que descubrimos fue que alguna parte de nuestra interfaz había fallos con algunas señales que no cambiaban cuando tenían que cambiar. Corrigiendo estos fallos conseguimos mejorar algo el diseño llegando hasta el punto de conseguir lecturas casi perfectas siempre pero que fallaban cada cierto tiempo. A partir del primer fallo las lecturas sucesivas ya no eran correctas.

Supusimos que estos fallos arbitrarios eran debidos a problemas de sincronización así que intentamos añadir las restricciones temporales y de localización generadas por el MIG a nuestro UCF. Con esto sorprendentemente no conseguimos mejorar nada incluso solía funcionar peor.

Optamos por revisar todo nuestro diseño, rehaciendo los módulos que habíamos hecho peor ya que al principio teníamos menos conocimientos que este instante. Probamos el nuevo diseño pero no mejoramos nada.

Pensamos también en que tal vez nuestra idea de eliminar el DCM que contenía el controlador no había sido acertada, aunque hay que reconocer que si no hubiésemos realizado tal paso tal vez no habríamos llegado tan lejos. Realizamos una nueva versión del prototipo pero dejando intacto el diseño original, salvo añadiendo las señales nuevas cke y csb. Con los conocimientos que habíamos adquirido desde que comenzamos en esta parte hasta el momento actual pudimos ser capaces de solucionar los problemas de alimentación del reloj. Le asignamos el reloj del sistema de 100 MHz pero en lugar de hacerlo directamente ya que hacía conflicto con el que entraba en el módulo *CLOCK_GEN*, le asignamos un buffer de entrada y la salida de dicho buffer se la pasamos al módulo controlador de memoria y al *CLOCK_GEN* que también tuvimos que modificar eliminando su buffer interno que le asignaba a la señal para que no produjera conflictos.

Con este nuevo prototipo pudimos generar el mapa de bits y tras probarlo comprobamos que los resultados que obteníamos eran los mismos que en el caso anterior. Probamos a añadir las restricciones y al igual que antes no mejoramos nada.

En un último esfuerzo por conseguir completar la parte del interfaz de memoria nos pusimos a comprobar la disposición de los módulos en la FPGA utilizando la herramienta FPGA Floorplanner incluida dentro del ISE de Xilinx. Comprobamos que había partes como eran los pines de entrada de datos que debían estar alineados con una parte del diseño. Retomando la documentación del MIG comprobamos que había una opción en la cual nos permitía establecer qué pines eran los que queríamos utilizar para cada tipo de señal, ya que por defecto la herramienta utilizaba unos cualquiera. De esta forma generamos un diseño en el cual las partes del diseño que tenían que estar alineadas con los pines de datos si que lo estaban. Al probar este nuevo diseño junto con sus restricciones que a simple vista parecían mejores que las anteriores comprobamos que funcionaba peor. Eliminamos las restricciones del UCF y el sistema funcionaba exactamente igual que el que teníamos antes de esta última supuesta mejora.

Durante una de las pruebas descubrimos que cuando se producía una lectura errónea, si reseteábamos la parte correspondiente al interfaz de memoria junto con el controlador del MIG, al volver a realizar la lectura, ésta volvía a ser correcta casi en la totalidad de los casos, alguna vez comprobamos que ni reseteando el componente la lectura se realizaba bien.

Parte 9

Para finalizar la explicación de nuestro desarrollo del proyecto vamos a comentar algunas metodologías de trabajo que hemos seguido y que hemos ido mejorando según íbamos avanzando.

En primer lugar hablaremos de cómo depurábamos el funcionamiento de nuestros módulos, principalmente de sus máquinas de estados y salidas. Al poder utilizar un monitor sobre el que podemos escribir cualquier cosa, a medida que fuimos desarrollando y encontrándonos con que lo que hacíamos no funcionaba del todo bien, o

que a veces simplemente se bloqueaba y no seguía la ejecución normal, optamos por mostrar por pantalla el estado de cada uno de los módulos. Así de esta forma conseguimos encontrar numerosos errores debidos principalmente a una mala interconexión de las señales o que les asignábamos valores que no eran los que tenían que tener, que realizaban ciclos de espera de más, etc.

También mostrábamos el valor de algunos datos cuyo contenido dudábamos que estuviese bien. Así mostramos al principio el valor de los datos que se guardaban en el banco de registros, el valor de algunos contadores, señales de control, incluso señales que llegaban desde el exterior y que no sabíamos muy bien su valor a priori.

Todo este tipo de pruebas eran las que realizábamos al principio ya que no sabíamos utilizar muy bien el simulador y tardábamos menos en mostrar la señal por pantalla o por un led y ver que podía ocurrir.

Cuando las cosas fueron tomando complejidad y el diseño tardaba más de diez minutos en generar el mapa de bits nos dimos cuenta que esta opción a parte de no ser efectiva era una pérdida de tiempo ya que para cada cambio de depuración que hacíamos teníamos que dedicar un gran tiempo a la generación del mapa de bits. Este tiempo a pesar de dejar este tipo de forma de depurar nos supuso muchos problemas ya que según avanzaba el proyecto y teníamos menos tiempo, más tiempo teníamos que dedicar a cada prueba y esto nos produjo perder mucho tiempo que podríamos haber dedicado a otras cosas o incluso podríamos haber avanzado mucho más si el proceso hubiera sido más rápido como al principio.

En el momento que dejamos de depurar mostrando estados y señales tuvimos que aprender a utilizar el simulador que durante una gran parte del proyecto no habíamos utilizado. Al final nos pareció bastante sencillo y útil y nos ayudó mucho en el desarrollo del interfaz de memoria para poder ver si las señales cumplían los tiempos que exigía el controlador.

Pero esto también tuvo su problema ya que a pesar de que la interfaz hacía lo que tenía que hacer, cuando la memoria no funcionaba no había forma de simular nada ya que se trataba de un componente externo hardware y no de un diseño en VHDL. Entonces en este punto del proyecto fue cuando hicimos uso del analizador digital de señales. Este analizador nos permitía comprobar en tiempo de ejecución el valor de distintas señales de una manera rápida y fácil, únicamente teníamos que sacar a unos pines de salida la señal que quisiésemos visualizar y mediante un cable conectábamos dicho pin al analizador y podíamos ver que sucedía. Además el analizador al ofrecer una gran variedad de opciones nos permitía ir directamente al punto exacto que queríamos analizar ignorando el resto.

A pesar de todas esta técnicas de depuración al final no conseguimos solucionar los problemas que teníamos con la memoria DDR ya que no sabíamos cual era la causa y tras darle muchas vueltas no sabíamos que mirar.

Lo último que hicimos para depurar el módulo de memoria es crear un componente aislado que únicamente leía de memoria y otro para escribir. Estos componentes los hicimos porque pensábamos que los problemas podían tener la causa en algún conflicto con el resto de módulos. El módulo que escribía en memoria tenía como control un

switch que cuando se accionaba escribía una serie de bytes en memoria y al finalizar se encendía un led. El módulo lector leía una serie de bytes de memoria, que generalmente eran escritos previamente por el módulo cargador, y después los visualizaba en la pantalla. Comprobamos con estas pruebas que no dependía el problema de la interconexión con el resto del sistema por lo que ya no supimos que más mirar. Al conectar estos módulos al analizador comprobamos que las señales de reloj no estaban demasiado bien sincronizadas y que la señal de strobe de los datos de memoria no parecía funcionar bien pero el proyecto llegó a su fin y no tuvimos más tiempo de seguir probando cosas.

Final

Al final conseguimos tener un sistema para la Virtex II que permitía leer y escribir de la memoria, lanzar tareas de manera virtual y comprobar como se gestionaba la cola de tareas a falta de ser cargadas en la FPGA.

En la Virtex II Pro conseguimos tener un prototipo que realizaba lecturas y escrituras en memoria pero que no eran correctas en todos los casos.

Capítulo 5: Arquitectura Final

La estructura general comentada en el apartado del objetivo del proyecto sufrió una serie de cambios pasando de ser el diseño inicial ideal del proyecto hasta convertirse en el diseño final.

La visión definitiva de nuestro sistema correspondería con el siguiente diagrama:

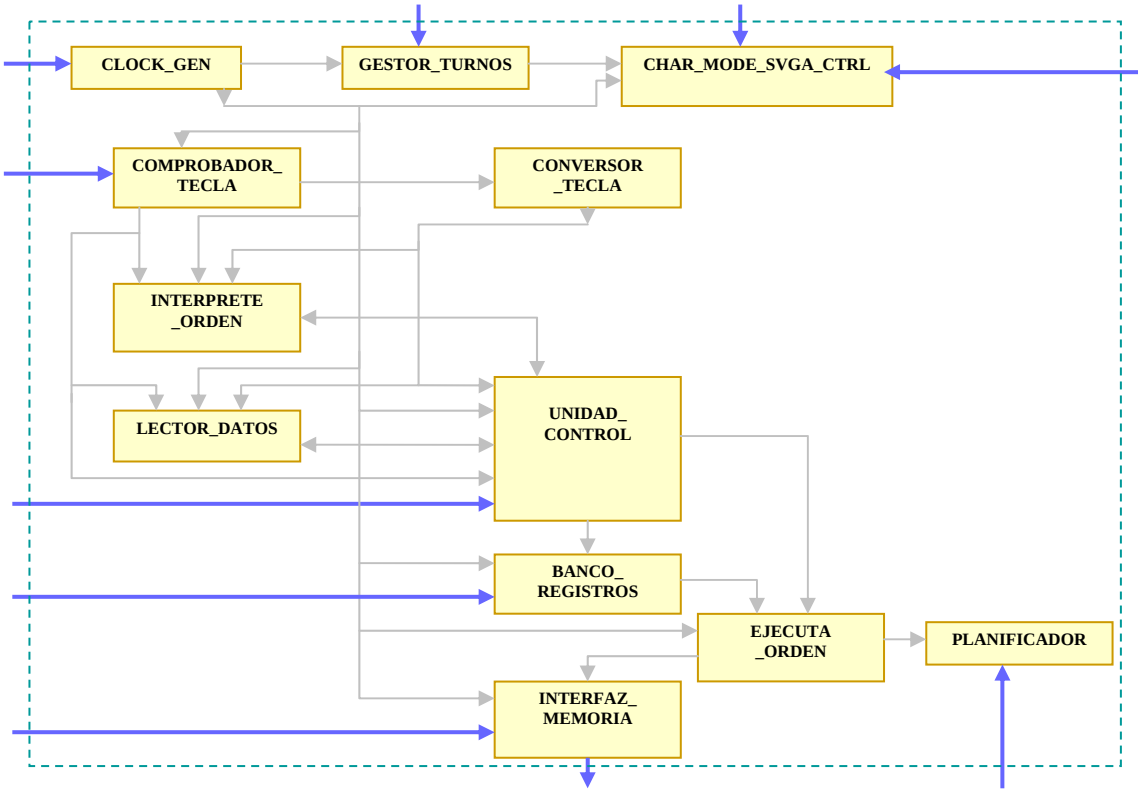


Figura 64: Esquema general de nuestro sistema

Como se puede ver hay una serie de módulos que se pueden asociar con algunos del diagrama original y también hay algunos que aparecían en el original pero que no aparecen en éste debido a que finalmente no llegaron a implementarse.

Todos estos módulos están interconectado en una entidad principal llamada *ordenes*. Dicha interfaz es la superior de toda la jerarquía del sistema y contiene dentro de ella a todos los otros módulos.

Además es la entidad que tiene salidas hacia el exterior por lo que en su descripción tiene todas aquellas señales correspondientes con componentes externos a la FPGA: memoria, teclado, pantalla, etc.

Un esquema general del componente principal del sistema es el siguiente:

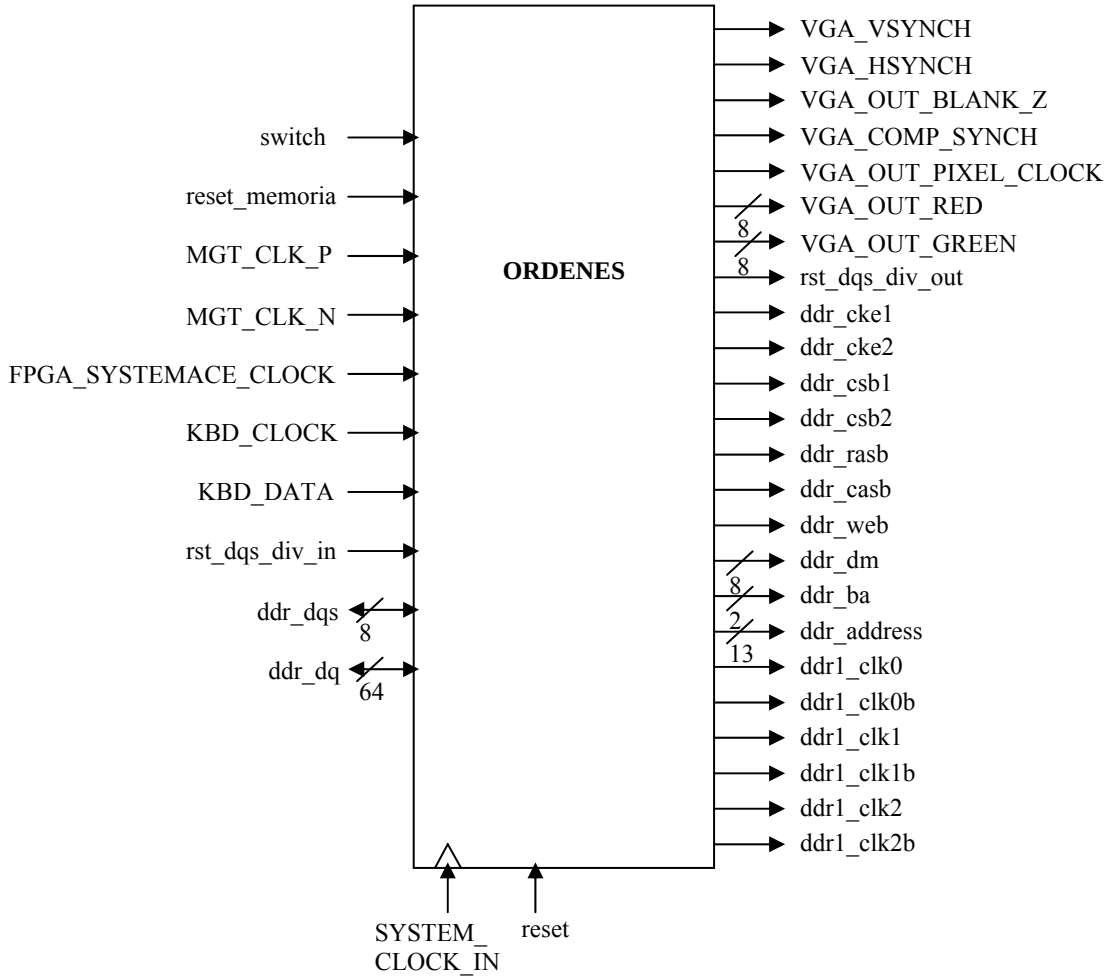


Figura 65: Componente principal del sistema

La lista de entradas y salidas del componente junto con su significado es la siguiente:

Tabla 10: Entradas del componente principal

Entradas	Descripción
reset	- Señal de reset del driver de teclado y de los DCMs del módulo generador de señales de reloj. Corresponde con el switch 2 de la placa
reset_memoria	- Señal de reset del controlador de memoria. Corresponde con el switch 1 de la placa
switch	- Señal de reset de cada uno de nuestros componentes. Corresponde con el switch 4 de la placa
MGT_CLK_P	- Señal de reloj generada por la FPGA de 75MHz. Esta y la siguiente son señales LVDS diferenciales
MGT_CLK_N	- Señal de reloj generada por la FPGA de

	75MHz. Esta y la anterior son señales LVDS diferenciales
SYSTEM_CLOCK_IN	- Señal de reloj generada por la FPGA de 100 MHz
FPGA_SYSTEMACE_CLOCK	- Señal de reloj generada por la FPGA de 32 MHz
KBD_CLOCK	- Señal del reloj que viene del teclado
KBD_DATA	- Señal de datos que viene del teclado
rst_dqs_div_in	- Señal de realimentación del reset del DQS

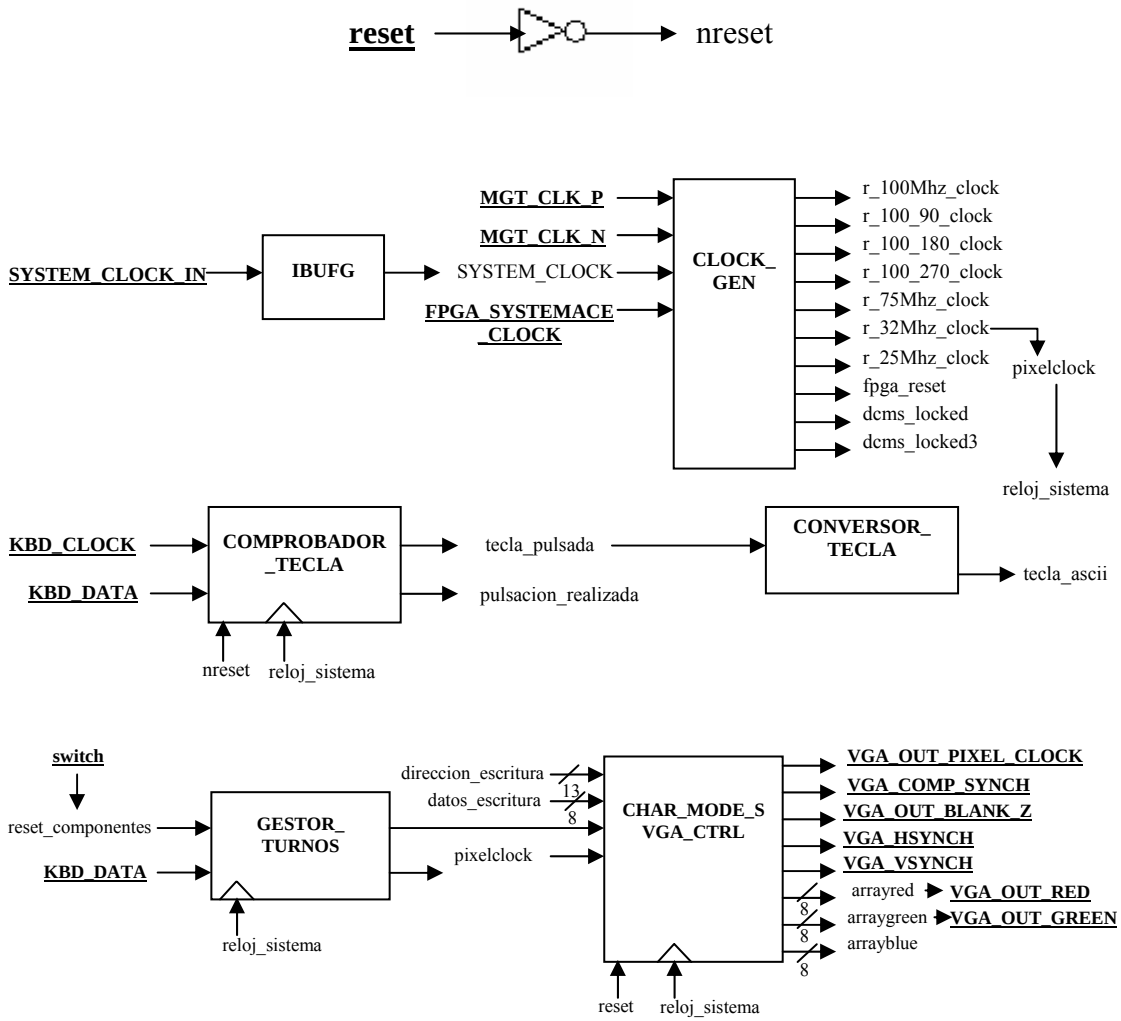
Tabla 11: Salidas del componente principal

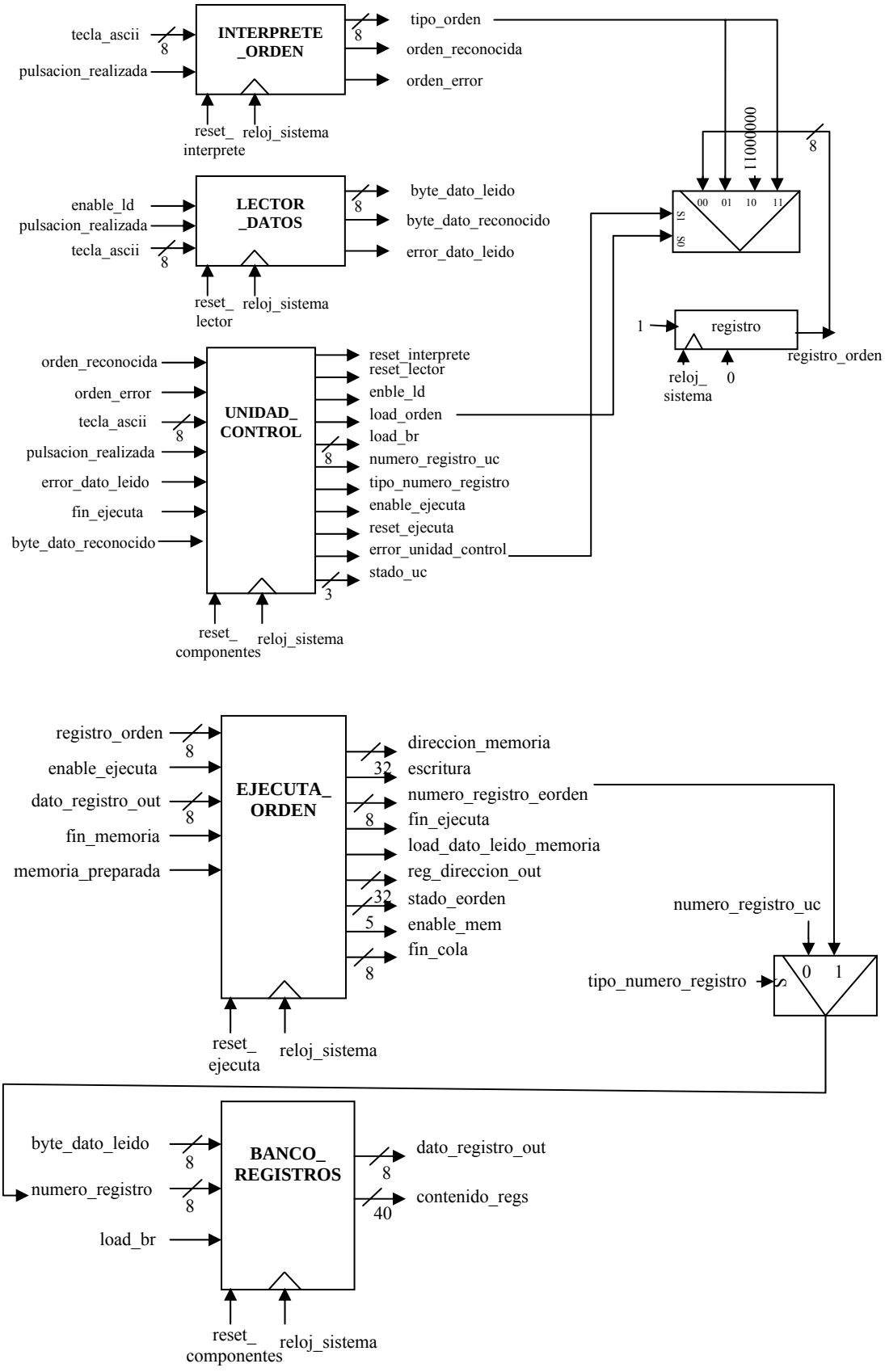
Salidas	Descripción
VGA_OUT_PIXEL_CLOCK	- Señal de reloj de refresco del monitor
VGA_COMP_SYNCH	- Señal de comp_synch
VGA_HSYNCH	- Señal de hsynch
VGA_VSYNCH	- Señal de vsynch
VGA_OUT_BLANK_Z	- Señal de blanking
VGA_OUT_RED [7:0]	- Señal de intensidad de color rojo
VGA_OUT_GREEN [7:0]	- Señal de intensidad de color verde
rst_dqs_div_out	- Señal de realimentación del reset del DQS
ddr_cke1	- Señal de clock enable
ddr_cke2	- Señal de clock enable
ddr_csb1	- Señal de chip select
ddr_csb2	- Señal de chip select
ddr_rasb	- Señal de comando
ddr_casb	- Señal de comando
ddr_web	- Señal de comando
ddr_dm [7:0]	- Señal de máscara de datos
ddr_ba [1:0]	- Señal de dirección de banco
ddr_address [22:0]	- Señal de dirección
ddr_clk0	- Señal de reloj
ddr_clk0b	- Señal de reloj
ddr_clk1	- Señal de reloj
ddr_clk1b	- Señal de reloj
ddr_clk2	- Señal de reloj
ddr_clk2b	- Señal de reloj

Tabla 12: Señales de entrada/salida del componente principal

Entrada/Salida	Descripción
ddr_dqs [7:0]	- Señal de strobe de datos
ddr_dq [63:0]	- Señal de bus de datos

Al comienzo del capítulo se dio un esquema general de las conexiones de los módulos por los que estaba formado el sistema. Pues bien, ahora vamos a ver un poco más en detalle las entradas y salidas de cada uno de estos componentes y su interconexión con cada uno del resto de los módulo. Las señales vienen representadas con una etiqueta representativa ya que nos resultaba imposible mostrar el conjunto de todos los módulos unidos mediante cables, a parte de que este tipo de visualización también es menos clara. Las señales que aparecen subrayadas y marcadas en negrita corresponden con señales que son de entrada/salida del componente principal. Se puede observar como la señal de intensidad de color azul no sale del componente principal, esto es debido a que hay una compartición de los pines de salida por parte del controlador de memoria y del de VGA. Como la memoria no puede perder ninguna señal y a nosotros no nos influía demasiado que no se pudiese ver el color azul entonces eliminamos esta señal.




INTERPRETE _ORDEN
tecla_ascii
8
pulsacion_realizada
reset_ interprete
reloj_sistema
tipo_orden
orden_reconocida
orden_error
LECTOR _DATOS
enable_ld
pulsacion_realizada
tecla_ascii
reset_ lector
reloj_sistema
byte_dato_leido
byte_dato_reconocido
error_dato_leido
UNIDAD_ CONTROL
orden_reconocida
orden_error
tecla_ascii
pulsacion_realizada
error_dato_leido
fin_ejecuta
byte_dato_reconocido
reset_ componentes
reloj_sistema
reset_interprete
reset_lector
enble_ld
load_orden
load_br
numero_registro_uc
tipo_numero_registro
enable_ejecuta
reset_ejecuta
error_unidad_control
stado_uc
3
11000000
00 01 10 11
registro
reloj_ sistema
registro_orden
EJECUTA_ ORDEN
registro_orden
enable_ejecuta
dato_registro_out
fin_memoria
memoria_preparada
reset_ ejecuta
reloj_sistema
direccion_memoria
escritura
numero_registro_eorden
fin_ejecuta
load_dato_leido_memoria
reg_direccion_out
stado_eorden
enable_mem
fin_cola
numero_registro_uc
tipo_numero_registro
BANCO_ REGISTROS
byte_dato_leido
numero_registro
load_br
reset_ componentes
reloj_sistema
dato_registro_out
contenido_regs
40

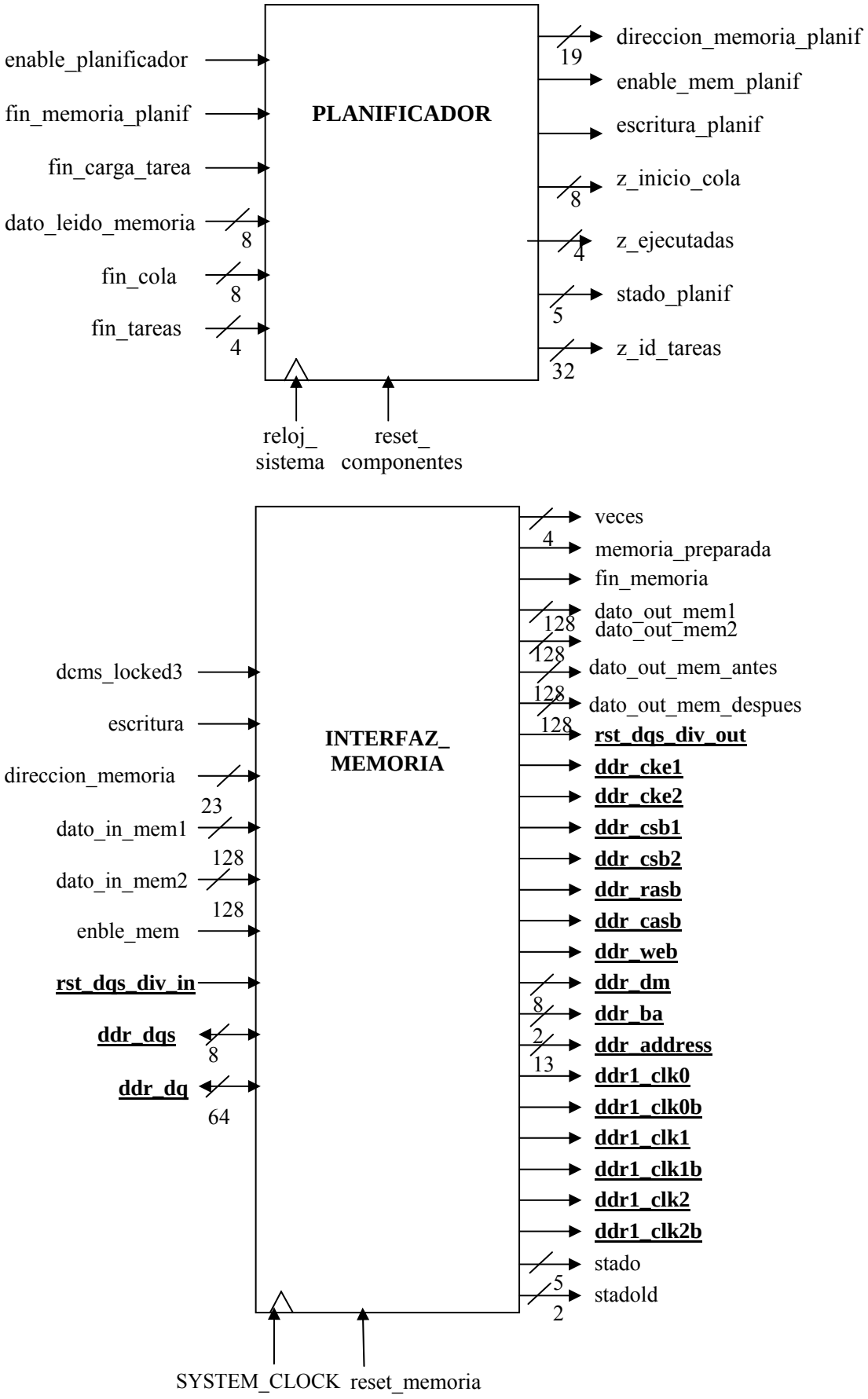


Figura 66: Ruta de datos del sistema

A continuación vamos a entrar en los detalles de implementación de cada uno de los componentes del sistema para comprender mejor su funcionamiento. Para ello agruparemos los módulos que realicen funciones relacionadas en una categoría común. Algunos de ellos debido a su complejidad formarán un apartado independiente.

5.1. Kernel del Sistema Operativo

El kernel o controlador principal del sistema es el encargado de realizar toda la parte de la gestión de los componentes. Controla los datos introducidos por el usuario, los interpreta y se encarga de visualizarlos. Además tiene la misión de activar las señales que sean necesarias para poder llevar a cabo la ejecución de las órdenes así como de interconectar y gestionar la comunicación entre los distintos componentes, para que se cumpla correctamente el flujo de tareas preestablecidas y todo pueda funcionar sin que haya situaciones de error o ambigüedad.

Una vez leídos los datos utilizando los drivers de entrada/salida es necesario interpretarlos en función de su tipo ya que pueden corresponder tanto al nombre de una orden como a sus argumentos de entrada. El proceso de reconocimiento es distinto para cada uno de ellos y hay un módulo independiente encargado de realizar la labor de reconocer órdenes (*interprete_orden*) y otro para los argumentos (*lector_datos*).

Cuando se conocen todos los datos de la operación que hay que realizar entonces toma el control otro módulo que es el que se encarga de ejecutarla (*ejecuta_orden*).

Por último y para hacer que todo funcione correctamente y la comunicación de los módulos sea efectiva, existe un módulo que gestiona cuándo debe actuar cada módulo y cuando tiene que dejar de hacerlo. Este módulo es la *unidad_control* y es la que se encarga de controlar la realización de todas las fases del proceso desde que se introducen los datos hasta que se finaliza la ejecución de la orden.

Ahora vamos a explicar un poco más en profundidad como funcionan cada uno de estos módulos.

5.1.1. Unidad de control

Descripción

Con este módulo simulamos la secuencia de flujo de la ejecución de una orden, desde que el usuario la introduce hasta que finaliza su ejecución.

En cada fase del proceso la unidad de control es la encargada de activar los módulos que realizan las tareas concretas de cada fase estableciendo los valores necesarios de sus señales de entrada para su correcto funcionamiento.

Esquema

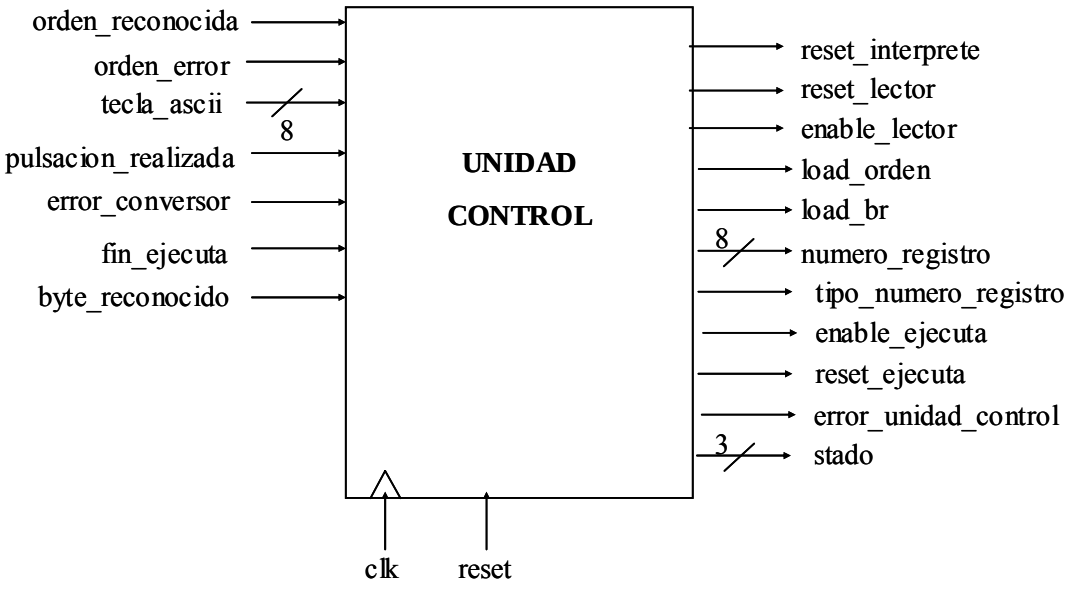


Figura 67: Unidad de control

Entradas/Salidas

Tabla 13: Entradas de la unidad de control

Entradas	Descripción
reset	- Señal de reset asíncrono del sistema
clk	- Señal de reloj
orden_reconocida	- Señal que indica si se ha reconocido una orden
orden_error	- Señal que indica si se ha producido un error en al reconocer una orden
tecla_ascii [7:0]	- Señal que contiene el codigo ASCII de la tecla pulsada
pulsacion_realizada	- Señal que se activa cada vez que se pulsa una tecla
error_conversor	- Señal que indica si se ha producido un error al leer los datos de la orden
fin_ejecuta	- Señal que indica si ha finalizado la ejecución de la orden actual
byte_reconocido	- Señal que indica si se ha reconocido un nuevo byte de los datos de la orden

Tabla 14: Salidas de la unidad de control

Salidas	Descripción
reset_interprete	- Señal que resetea el módulo *interprete_orden*
reset_lector	- Señal que resetea el módulo *lector_datos*
Enable_lector	- Señal que capacita el módulo *lector datos*
load_orden	- Señal que activa la señal de carga de un registro que almacena la orden actual reconocida
load_br	- Señal que indica al banco de registros cuando debe cargar el dato que tiene sobre el bus
numero_registro [7:0]	- Señal que indica el número del registro que utiliza la unidad de control
tipo_numero_registro	- Señal que indica al multiplexor de selección de la señal número registro si tiene que utilizar la que procede del módulo *ejecuta_orden* o bien la de la *unidad_control*
enable_ejecuta	- Señal que capacita el módulo *ejecuta_orden*
reset_ejecuta	- Señal que resetea el módulo *ejecuta_orden*
error_unidad_control	- Señal que indica que se ha producido un error en *unidad_control*

Unidad de control

La máquina de estados correspondiente con el controlador del módulo es la siguiente:

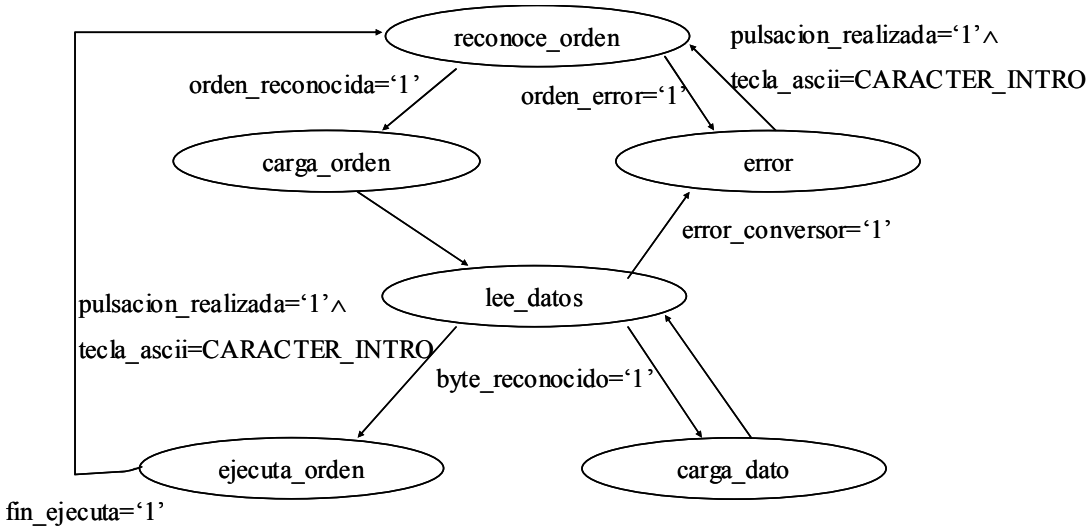


Figura 68: Diagrama de estados de la unidad de control

Como se puede observar el funcionamiento es de lo más sencillo ya que únicamente hay un estado por cada una de las fases por la que tiene que pasar cualquier orden.

Todo parte de un estado inicial de espera (*reconoce_orden*) en el cual se permanece mientras el usuario está introduciendo el nombre de la orden que desea ejecutar. En este estado se capacita el módulo intérprete de órdenes cuyo resultado se almacena para su posterior uso.

Si se ha producido un error se pasará a un estado de bloqueo del cual se saldrá con una pulsación de la tecla intro.

Sino entonces se pasa al estado encargado de leer los datos propios de cada orden *lee_datos*. Estos datos siempre van a ser bytes representados por números hexadecimales de dos dígitos y cada vez que el módulo encargado de leer los datos reconozca un nuevo byte válido se producirá una transición al estado *carga_dato* en el cual se almacena el nuevo dato en el banco de registros. Si por el contrario se hubiera producido un error durante el reconocimiento del dato actual entonces el sistema pasará al estado de bloqueo de error al igual que en el caso anterior.

El ciclo leer-dato y almacenar-dato se realiza tantas veces como datos introduzca el usuario por teclado. Una vez que el usuario finalice pulsando la tecla intro el sistema pasará a ejecutar la nueva orden introducida. Para ello se realiza una transición al estado *ejecuta_orden*.

En este estado se permanece hasta que el módulo *ejecuta_orden* no active su señal de *fin_ejecuta*. Una vez ocurrido esto la unidad de control regresará al estado inicial para reconocer una nueva orden.

A partir de la máquina de estados obtenemos las siguientes tablas de transición de estados y de salidas:

Transición de estados

Tabla 15: Transición de estados de la unidad de control

Estado	orden_reconocida	orden_error	error_conversor	byte_reconocido	pulsacion_realizada	fin_ejecuta	tecla_ascii	Estado Siguiente
reconoce_orden	1	-	-	-	-	-	-	carga_orden
reconoce_orden	-	1	-	-	-	-	-	error
carga_orden	-	-	-	-	-	-	-	lee_datos
lee_datos	-	-	1	-	-	-	-	error
lee_datos	-	-	-	1	-	-	-	carga_dato
lee_datos	-	-	-	-	1	-	-	ejecuta_orden
carga_dato	-	-	-	-	-	-	-	lee_datos
ejecuta_orden	-	-	-	-	-	1	-	reconoce_orden
error	-	-	-	-	1	-	CARACTER_INTRO	reconoce_orden

Salidas

Tabla 16: Valores de las salidas de la unidad de control

Estado	pulsacion_realizada	tecla_ascii	reset_interprete	reset_lector	load_orden	load_br	enable_ejecuta	tipo_numero_registro	enable_lector	error_unidad_control	numero_registro	reset_ejecuta	stado
reconoce_orden	-	-	0	1	0	0	0	0	0	0	-	0	"000"
carga_orden	-	-	0	0	1	0	0	0	1	0	"00000000"	0	"111"
lee_datos	1	CARACTER _INTRO	0	0	0	0	1	1	0	0	-	0	"001"
lee_datos	≠1	≠CARACTER _INTRO	0	0	0	0	0	0	0	0	-	1	"001"
carga_dato	-	-	0	0	0	1	0	0	0	0	numero_ registro+'1'	0	"111"
ejecuta_orden	-	-	1	0	0	0	0	1	0	0	-	0	"010"
error	-	-	1	0	0	0	0	0	0	1	-	0	"111"

Simulaciones

Ahora vamos a realizar algunas simulaciones para mostrar el funcionamiento del módulo.

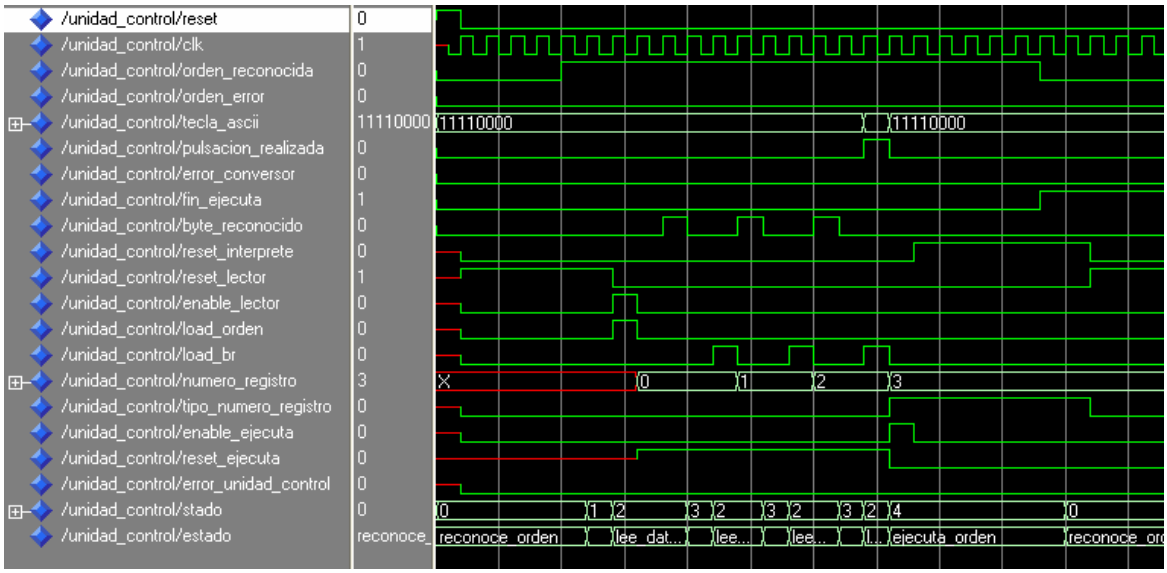


Figura 69: Simulación de la unidad de control

En la simulación se puede ver en la parte inferior como el sistema va cambiando de estado a medida que se van activando las señales de entrada correspondiente que indican el final de una fase. También se puede ver como se generan las señales de salida para controlar los distintos módulos.

5.1.2. Intérprete de órdenes

Descripción

Este módulo es el encargado de reconocer una secuencia de caracteres ASCII como una orden completa. Los caracteres son proporcionados desde el exterior a cada ciclo de reloj y solo se tendrán en cuenta en aquellos ciclos en los que esté activa la señal de *pulsacion_realizada*.

En el caso de que se reconozca una orden completa el módulo se queda en un estado de aceptación activando la señal correspondiente. Además se establece en el bus de la orden reconocida el valor codificado de la orden.

Si se produjese algún error durante la detección de la orden porque la entrada sea una orden no válida entonces el módulo finalizaría su ejecución quedando en un estado de error. La señal que se activa de salida es la de error y el valor que haya en el bus de orden codificada no será válido.

Cada vez que se desee reconocer una orden nueva deberá resetearse el módulo ya que siempre acaba en un estado de bloqueo bien sea por haber detectado una orden válida o por que se haya producido un error.

Esquema

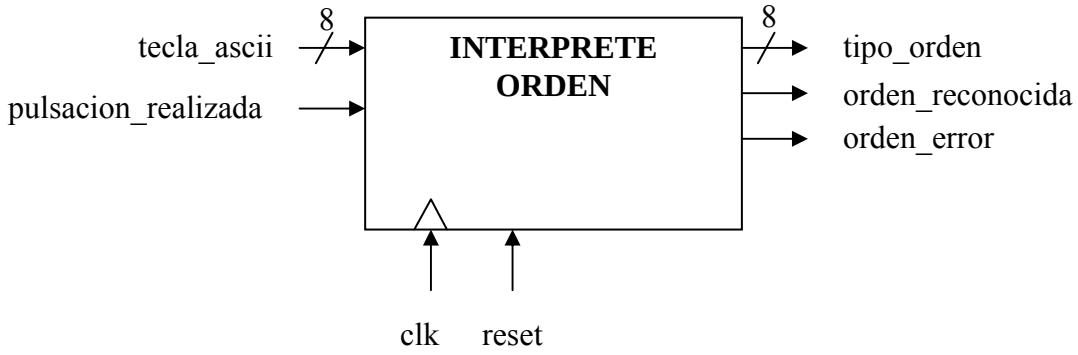


Figura 70: Intérprete de órdenes

Entradas/Salidas

Tabla 17: Entradas del interprete de órdenes

Entradas	Descripción
tecla_ascii [7:0]	- Contiene el valor ASCII del carácter que hay que reconocer en el ciclo actual
reset	- Señal de reset asíncrono del sistema
pulsacion_realizada	- Señal que indica que la tecla que hay en la entrada *tecla_ascii* es válida en el ciclo actual
clk	- Señal de reloj

Tabla 18: Salidas del interprete de órdenes

Salidas	Descripción
tipo_orden [7:0]	- Señal que indica la orden que se ha reconocido - Únicamente será válida si la salida *orden_reconocida* está activada
orden_reconocida	- Señal que indica si la orden que se ha introducido es válida
orden_error	- Señal que indica que la orden que se ha introducido no es válida

Ruta de datos

La representación esquemática de la ruta de datos de este módulo es la siguiente:

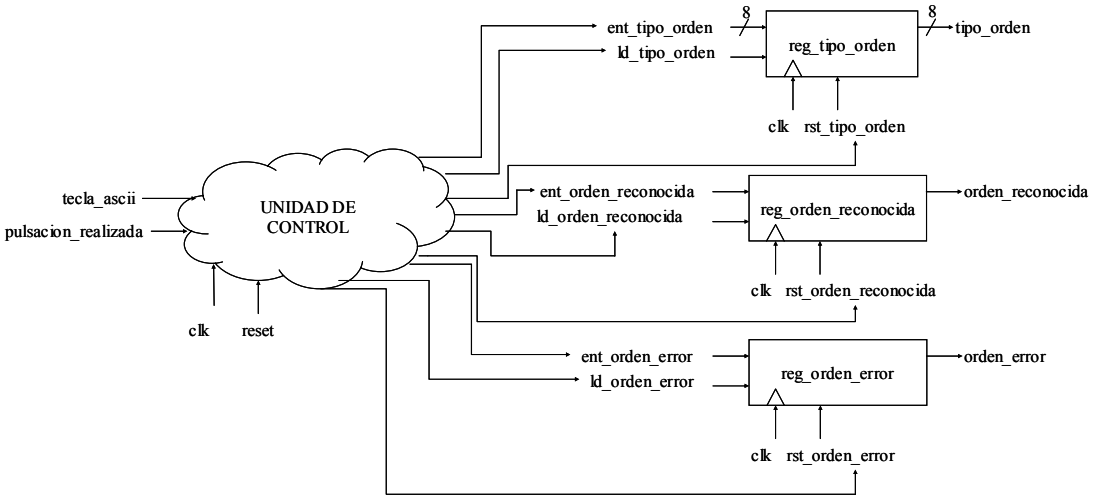


Figura 71: Ruta de datos del intérprete de órdenes

Como se puede observar el módulo está compuesto principalmente por tres registros que almacenan el resultado de la evaluación de la orden. Esto es así para que el resultado sea accesible durante el resto del proceso de ejecución a pesar de haber finalizado.

Unidad de control

La máquina de estados correspondiente con el controlador del módulo es la siguiente:

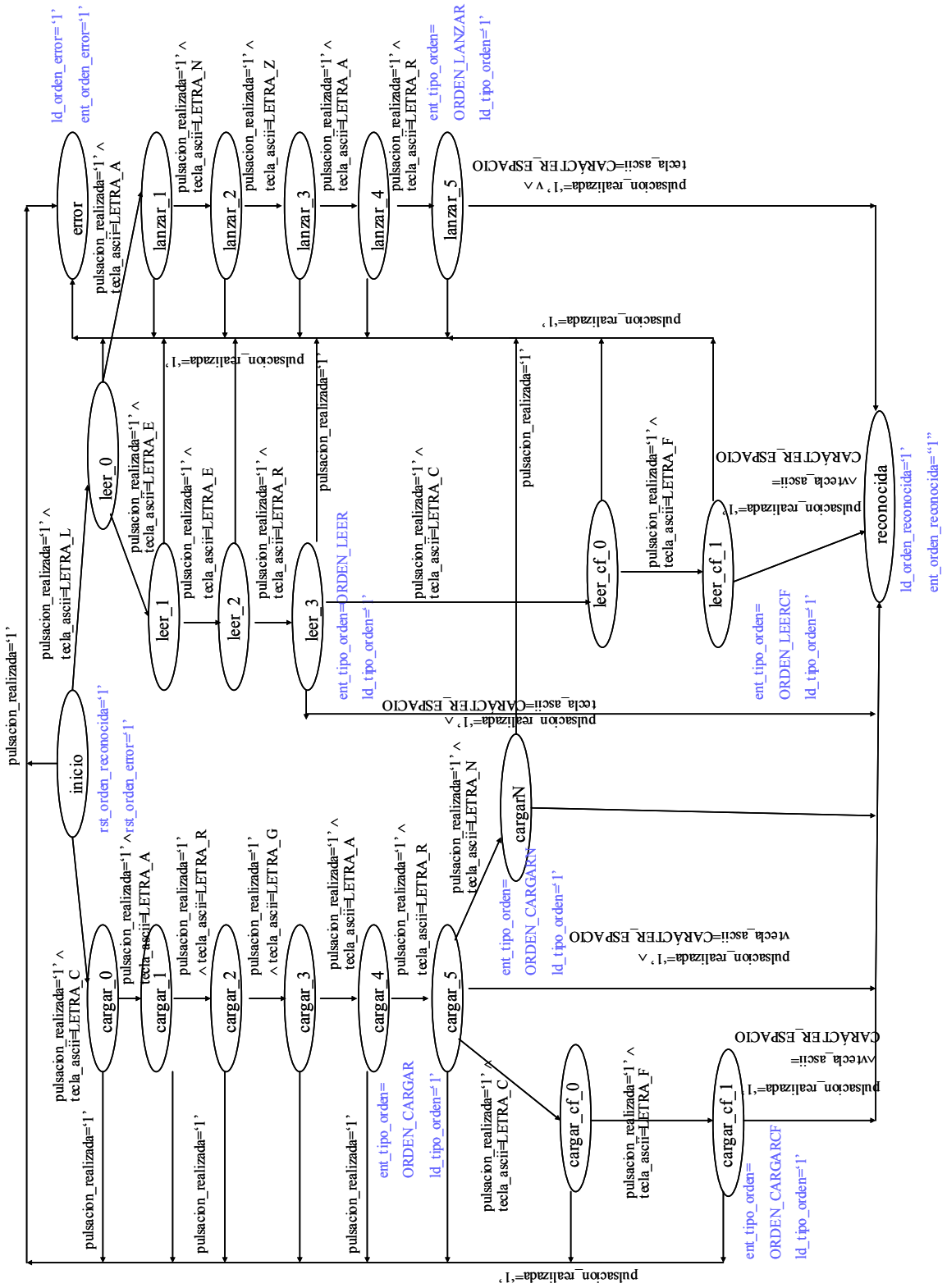


Figura 72: Diagrama de transición de estados del intérprete de órdenes

Como se puede observar el control principal del módulo basa su funcionamiento en un autómata finito que va reconociendo los caracteres de la orden uno tras otro. En el caso de que tras el último se pulse espacio y lo anterior corresponda con una órden válida entonces se pasa a un estado de aceptación y se activa la salida de *orden_reconocida* además de establecer el valor de la orden en el bus de *tipo_orden*. Si no se hubiese reconocido una orden válida o si en cualquiera de los estados correspondientes a una orden se introdujera un carácter no esperado como siguiente de la cadena, el autómata pasaría a un estado de error activando la señal de *orden_error* que invalida el posible valor que hubiera en el bus de *tipo_orden*.

A partir de la máquina de estados obtenemos las siguientes tablas de transición de estados y de salidas:

Transición de estados

Tabla 19: Transición de estados del intérprete de órdenes

Estado	pulsacion _realizada	tecla_ascii	Estado siguiente
inicio	1	LETRA_C	cargar_0
inicio	1	LETRA_L	leer_0
inicio	1	≠LETRA_C ∧ ≠LETRA_L	error
cargar_0	1	LETRA_A	cargar_1
cargar_0	1	≠LETRA_A	error
cargar_1	1	LETRA_R	cargar_2
cargar_1	1	≠LETRA_R	error
cargar_2	1	LETRA_G	cargar_3
cargar_2	1	≠LETRA_G	error
cargar_3	1	LETRA_A	cargar_4
cargar_3	1	≠LETRA_A	error
cargar_4	1	LETRA_R	cargar_5
cargar_4	1	≠LETRA_R	error
cargar_5	1	LETRA_N	cargar_N
cargar_5	1	CARACTER _ESPACIO	reconocida
cargar_5	1	LETRA_N	cargarN
cargar_5	1	LETRA_C	cargar_cf_0
cargar_5	1	≠LETRA_N ∧ ≠CARÁCTER _ESPACIO ∧≠LETRA_C	error
cargar_N	1	CARACTER _ESPACIO	reconocida
cargar_N	1	≠CARACTER _ESPACIO	error
leer_0	1	LETRA_E	leer_1
leer_0	1	LETRA_A	lanzar_1
leer_0	1	≠LETRA_E∧ ≠LETRA_A	error
leer_1	1	LETRA_E	leer_2
leer_1	1	≠LETRA_E	error
leer_2	1	LETRA_R	leer_3
leer_2	1	≠LETRA_R	error
leer_3	1	CARACTER _ESPACIO	reconocida
leer_3	1	LETRA_C	leer_cf_0
leer_3	1	≠CARACTER _ESPACIO ∧≠LETRA_C	error
lanzar_1	1	LETRA_N	lanzar_2
lanzar_1	1	≠LETRA_N	error
lanzar_2	1	LETRA_Z	lanzar_3

lanzar_2	1	≠LETRA_Z	error
lanzar_3	1	LETRA_A	lanzar_4
lanzar_3	1	≠LETRA_A	error
lanzar_4	1	LETRA_R	lanzar_5
lanzar_4	1	≠LETRA_R	error
lanzar_5	1	CARACTER_ESPACIO	reconocida
lanzar_5	1	≠CARACTER_ESPACIO	error
leer_cf_0	1	LETRA_F	leer_cf_1
leer_cf_0	1	≠LETRA_F	error
leer_cf_1	1	CARACTER_ESPACIO	reconocida
leer_cf_1	1	≠CARACTER_ESPACIO	error
cargar_cf_0	1	LETRA_F	cargar_cf_1
cargar_cf_0	1	≠LETRA_F	error
cargar_cf_1	1	CARACTER_ESPACIO	reconocida
cargar_cf_1	1	≠CARACTER_ESPACIO	error
reconocida	-	-	reconocida
error	-	-	error

Salidas

Tabla 20: Valores de las salidas del interprete de órdenes

ESTADO	**rst_tipo_orden**	**rst_orden_reconocida**	**rst_orden_error**	**ld_tipo_orden**	**ld_orden_reconocida**	**ld_orden_error**	**ent_tipo_orden**	**ent_orden_reconocida**	**ent_orden_error**
inicio	0	1	1	0	0	0	-	-	-
cargar_5	0	0	0	1	0	0	ORDEN_CARGAR	-	-
cargar_N	0	0	0	1	0	0	ORDEN_CARGARN	-	-
leer_3	0	0	0	1	0	0	ORDEN_LEER	-	-
lanzar_5	0	0	0	1	0	0	ORDEN_LANZAR	-	-
leer_cf_1	0	0	0	1	0	0	ORDEN_LEERCF	-	-
cargar_cf_1	0	0	0	1	0	0	ORDEN_CARGARCF	-	-
error	0	0	0	0	0	1	-	-	“1”
reconocida	0	0	0	0	1	-	-	“1”	-

Simulaciones

Ahora vamos a realizar algunas simulaciones para mostrar el funcionamiento del módulo.

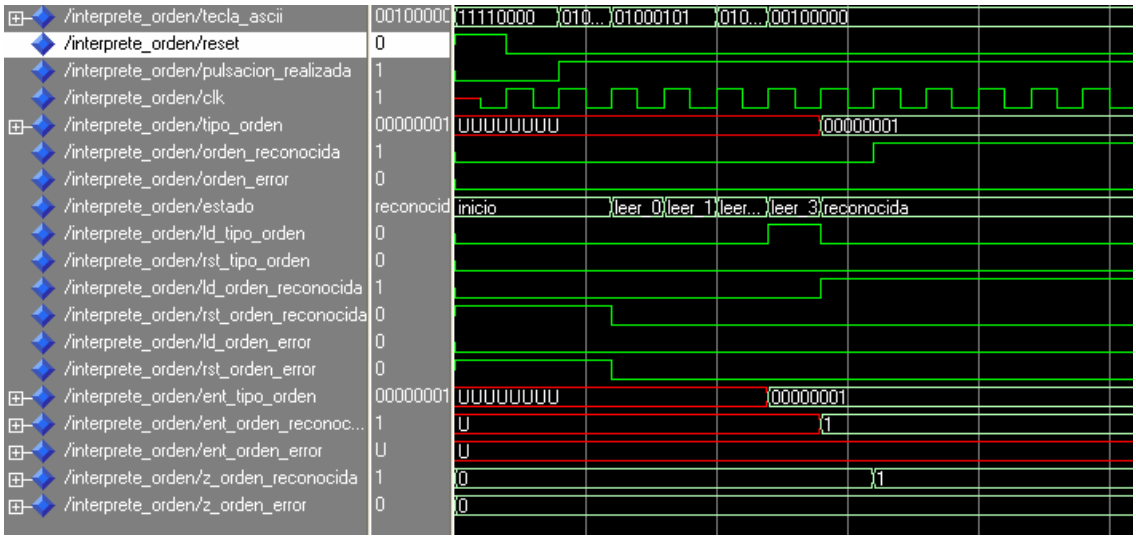


Figura 73: Simulación del intérprete de órdenes

La simulación muestra como el módulo es capaz de reconocer la orden compuesta por los caracteres *LEER* y se ve como va pasando por los distintos estados hasta que llega al estado final de reconocida.

5.1.3. Lector de datos

Descripción

Es el módulo que se utiliza para reconocer los datos de cada una de las órdenes. Permite obtener a partir de dos caracteres ASCII hexadecimales el valor binario que representan.

Los caracteres son proporcionados uno por uno por un bus de entrada y se evalúan en cada ciclo de reloj en el que además se encuentre activa la señal de *pulsacion_realizada*.

Si después de dos caracteres se ha reconocido correctamente un byte entonces se activa la señal *byte_reconocido* y se establece su valor binario en el bus de salida.

Si se ha producido algún error durante el proceso de reconocimiento del byte se activa la señal de error del módulo.

Si tras producirse un error se desean reconocer nuevos bytes es necesario resetear el componente y capacitarlo para comenzar el proceso de reconocimiento.

Esquema

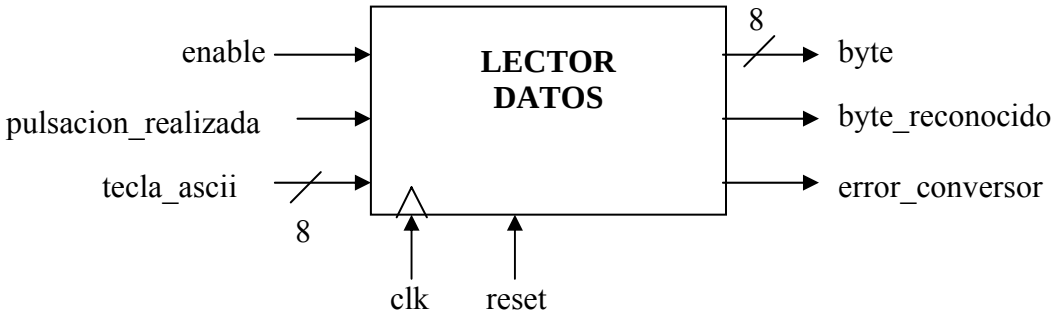


Figura 74: Lector de datos

Entradas/Salidas

Tabla 21: Entradas del lector de datos

Entradas	Descripción
tecla_ascii [7:0]	- Contiene el valor ASCII del carácter que hay que reconocer en el ciclo actual
reset	- Señal de reset asíncrono del sistema
pulsacion_realizada	- Señal que indica que la tecla que hay en la entrada *tecla_ascii* es válica en el ciclo actual
clk	- Señal de reloj
enable	- Señal de capacitación del módulo

Tabla 22: Salidas del lector de datos

Salidas	Descripción
byte [7:0]	- Señal que devuelve el valor del byte que se ha reconocido - Únicamente será válido si la salida *byte_reconocido* está activada
byte_reconocido	- Señal que indica si se ha reconocido un dato válido
error_conversor	- Señal que indica se ha producido un error durante el reconocimiento del dato

Ruta de datos

La representación esquemática de la ruta de datos de este módulo es la siguiente:

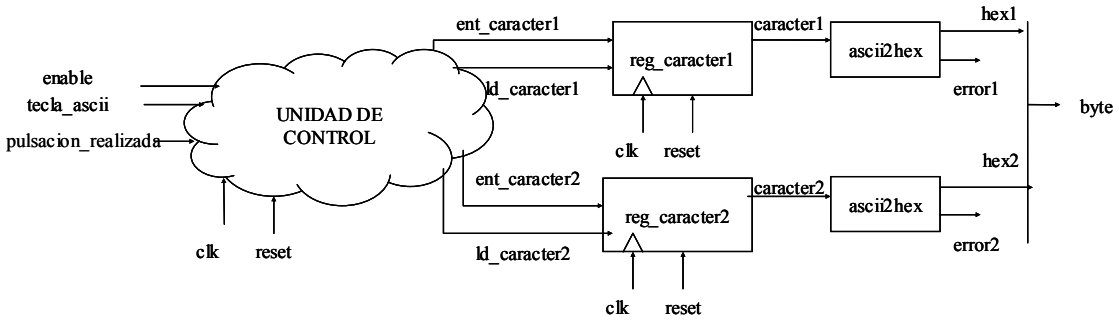


Figura 75: Ruta de datos del lector de datos

El funcionamiento del módulo se basa en el uso de dos registros que almacenan los dos caracteres que son necesarios para reconocer un byte de datos. Asociados con cada uno de ellos se encuentran dos módulos conversores de código ASCII hexadecimal a un valor binario de 4 bits. La concatenación de los 8 bits que salen de los conversores produce como resultado el valor binario del byte representado por los dos caracteres ASCII.

Se producirá un error en el caso de que alguno de los caracteres que se le suministren a los conversores no sea hexadecimal.

Cada uno de los bytes reconocidos es llevado por la unidad de control a un banco de registros. Este banco tiene la siguiente estructura:

Figura 76: Banco de registros

Este banco de registros no tiene ninguna característica especial, únicamente se muestra en la salida *datos_out* el contenido del registro indicado por la referencia *numero_registro*. Si se desean cargar nuevos bytes sólo habrá que indicarle el byte que se quiere cargar, el registro en el que se almacenará y activar la señal de load para que en el ciclo siguiente el valor quede almacenado.

Unidad de control

La máquina de estados correspondiente con el controlador del módulo es la siguiente:

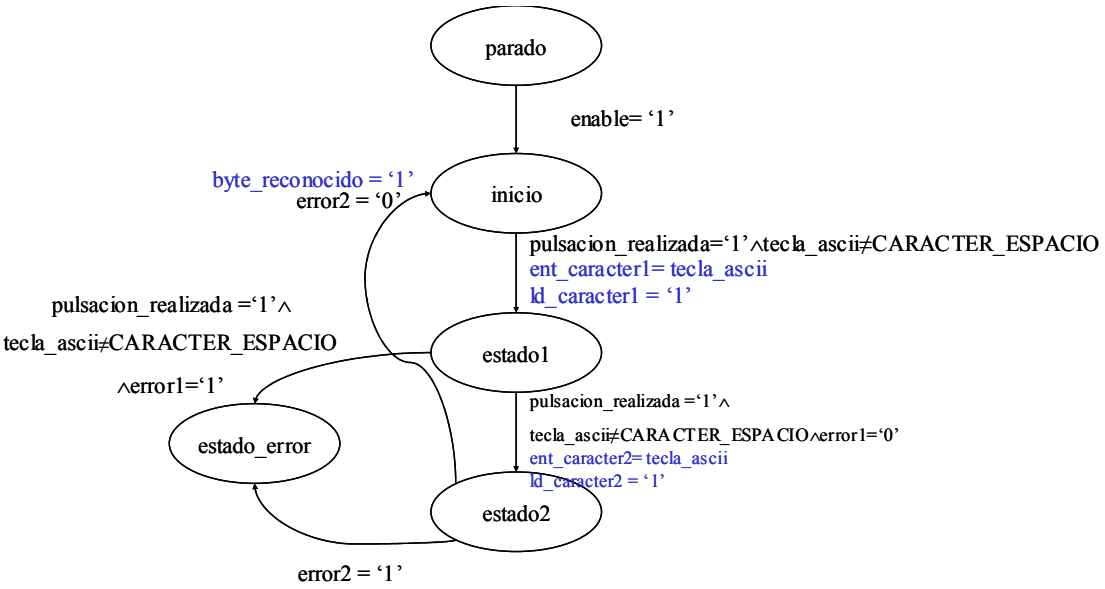


Figura 77: Diagrama de transición del lector de datos

El sistema parte de un estado inicial *parado* en el cual permanece hasta que sea capacitado. Entonces a cada ciclo de reloj que la señal *pulsacion_realizada* se encuentre activada se producirá una transición de estado que reconocerá la mitad de un byte representado en hexadecimal. Si se producen correctamente dos reconocimientos se producirá una transición que va del *estado2* al *inicio* durante la cual se activará la señal de *byte_reconocido* y se pondrá en el bus *byte* la representación binaria del byte compuesto de los dos caracteres. Si alguno de los caracteres introducidos no corresponde con un valor hexadecimal entonces se producirá un error y el sistema quedará bloqueado y se activará la señal de *error_conversor*.

A partir de la máquina de estados obtenemos las siguientes tablas de transición de estados y de salidas:

Transición de estados

Tabla 22: Transición de estados del lector de datos

Estado	enable	pulsacion _realizada	tecla_ascii	error1	error2	Estado Siguiente
parado	1	-	-	-	-	inicio
inicio	-	1	≠CARACTER _ESPACIO	-	-	estado1
estado1	-	1	≠CARACTER _ESPACIO	0	-	estado2
estado1	-	1	≠CARACTER _ESPACIO	1	-	estado_error
estado2	-	-	-	-	1	estado_error
estado2	-	-	-	-	0	inicio

Salidas

Tabla 24: Valores de las salidas del lector de datos

Estado	enable	pulsacion _realizada	tecla_ascii	error1	error2	ent_caracter1	ld_caracter1	ent_caracter2	ld_caracter2	error_conversor	byte_reconocido
inicio	-	1	≠CARACTER _ESPACIO	-	-	tecla_ascii	1	-	0	0	0
estado1	-	1	≠CARACTER _ESPACIO	-	-	-	0	tecla_ascii	1	0	0
estado2	-	-	-	-	0	-	0	-	0	0	1
estado _error	-	-	-	-	-	-	0	-	0	1	0

Simulaciones

Ahora vamos a realizar algunas simulaciones para mostrar el funcionamiento del módulo.

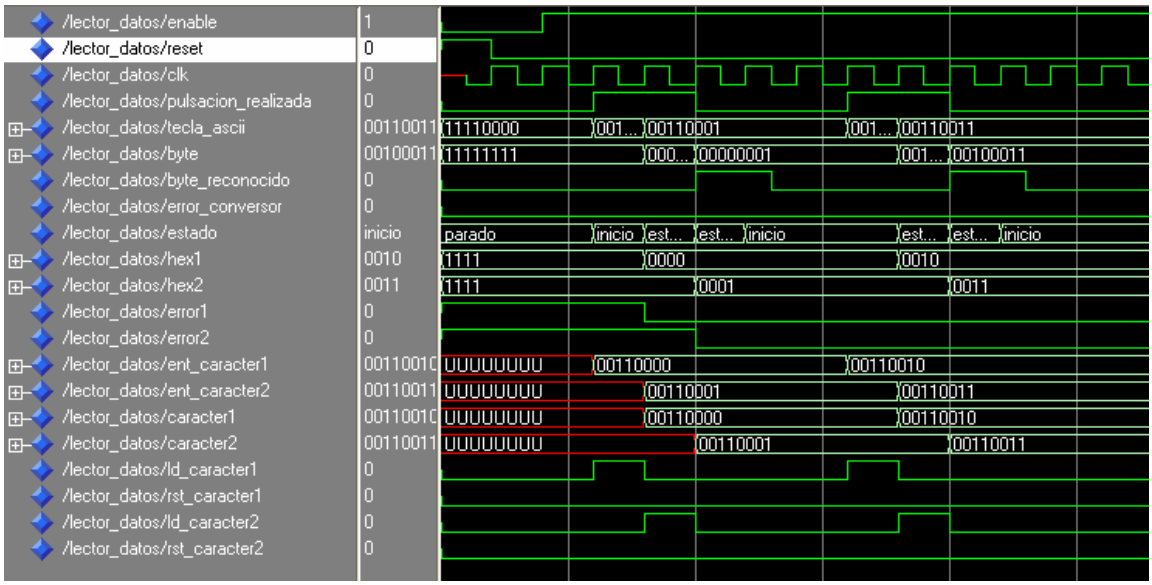


Figura 78: Simulación del lector de datos

En la simulación pueden verse todas las señales activadas y las transiciones de estados asociadas al reconocimiento de dos bytes hexadecimales. El proceso continuaría de la misma forma indefinidamente o hasta que se introdujese un byte mal.

Se puede ver también como la salida es la concatenación de los dos valores binarios de los hexadecimales de la entrada.

5.1.4. Ejecutor de órdenes

Descripción

Este es el módulo encargado de realizar la ejecución de la orden introducida por el usuario cuando el proceso se encuentra en la fase de ejecución.

En función del tipo de orden que se haya reconocido realizará unas acciones u otras activando las señales que sean necesarias para llevar a cabo la ejecución de la orden.

Esquema

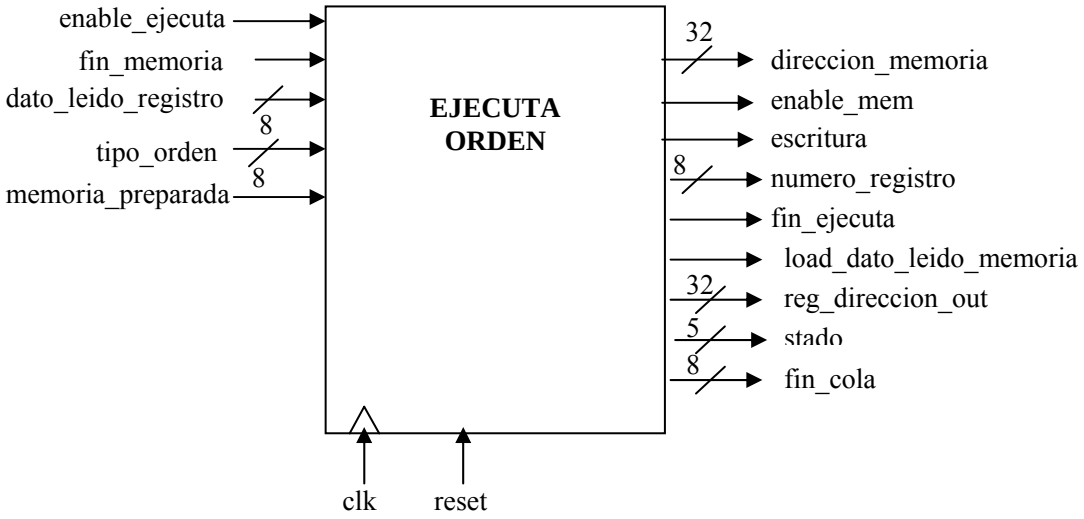


Figura 79: Ejecutor de órdenes

Entradas/Salidas

Tabla 25: Entradas del ejecutor de órdenes

Entradas	Descripción
reset	- Señal de reset asíncrono del sistema
clk	- Señal de reloj
tipo_orden [7:0]	- Señal que indica el tipo de orden que se ha reconocido
enable_ejecuta	- Señal que capacita al módulo
dato_leido_registro [7:0]	- Señal que contiene el dato leído del banco de registros cuyo número viene especificado por la señal *numero_registro*
fin_memoria	- Señal que indica al módulo cuando se ha finalizado una operación de memoria

Tabla 26: Salidas del ejecutor de órdenes

Salidas	Descripción
direccion_memoria [31:0]	- Señal que indica la dirección de memoria sobre la que se realizaría la posible operación de memoria
escritura	- Señal que indica si se va a realizar sobre la memoria una lectura, si la señal está en baja, o una escritura, si está en alta
numero_registro [7:0]	- Señal que indica el número de registro que desea ser leído por el módulo
fin_ejecuta	- Señal que indica cuando la orden actual ha terminado de ejecutarse
load_dato_leido_memoria	- Señal que activa la señal de carga del registro que almacena el dato que se hubiera leido de la memoria tras una orden de lectura
reg_direccion_out [31:0]	- Señal que indica la dirección de memoria que se va a utilizar en la orden
enable_mem	- Señal que capacita el módulo interfaz de memoria
fin_cola [7:0]	- Señal que contiene el valor del final de la cola
memoria_preparada	- Señal que indica cuando se puede realizar una operación de memoria

Ruta de datos

La representación esquemática de la ruta de datos de este módulo es la siguiente:

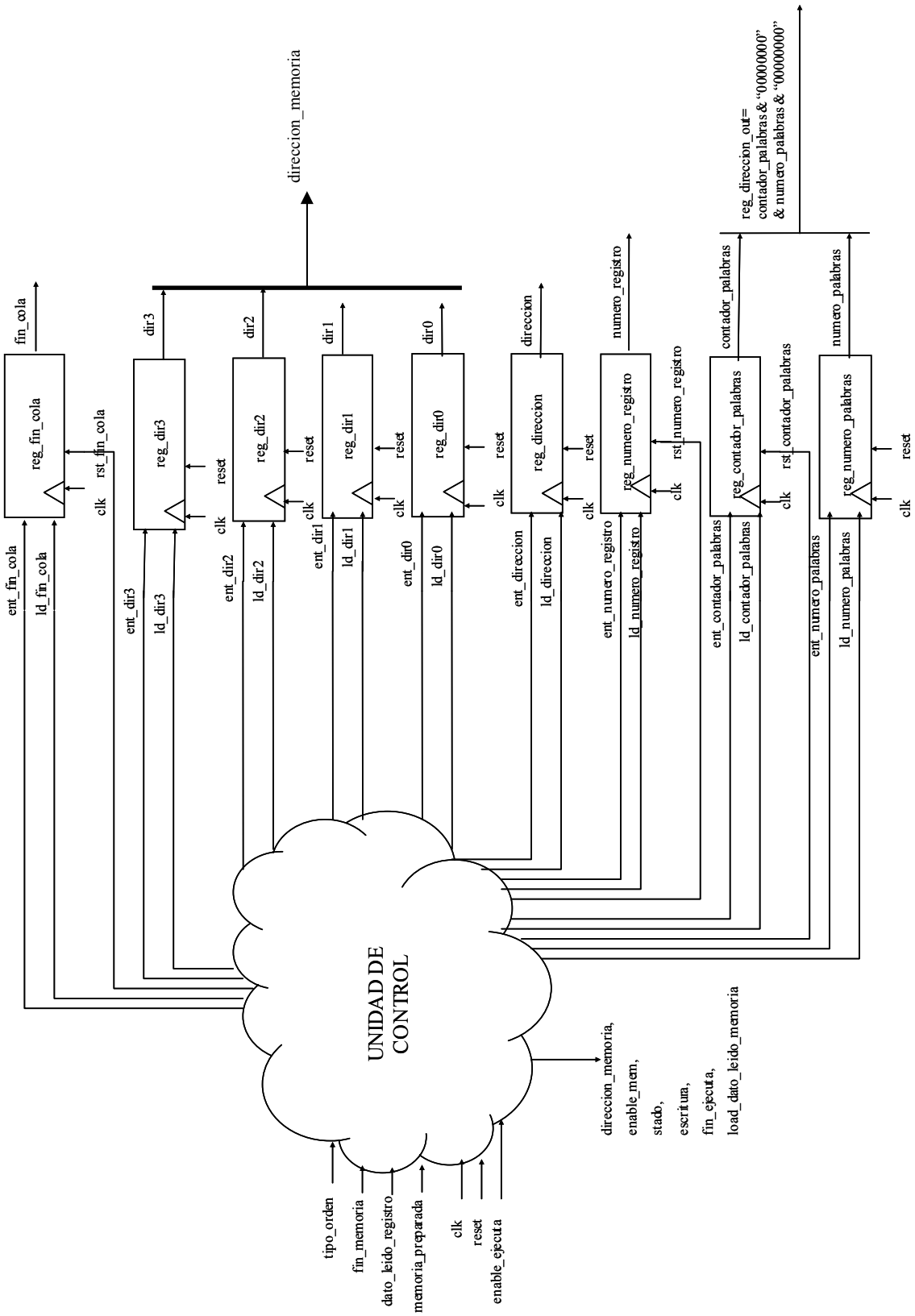


Figura 80: Ruta de datos del ejecutor de órdenes

Como se puede observar la ruta de datos del módulo está formada principalmente por registros ya que su función consiste principalmente en la generación de señales que envía a los módulos que intervienen en la instrucción. Estas señales se almacenan en los registros de la ruta de datos antes de ser enviadas para evitar problemas de estabilidad.

Unidad de control

La máquina de estados correspondiente con el controlador del módulo es la siguiente:

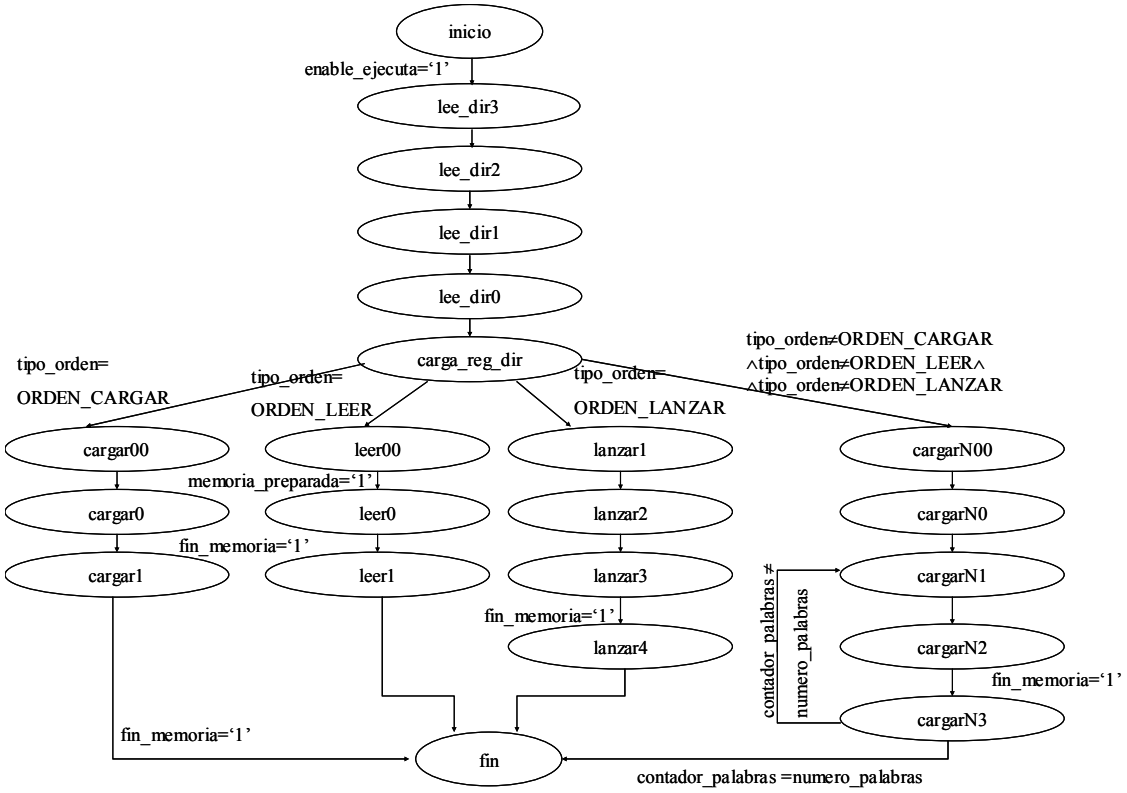


Figura 81: Diagrama de transición de estados del ejecutor de órdenes

El funcionamiento del módulo consiste básicamente en tomar la orden y en función de ésta sus argumentos del banco de registro y activar las señales necesarias para llevar a cabo dicha orden.

Como se puede observar, independientemente de la orden a realizar, los primeros estados se encargan de leer los cuatro primeros registros del banco para obtener la dirección de memoria. En el caso de que la orden sea lanzar dichos bancos no contendrán una dirección ya que no se utiliza pero como en la mayoría de los casos si que se utiliza entonces se ha dejado como una opción por defecto.

Una vez leída esta dirección se procede a la distinción de casos en función del tipo de orden. Aquí se divide el flujo de ejecución ya que cada orden tiene sus propias señales de control para funcionar.

Como se ve en el caso de las operaciones de memoria la ejecución consiste en establecer los valores necesarios de dirección y tipo de operación al interfaz, esperar a que finalice la ejecución y por último almacenar el resultado.

La operación lanzar que en teoría es diferente también realiza una operación de memoria ya que modifica los valores de la cola de tareas que está almacenada en memoria.

Cuando se ha finalizado la ejecución de la orden, el controlador de ejecución queda bloqueado en un estado *fin* en el cual permanece siempre hasta que el módulo se resetea. Esto quiere decir que si se desean ejecutar órdenes sucesivas es necesario resetear el módulo antes de la ejecución de cada una para que vuelva a su estado inicial y permita repetir el flujo para la nueva orden.

A partir de la máquina de estados obtenemos las siguientes tablas de transición de estados y de salidas:

Transición de estados

Tabla 27: Transición de estados del ejecutor de órdenes

Estado	enable_ ejecuta	tipo_ orden	fin_ memoria	memoria_ preparada	contador_ palabras	Estado Siguiente
inicio	1	-	-	-	-	lee_dir3
lee_dir3	-	-	-	-	-	lee_dir2
lee_dir2	-	-	-	-	-	lee_dir1
lee_dir1	-	-	-	-	-	lee_dir0
lee_dir0	-	-	-	-	-	carga_reg_dir
carga_reg_dir	-	ORDEN_ CARGAR	-	-	-	cargar00
carga_reg_dir	-	ORDEN _LEER	-	-	-	leer00
carga_reg_dir	-	ORDEN_ LANZAR	-	-	-	lanzar1
cargar_reg_dir		≠ORDEN_CARGAR ∧≠ORDEN_LEER ∧≠ORDEN_LANZAR	-	-	-	cargarN00
cargar00	-	-	-	-	-	cargar0
cargar0	-	-	-	-	-	cargar1
cargar1	-	-	1	-	-	fin
leer00	-	-	-	1	-	leer0
leer0	-	-	1	-	-	leer1
leer1	-	-	-	-	-	fin
cargarN00	-	-	-	-	-	cargarN0
cargarN0	-	-	-	-	-	cargarN1
cargarN1	-	-	-	-	-	cargarN2
cargarN2	-	-	1	-	-	cargarN3
cargarN3	-	-	-	-	numero_palabras	fin
cargarN3	-	-	-	-	≠numero_palabras	cargarN1
lanzar1	-	-	-	-	-	lanzar2
lanzar2	-	-	-	-	-	lanzar3
lanzar3	-	-	1	-	-	lanzar4
lanzar4	-	-	-	-	-	fin
fin	-	-	-	-	-	fin

Salidas

Tabla 28: Valores de las salidas del ejecutor de órdenes

Estado	tipo_orden	fin_memoria	ld_numero_registro	rst_numero_registro	ent_numero_registro	ld_dir3	ent_dir3	ld_dir2	ent_dir2	ld_dir1	ent_dir1	ld_dir0	ent_dir0	ld_direccion	ent_direccion	rst_contador_palabras	ld_contador_palabras	ent_contador_palabras	ld_numero_palabras	ent_numero_palabras	ld_fin_cola	rst_fin_cola	ent_fin_cola	escritura	fin_ejecuta	load_dato_leido_memoria	enable_mem
inicio	-	-	0	1	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	0
lee_dir3	-	-	1	0	"00000001"	1	dato_ leido_ registro	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	0
lee_dir2	-	-	1	0	"00000010"	0	-	1	dato_ leido_ registro	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	0
lee_dir1	-	-	1	0	"00000011"	0	-	0	-	1	dato_ leido_ registro	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	0
lee_dir0	-	-	1	0	"00000100"	0	-	0	-	0	-	1	dato_ leido_ registro	0	-	0	0	-	0	-	0	0	-	0	0	0	0
carga_ regdir	ORDEN_ CARGAR	-	1	0	-	0	-	0	-	0	-	0	-	1	dir3& dir2& dir& dir0	0	0	-	0	-	0	0	-	0	0	0	0
carga_ regdir	ORDEN_ LANZAR	-	1	0	-	0	-	0	-	0	-	0	-	1	dir3& dir2& dir1& dir0	0	0	-	0	-	0	0	-	0	0	0	0
carga_ regdir	'ORDEN _CARGAR Ù'ORDEN _LANZAR	-	1	0	-	0	-	0	-	0	-	0	-	1	dir3& dir2& dir1& dir0	0	0	-	0	-	0	0	-	0	0	0	0
cargar0	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	0
cargar1	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	1	0	0	1
leer00	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	1
leer0	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	1
leer1	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	1	0
cargarN0	-	-	1	0	"00000101"	0	-	0	-	0	-	0	-	0	-	1	0	-	1	dato_ leido_ registro	0	0	-	0	0	0	0
cargarN1	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	0
cargarN2	-	1	0	0	-	0	-	0	-	0	-	0	-	0	-	0	1	contador _palabras +'1'	0	-	0	0	-	1	0	0	1
cargarN2	-	0	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	1	0	0	1
cargarN3	-	-	1	0	"00000101" +contador _palabras	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	0	0	0
lanzar1	-	-	1	0	"00000000"	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	1	0	0	0
lanzar2	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	1	0	0	1
lanzar3	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	1	0	0	1
lanzar4	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	1	0	z_fin _cola +'1'	0	0	0	0
fin	-	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	0	-	0	0	-	0	1	0	0

Simulaciones

Ahora vamos a realizar algunas simulaciones para mostrar el funcionamiento del módulo.

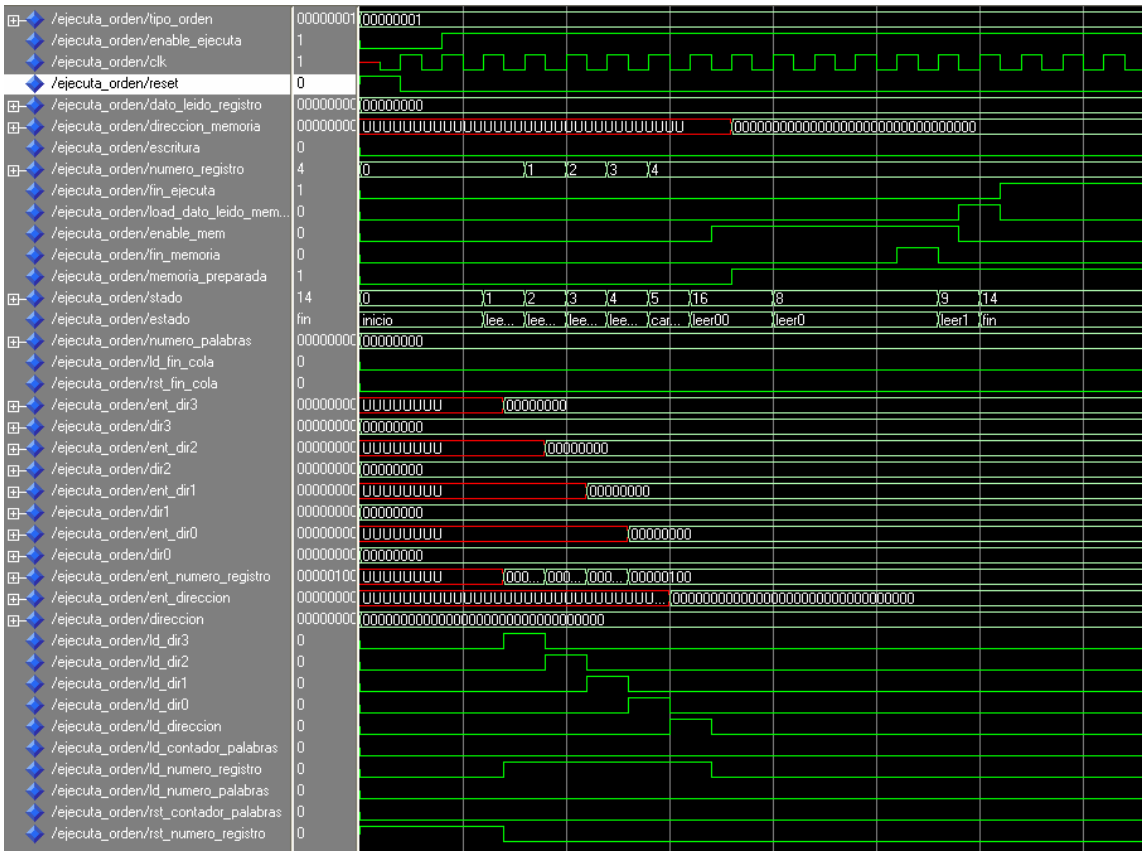


Figura 82: Simulación del ejecutor de órdenes (1)

En esta simulación se pueden ver las señales que se activan durante la ejecución de una orden de lectura.

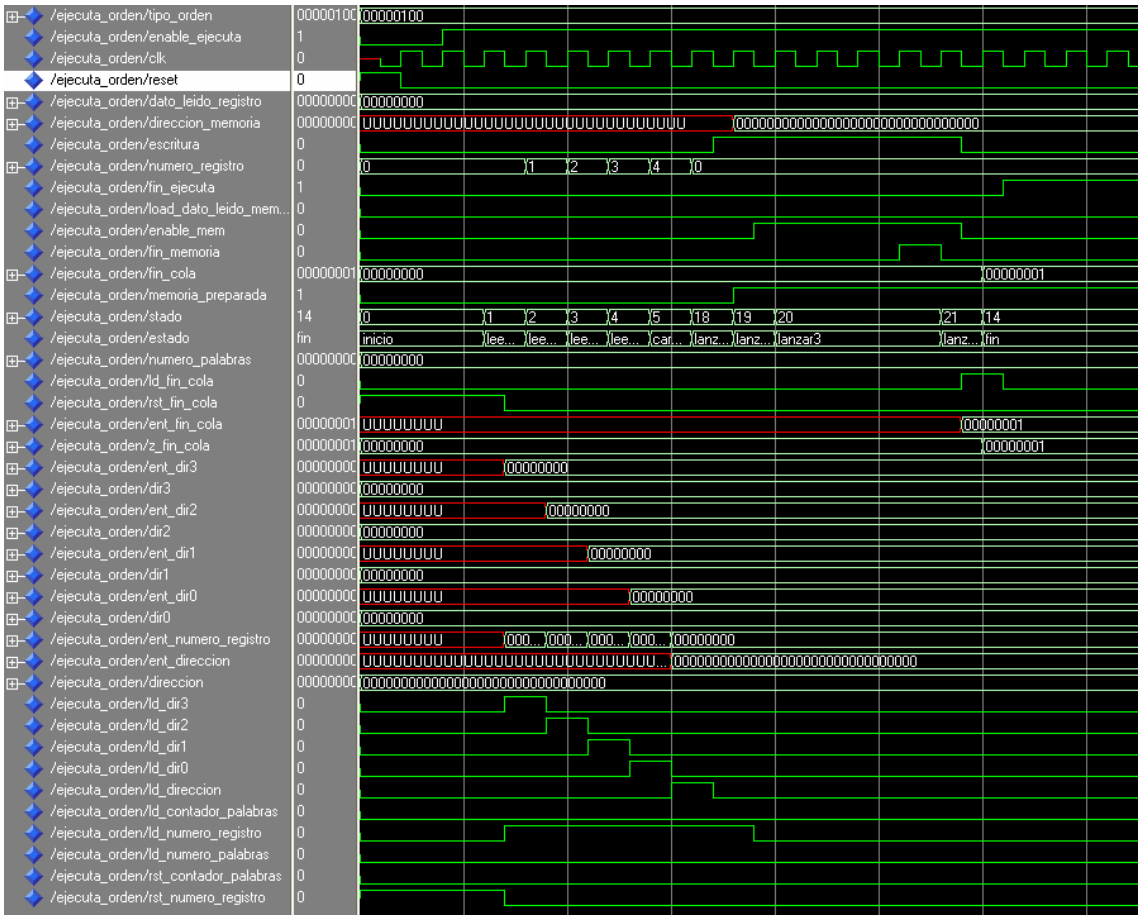


Figura 83: Simulación del ejecutor de órdenes (2)

En esta otra simulación tenemos lo mismo pero para una orden de lanzamiento de tareas.

5.2. Interfaz de memoria

Descripción

Este es el módulo encargado de gestionar las señales del controlador generado por el MIG. Simplifica su utilización hasta el punto que sólo es necesario establecer los parámetros de ejecución en los buses de entrada y tras capacitar el módulo, esperar a que se active su señal de salida de *fin_memoria* indicando que la operación actual realizada sobre la memoria ha finalizado.

Esquema

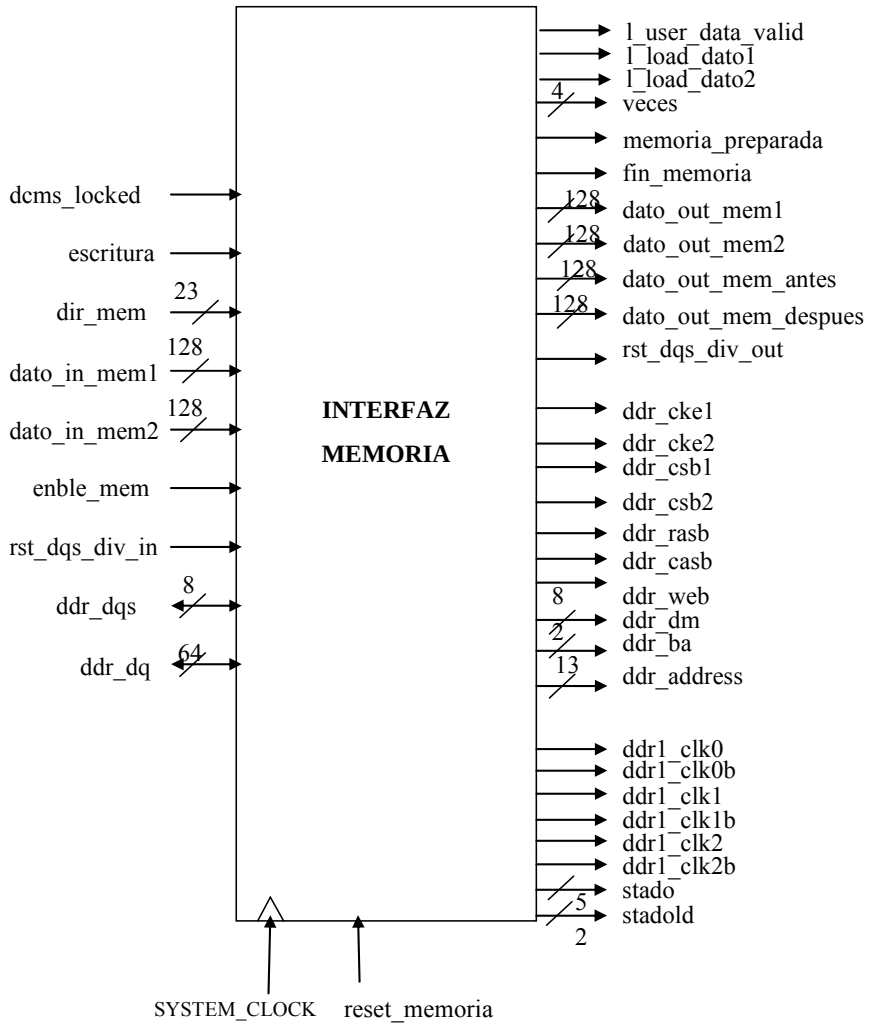


Figura 84: Interfaz de memoria

Entradas/Salidas

Tabla 29: Entradas del interfaz de memoria

Entradas	Descripción
SYSTEM_CLOCK	- Señal de reloj de sistema
reset	- Señal de reset
escritura	- Señal que indica cuando se desea realizar una operación de lectura si está en baja o de escritura si está en alta
dir_mem [22:0]	- Señal que contiene la dirección de memoria sobre la que se va a trabajar
dato_in_mem1[127:0]	- Señal que contiene la primera palabra de 128 bits que se escribirá en una operación de escritura
dato_in_mem2[127:0]	- Señal que contiene la segunda palabra de 128 bits que se escribirá en una operación de escritura
rst_dqs_div_in	- Señal de realimentación del reset del DQS

Tabla 30: Salidas del interfaz de memoria

Salidas	Descripción
memoria_preparada	- Señal que indica cuando la memoria está disponible para ser utilizada
fin_memoria	- Señal que indica cuando el módulo de memoria ha terminado de realizar una operación. Si esta señal está activa querrá decir que si la operación realizada ha sido una lectura, los datos leídos se encontrarán en el bus de datos. Si por el contrario era una operación de escritura, la señal indicará que el proceso se ha completado
dato_out_mem1 [127:0]	- Señal que contiene la primera palabra de 128 bits leída tras una operación de lectura
dato_out_mem2 [127:0]	- Señal que contiene la segunda palabra de 128 bits leída tras una operación de lectura
rst_dqs_div_out	- Señal de realimentación del reset del DQS
ddr_cke1	- Señal de clock enable
ddr_cke2	- Señal de clock enable
ddr_csb1	- Señal de chip select
ddr_csb2	- Señal de chip select
ddr_rasb	- Señal de comando
ddr_casb	- Señal de comando

ddr_web	- Señal de comando
ddr_dm [7:0]	- Señal de máscara de datos
ddr_ba [1:0]	- Señal de dirección de banco
ddr_address [22:0]	- Señal de dirección
ddr_clk0	- Señal de reloj
ddr_clk0b	- Señal de reloj
ddr_clk1	- Señal de reloj
ddr_clk1b	- Señal de reloj
ddr_clk2	- Señal de reloj
ddr_clk2b	- Señal de reloj

Tabla 31: Señales de entrada/salida del interfaz de memoria

Entrada/Salida	Descripción
ddr_dqs [7:0]	- Señal de strobe de datos
ddr_dq [63:0]	- Señal de bus de datos

Ruta de datos

La representación esquemática de la ruta de datos de este módulo es la siguiente:

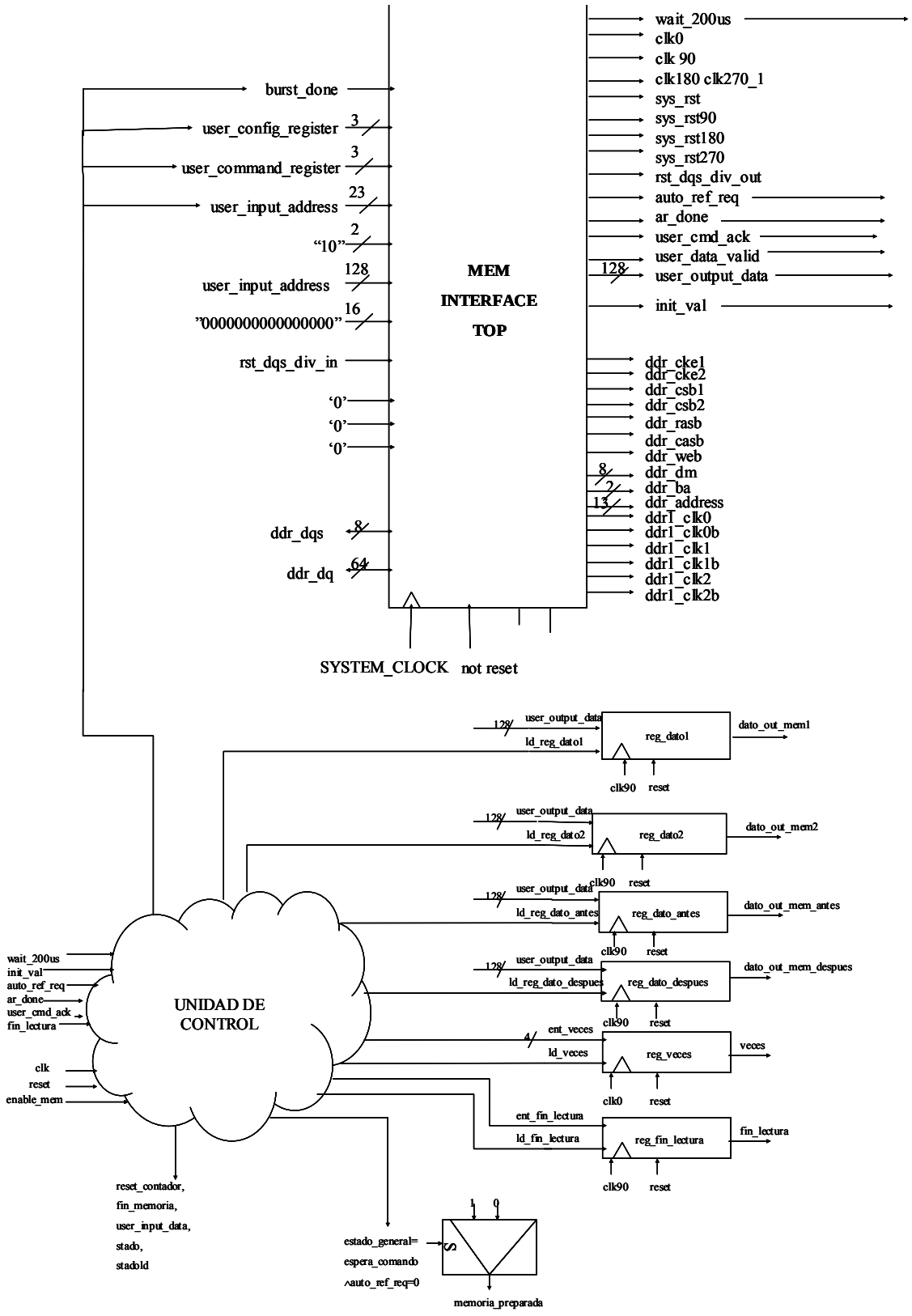


Figura 85: Ruta de datos del interfaz de memoria

La ruta de datos está formada casi exclusivamente por registros. Estos registros almacenan la señal de dirección controlador de memoria, datos de entrada, datos de salida, etc. Además existe un contador utilizado para llevar la cuenta de los ciclos de espera que hay que realizar en algunas operaciones.

Hasta aquí podría ser una ruta de datos cualquiera pero lo que la hace diferente de otra normal es que esta ruta de datos está gobernada por cuatro relojes los cuales van a unos componente u otros en función de los requerimientos impuestos por el controlador de memoria.

Unidad de control

La máquina de estados correspondiente con el controlador del módulo es la siguiente:

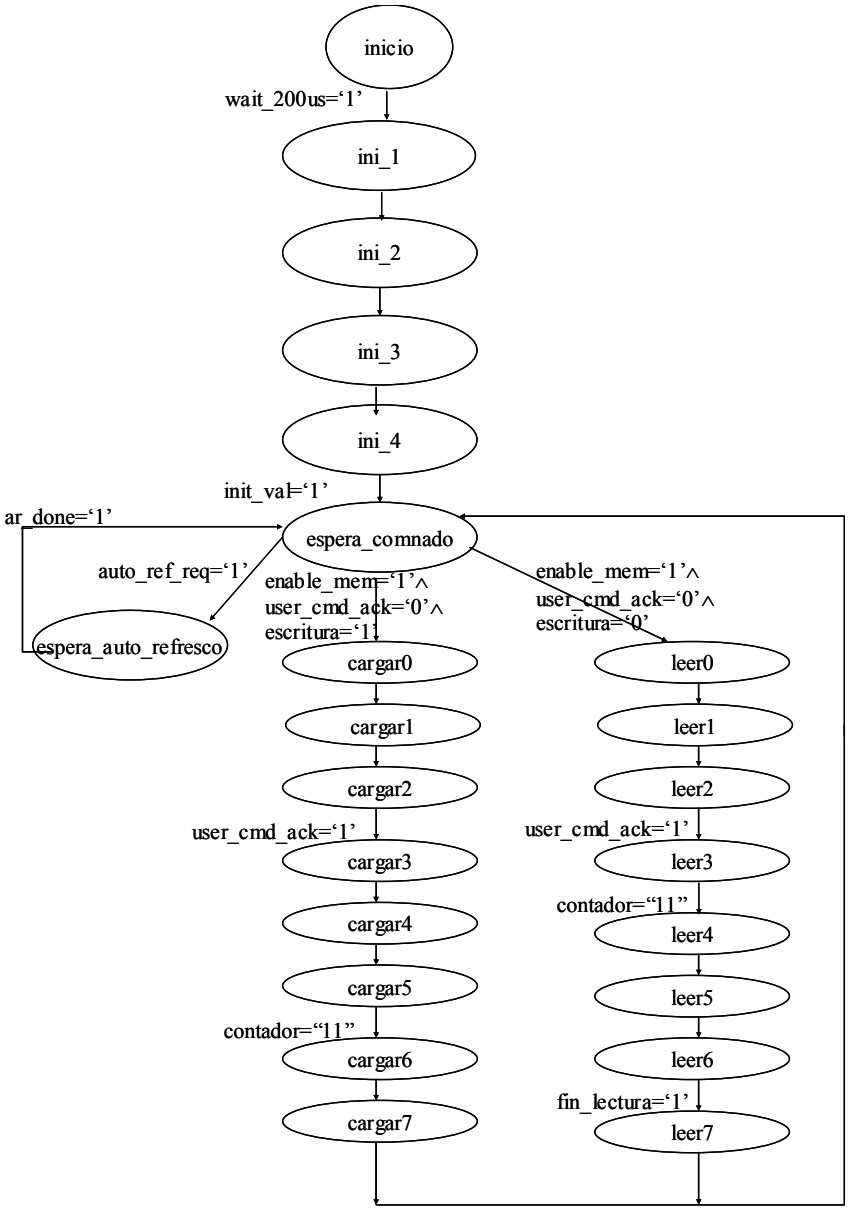


Figura 86: Diagrama de transición de estados del interfaz de memoria (1)

Como se puede observar esta máquina de estados es la encargada de poner las señales en unos tiempos concretos que son necesarios para un correcto funcionamiento del controlador de memoria.

En función de los estados y del tipo de señal se generarán las salidas utilizando un tipo de reloj u otro. Esto incrementa un poco la dificultad del diseño ya que ha sido necesario realizar un sistema en el que señales generadas con distinto reloj deben ser asignadas siguiendo un orden preestablecido y algunas incluso deben estar sincronizadas.

El interfaz parte de un estado inicial en el que permanece hasta que se activa la señal que india que han pasado 200 μs que son necesarios para poder utilizar la memoria una vez que se ha alimentado.

Pasado este tiempo se procede a realizar la inicialización de la memoria. Esto se consigue en los cuatro estados siguientes. En éste último estado permanece hasta que la memoria le indica al sistema que ha completado la inicialización activando la señal *init_val*.

Ocurrido esto el interfaz permanece en un estado de espera del cual saldrá únicamente para realizar la operación de auto-refresco o bien tras la llegada de una nueva operación.

Si se produce una petición de auto-refresco por parte del controlador, el interfaz transita a un estado en el cual permanece hasta que finaliza dicho proceso. Una vez terminado volvería al estado de espera.

Cuando el interfaz se encuentra en el estado de espera y llega un comando de lectura se produce una transición a la rama que contiene los estados propios de la lectura. En dichos estados el interfaz genera las señales necesarias sobre el controlador para realizar una lectura completa. Esta lectura consta de varias fases como se explica en el capítulo dedicado a la memoria DDR SDRAM. Una vez finalizada la lectura y almacenados los datos leídos el interfaz regresa al estado de espera.

Si el comando que le llega en el estado de espera es una escritura, al igual que en el caso de la lectura, el interfaz recorrería la rama dedicada a la escritura en memoria que permite generar la señales sobre el controlador para poder llevarla a cabo. Estas señales junto con las fases que requiere una escritura están explicadas en el capítulo de la memoria DDR SDRAM.

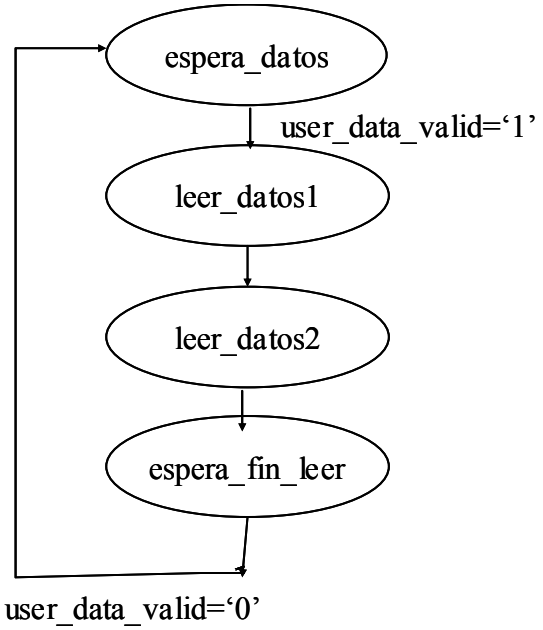


Figura 87: Diagrama de transición de estados del interfaz de memoria (2)

Esta otra máquina de estados es la encargada de, en el caso de realizar un comando de lectura, coger los datos del bus y almacenarlos en un registro cuando sean válidos. Para comprobar el momento en el que los datos son válidos, el módulo monitoriza el estado de la señal *data_valid* del controlador como se explica en el capítulo de la memoria DDR SDRAM. El funcionamiento de esta máquina de estados es que mientras no esté activa la señal anterior el sistema permanece en un estado de espera. Una vez activada dicha señal se produce el almacenamiento de los datos leídos que se encuentran sobre el bus. Es necesario realizar dos almacenados de datos, uno por cada ciclo, en cada lectura que se realice. Cuando finaliza la captura de los dos datos se regresa al estado de espera donde permanece hasta la siguiente lectura. Esta máquina de estados es independiente de la otra y su funcionamiento como ya hemos dicho sólo depende del valor de la señal que indica si hay datos en el bus por lo que puede ejecutarse en cualquier momento.

A partir de la máquina de estados obtenemos las siguientes tablas de transición de estados y de salidas:

Transición de estados

Tabla 32: Transición de estados del interfaz de memoria (1)

Estado	wait_200us	init_val	auto_ref_req	ar_done	enable_mem	user_cmd_ack	escritura	contador	fin_lectura	Estado Siguiente
inicio	0	-	-	-	-	-	-	-	-	ini_1
ini_1	-	-	-	-	-	-	-	-	-	ini_2
ini_2	-	-	-	-	-	-	-	-	-	ini_3
ini_3	-	-	-	-	-	-	-	-	-	ini_4
ini_4	-	1	-	-	-	-	-	-	-	espera_comando
espera_comando	-	-	1	-	-	-	-	-	-	espera_auto_refresco
espera_comando	-	-	-	-	1	0	1	-	-	cargar0
espera_comando	-	-	-	-	1	0	0	-	-	leer0
espera_auto_refresco	-	_	-	1	-	-	-	-	-	espera_comando
leer0	-	-	-	-	-	-	-	-	-	leer1
leer1	-	-	-	-	-	-	-	-	-	leer2
leer2	-	-	-	-	-	1	-	-	-	leer3
leer3	-	-	-	-	-	-	-	"11"	-	leer4
leer4	-	-	-	-	-	-	-	-	-	leer5
leer5	-	-	-	-	-	-	-	-	-	leer6
leer6	-	-	-	-	-	-	-	-	1	leer7
leer7	-	-	-	-	-	-	-	-	-	espera_comando
cargar0	-	-	-	-	-	-	-	-	-	cargar1
cargar1	-	-	-	-	-	-	-	-	-	cargar2
cargar2	-	-	-	-	-	1	-	-	-	cargar3
cargar3	-	-	-	-	_	_	_	_	_	cargar4
cargar4	-	-	-	-	-	-	-	-	-	cargar5
cargar5	-	-	-	-	-	-	-	"11"	-	cargar6
cargar6	-	-	-	-	-	-	-	-	-	cargar7
cargar7	-	-	-	-	-	-	-	-	-	espera_comando

Tabla 33: Transición de estados del interfaz de memoria (2)

Estado	user_data_valid	Estado Siguiente
espera_dato	1	leer_datos1
leer_datos1	-	leer_datos2
leer_datos2	-	espera_fin_leer
espera_fin_leer	0	espera_datos

Salidas

Tabla 34: Valores de las salidas del interfaz de memoria

Estado	user_config_register	user_input_address	ld_veces	ent_veces	reset_contador	burst_done	fin_memoria	user_input_data	user_command_register	ent_fin_lectura	ld_fin_lectura	ld_reg_dato2	ld_reg_dato_despues	stado_ld	stado
inicio	-	-	0	-	0	0	0	-	-	-	0	0	0		"00000"
ini_1	"1000110010"	-	0	-	0	0	0	-	-	-	0	0	0	-	"00001"
ini_2	-	-	0	-	0	0	0	-	-	-	0	0	0	-	"00010"
ini_3	-	-	0	-	0	0	0	-	"010"	-	0	0	0	-	"00011"
ini_4	-	-	0	-	0	0	0	-	"000"	-	0	0	0	-	"00100"
espera_ comando	-	-	0		0	0	0	-	"000"	-	0	0	0	-	"00101"
espera _auto_ refresco	-	-	0	-	0	0	0	-	-	-	0	0	0	-	"00110"
leer0	-	dir_ mem	1	z_veces +'1'	0	0	0	-	-	-	0	0	0	-	"00111"
leer1	-	-	0	-	0	0	0	-	"110"	-	0	0	0	-	"01000"
leer2	-	-	0	-	1	0	0	-	-	-	0	0	0	-	"01001"
leer3	-	-	0	-	0	0	0	-	-	-	0	0	0	-	"01010"
leer4	-	-	0	-	0	1	0	-	-	-	0	0	0	-	"01011"
leer5	-	-	0	-	0	1	0	-	-	-	0	0	0	-	"01100"
leer6	-	-	0	-	0	0	0	-	"000"	-	0	0	0	-	"01101"
leer7	-	-	0	-	0	0	1	-	-	-	0	0	0	-	"01110"
cargar0	-	dir_ mem	1	z_veces +'1'	0	0	0	-	-	-	0	0	0	-	"01111"
cargar1	-	-	0	-	0	0	0	-	"100"	-	0	0	0	-	"10000"
cargar2	-	-	0	-	1	0	0	-	-	-	0	0	0	-	"10001"
cargar3	-	-	0	-	0	0	0	dato _in_ mem1	-	-	0	0	0	-	"10010"
cargar4	-	-	0	-	0	0	0	dato _in_ mem2	-	-	0	0	0	-	"10011"
cargar5	-	-	0	-	0	0	0	-	-	-	0	0	0	-	"10100"
cargar6	-	-	0	-	0	1	0	-	-	-	0	0	0	-	"10101"
cargar7	-	-	0	-	0	1	1	-	-	-	0	0	0	-	"10110"
espera _datos	-	-	0	-	0	0	0	-	-	0	1	0	0	"00"	-
leer_ datos1	-	-	0	-	0	0	0	-	-	-	0	1	0	"01"	-
leer_ datos2	-	-	0	-	0	0	0	-	-	-	0	0	1	"10"	-
espera _fin_ leer	-	-	0	-	0	0	0	-	-	1	1	0	0	-	-

Simulaciones

Ahora vamos a realizar algunas simulaciones para mostrar el funcionamiento del módulo.

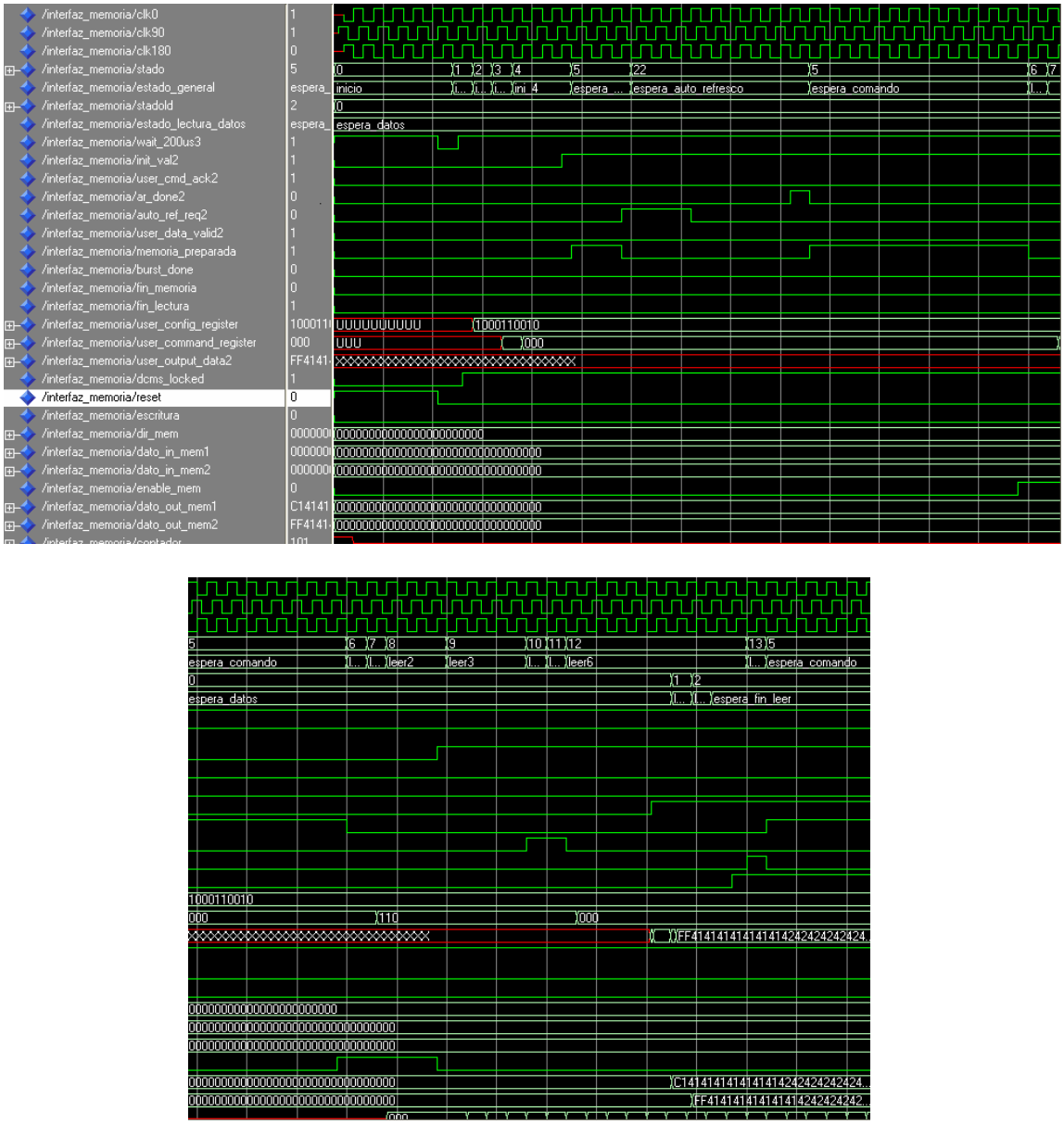


Figura 88: Simulación del interfaz de memoria

Esta simulación corresponde con una operación de lectura realizada sobre la memoria. Se pueden observar los tiempos y el momento en el que se activan las señales de control que van al controlador de la memoria.

5.3. Planificador

Descripción

Este módulo es el encargado de controlar las tareas que hay en la FPGA. Entre sus funcionalidades están la de almacenar en una cola las tareas que el usuario que quiere poner en ejecución, llevar desde la cola hasta la FPGA la tarea que le toque ejecutarse cuando haya un hueco y además liberar los huecos ocupados por tareas que ya hayan finalizado.

Esquema

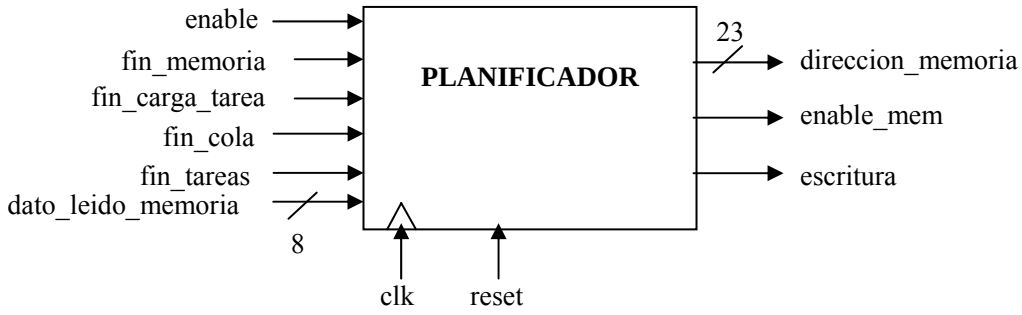


Figura 89: Planificador

Entradas/Salidas

Tabla 35: Entradas del planificador

Entradas	Descripción
clk	- Señal de reloj del módulo
reset	- Señal de reset asíncrono
enable	- Señal de capacitación
fin_memoria	- Señal que indica si la memoria ha terminado de realizar la operación solicitada
fin_carga_tarea	- Señal que indica si una tarea se ha terminado de cargar en la FPGA
dato_leido_memoria [7:0]	- Señal que contiene un byte leído de memoria
fin_cola [7:0]	- Señal que indica al planificador la posición del final de la cola de tareas
fin_tareas [3:0]	- Señal tendrá activos aquellos bits correspondientes con cada una de las cuatro tareas que haya terminado de ejecutarse

Tabla 36: Salidas del planificador

Salidas	Descripción
direccion_memoria [22:0]	- Señal que indica la dirección sobre la que se desea acceder a la memoria
enable_mem	- Señal que capacitará la memoria para poder realizar operaciones sobre ella
escritura	- Señal que indicará a la memoria si se desea realizar un acceso de escritura si está activa o de lectura si no
z_inicio_cola [7:0]	- Señal que contiene la posición de inicio de la cola de tareas
z_ejecutadas [3:0]	- Señal que tendrá activados aquellos bits en los que en aquella posición correspondiente de las cuatro que permite ejecutar tareas, haya una tarea en ejecución
z_id_tareas [31:0]	- Señal que contiene un byte con el identificador de cada una de las tareas que haya en ejecución. El byte correspondiente solo contendrá información válida en el caso de que su correspondiente bit de la señal *z_ejecutadas* se encuentre activado

Ruta de datos

La representación esquemática de la ruta de datos de este módulo es la siguiente:

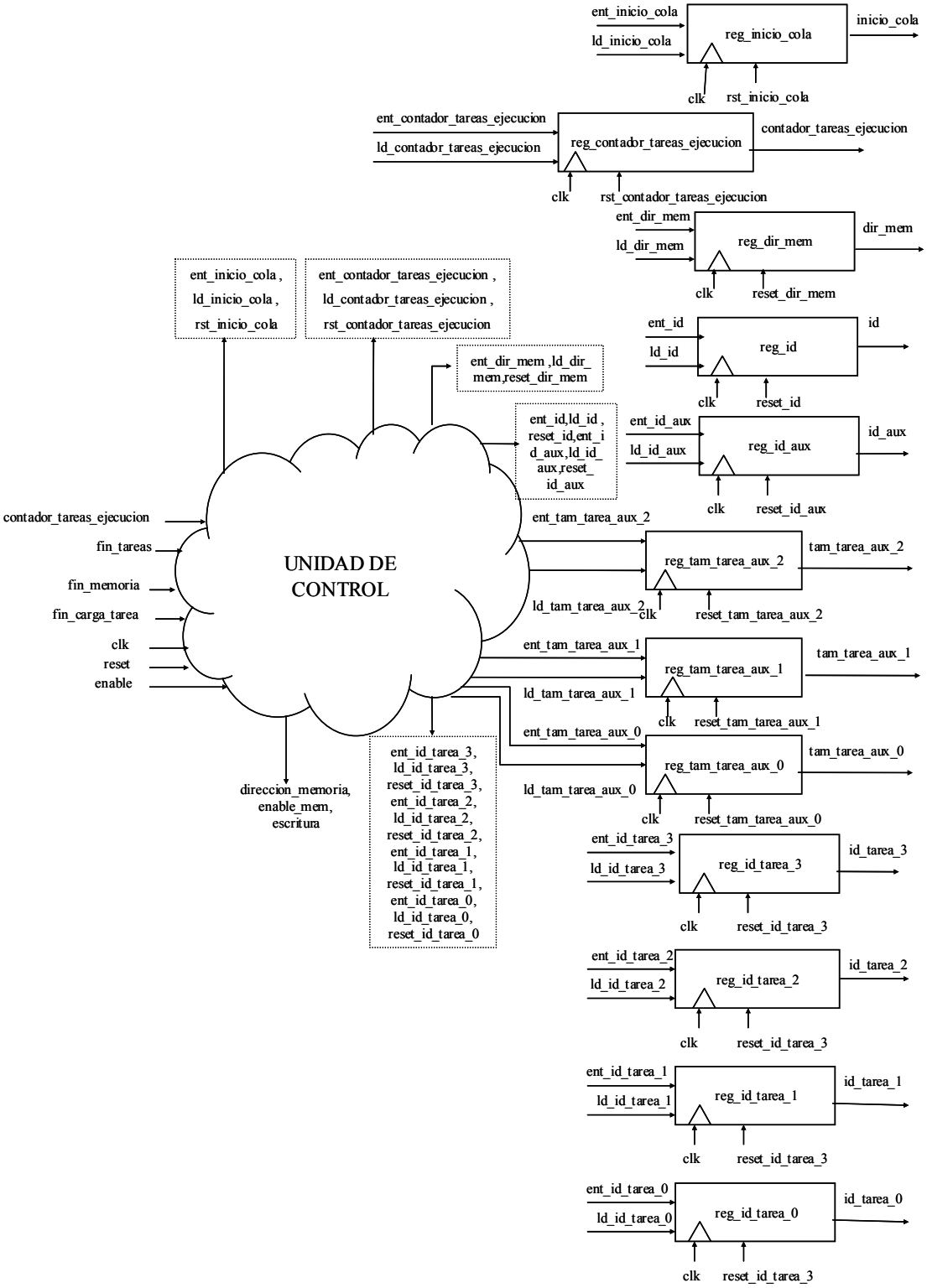


Figura 90: Ruta de datos del planificador

La ruta de datos del planificador está compuesta por registros los cuales se utilizan para almacenar todos los valores de las señales que le permiten gestionar las tareas.

Hay cuatro registros que contienen los identificadores de cada una de las tareas que haya en ejecución.

Hay un registro que almacena la posición de inicio de la cola que hay en memoria para planificar las tareas que se han lanzado a ejecución.

Existe otro registro contador que contiene el número total de tareas que hay en un momento determinado en memoria.

Además existen otra serie de registros auxiliares que se utilizan durante la búsqueda de una tarea existente en memoria antes de ser llevada a la cola.

Hemos hablado de la cola que hay en la memoria que planifica la ejecución de las tareas. A pesar de estar fuera de la FPGA, al ser una estructura de datos la tomaremos como parte de la ruta de datos por lo que vamos a hablar un poco ahora de la estructura que tiene así como la forma en la que se encuentran las tareas almacenadas en la memoria.

La cola es un array de 256 bytes cada uno de los cuales contiene como información el valor del identificador de la tarea que ocupa una determinada posición de la cola. Esta cola es circular se encuentra almacenada en memoria desde la posición 0 hasta la 255.

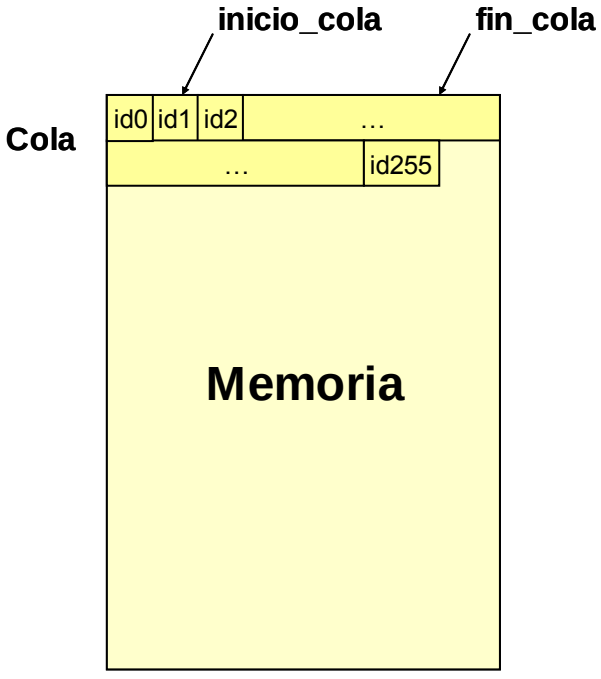
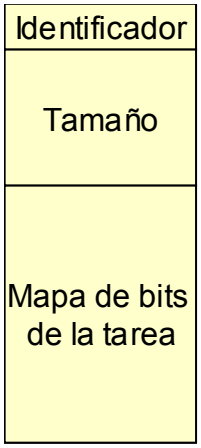


Figura 91: Estructura de la cola de tareas en la memoria

En el resto de la memoria se encuentran almacenadas las tareas. Estas tareas se almacenan en forma de lista enlazada a partir de la posición de memoria 256. A pesar de que la cola ocupa siempre 256 bytes, no tiene porqué haber tantas tareas cargadas en la memoria.

Cada uno de los nodos de la lista, además del mapa de bits de la propia tarea contiene la siguiente información:



- *Identificador*: Mediante este campo de un byte se identifica a una tarea y la diferencia de las demás. A la hora de lanzar una tarea a ejecución el usuario especificará al sistema el identificador de la tarea que desea ejecutar, y a partir de éste se recorrerá la lista enlazada buscando a la tarea correspondiente.

- *Tamaño*: Este campo de 3 bytes contiene el valor del tamaño que ocupa la tarea realmente. Es útil a la hora de recorrer la lista enlazada ya que se tiene en cuenta para pasar de una tarea a otra. El tamaño del campo es de 3 bytes ya que supusimos que 16 MB más que suficiente para almacenar el tipo de tareas que se iban a manejar.

- *Mapa de bits*: Al final y como último campo se encuentra el mapa de bits de la tarea.

Con todo esto podemos dar una visión general de cómo quedaría la memoria una vez se hubieran cargado las tareas en ella:

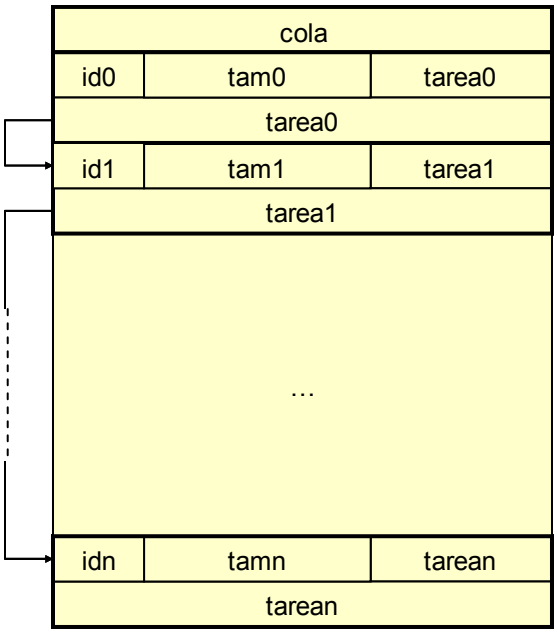


Figura 92: Estructura de la memoria

Unidad de control

La máquina de estados correspondiente con el controlador del módulo es la siguiente:

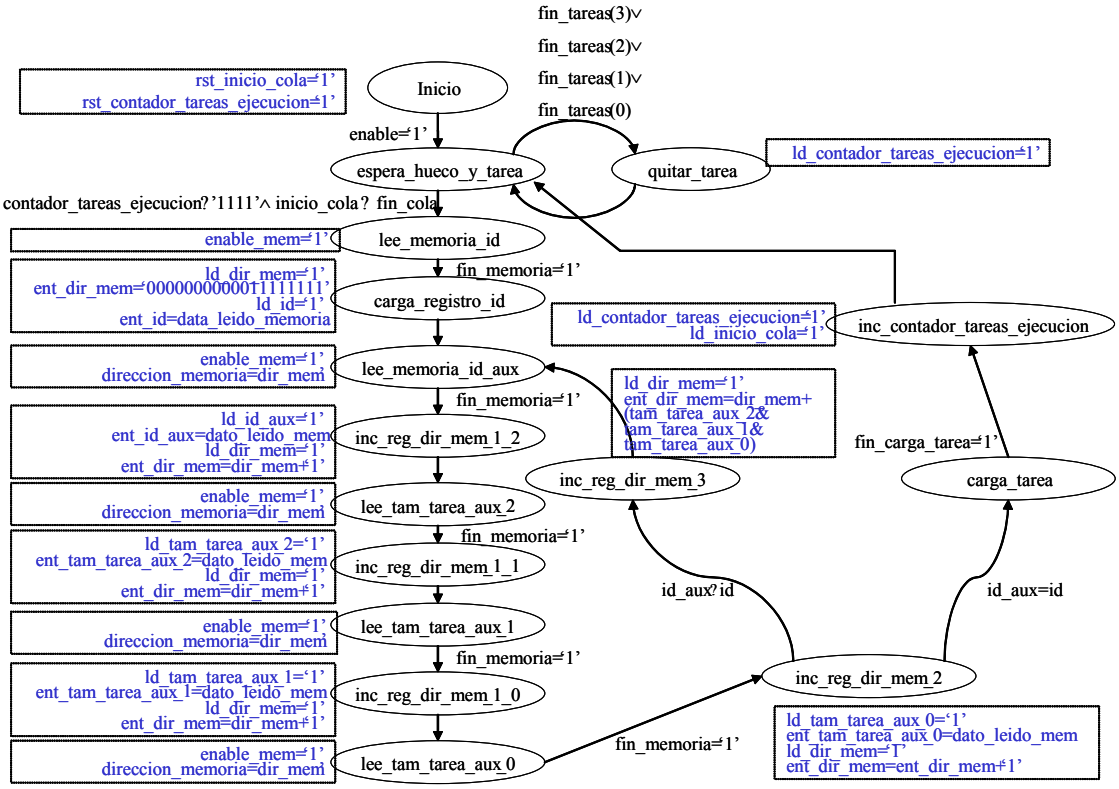


Figura 93: Diagrama de transición de estados del planificador

Todas las funcionalidades del planificador se reúnen en esta máquina de estados. Su funcionamiento es el siguiente: parte de un estado inicial en el cual permanece hasta que se capacita el componente, una vez hecho esto se pasa a un estado de espera en el cual puede o lanzar una tarea o liberar una tarea ya finalizada.

El planificador lanzará una tarea nueva cuando exista alguna disponible en la cola y además haya un hueco en el que la pueda colocar. Si esto es así, tomará el identificador de la primera tarea de la cola, recorrerá la lista enlazada de la memoria hasta que encuentre una tarea cuyo identificador coincida. Para recorrer la memoria sigue el siguiente proceso: toma el identificador de la primera tarea cuya posición es conocida ya que se encuentra en la posición 256. Una vez hecho esto comprueba si coincide con el que tiene que encontrar, si coincide entonces tomaría el mapa de bits de la tarea situado a partir de la posición 260 y cuyo tamaño es el indicado por los bytes 257-259. Si no coincide esta primera tarea entonces se toma el valor del tamaño de la tarea y se suma a 260 para acceder a la siguiente tarea. Se repite el proceso y en el caso de no encontrarse se suma el tamaño actual a la posición de inicio de la tarea actual más cuatro para apuntar al inicio de la tarea siguiente, así hasta encontrar la tarea deseada. Tras encontrar la tarea, ésta se carga sobre la FPGA, se actualizan los valores de los registros que controlan el número de tareas en ejecución, la lista con los identificadores y los huecos y se incrementa la posición de inicio de la cola eliminando la tarea cargada. Una vez termine esto el control vuelve al estado de espera.

Mientras el control se encuentre en el estado de espera, se procederá a liberar una tarea si se activa alguna de las señales que indican la finalización de las tareas de ejecución. La liberación de la tarea es un proceso lógico ya que no se modifica lo que hubiera en la FPGA ya que cuando se cargue una nueva tarea sobrescribirá a la anterior. Lo único que se actualiza para indicar que hay un nuevo hueco disponible es el registro que almacena un bit por cada tarea en ejecución. Entonces el bit de la tarea que acabe de finalizar se pone a cero y con eso el planificador ya sabe que ahí tiene un hueco para poder cargar una nueva tarea.

A partir de la máquina de estados obtenemos las siguientes tablas de transición de estados y de salidas:

Transición de estados

Tabla 37: Transición de estados del planificador

Estado	enable	contador_tareas_ejecucion	inicio_cola	fin_tareas	fin_memoria	id_aux	fin_carga_tarea	Estado Siguiente
inicio	1	-	–	-	-	-	-	espera_hueco_y_tarea
espera_hueco_y_tarea	-	≠"1111"	≠fin_cola	-	-	-	-	lee_memoria_id
espera_hueco_y_tarea	-	–	–	"1---"∨ "-1--"∨ "--1-"∨ "---1"	-	-	-	quitar_tarea
lee_memoria_id	-	-	–	-	1	-	-	carga_registro_id
carga_registro_id	-	-	–	-	-	-	-	lee_memoria_id_aux
lee_memoria_id_aux	-	-	-	-	1	-	-	inc_reg_dir_mem_1_2
inc_reg_dir_mem_1_2	-	-	-	-	-	-	-	lee_tam_tarea_aux_2
lee_tam_tarea_aux_2	-	-	-	-	1	-	-	inc_reg_dir_mem_1_1
inc_reg_dir_mem_1_1	-	-	-	-	-	-	-	lee_tam_tarea_aux_1
lee_tam_tarea_aux_1	-	-	-	-	1	-	-	inc_reg_dir_mem_1_0
inc_reg_dir_mem_1_0	-	-	-	-	-	-	-	lee_tam_tarea_aux_0
lee_tam_tarea_aux_0	-	-	-	-	1	-	-	inc_reg_dir_mem_2
inc_reg_dir_mem_2	-	-	-	-	-	id	-	carga_tarea
inc_reg_dir_mem_2	-	-	-	-	-	≠id	-	inc_reg_dir_mem_3
inc_reg_dir_mem_3	-	-	-	-	-	-	-	lee_memoria_id_aux
carga_tarea	-	-	-	-	-	-	1	inc_contador_ tareas_ejecucion
inc_contador_tareas_ejecucion	-	-	-	-	-	-	-	espera_hueco_y_tarea
quitar_tarea	-	-	-	-	-	-	-	espera_hueco_y_tarea

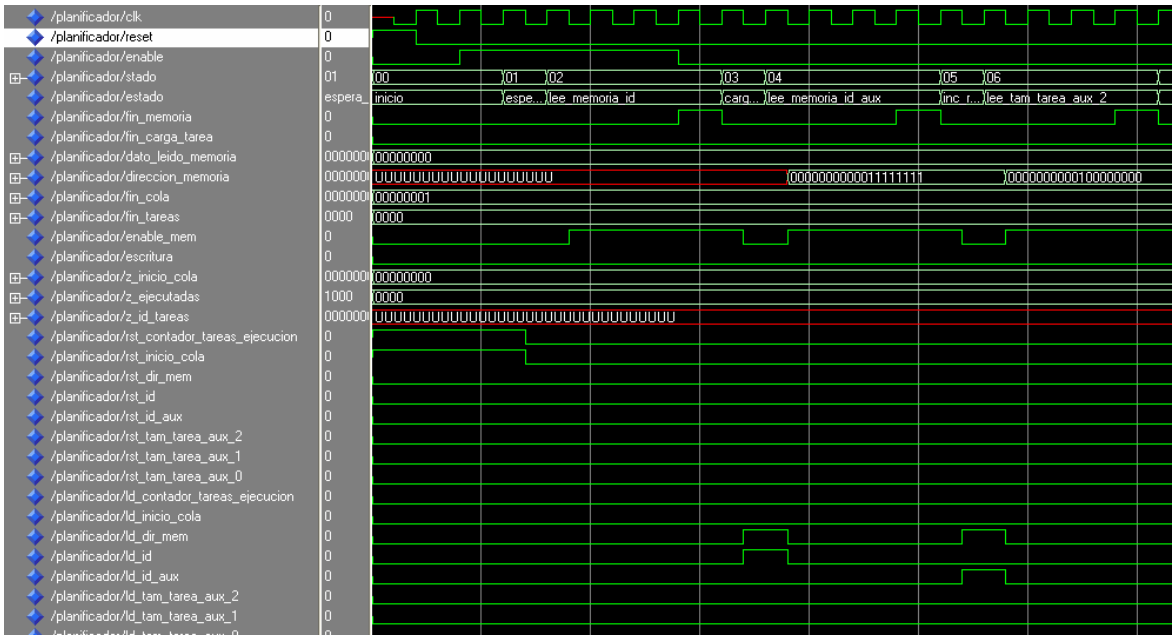
Salidas

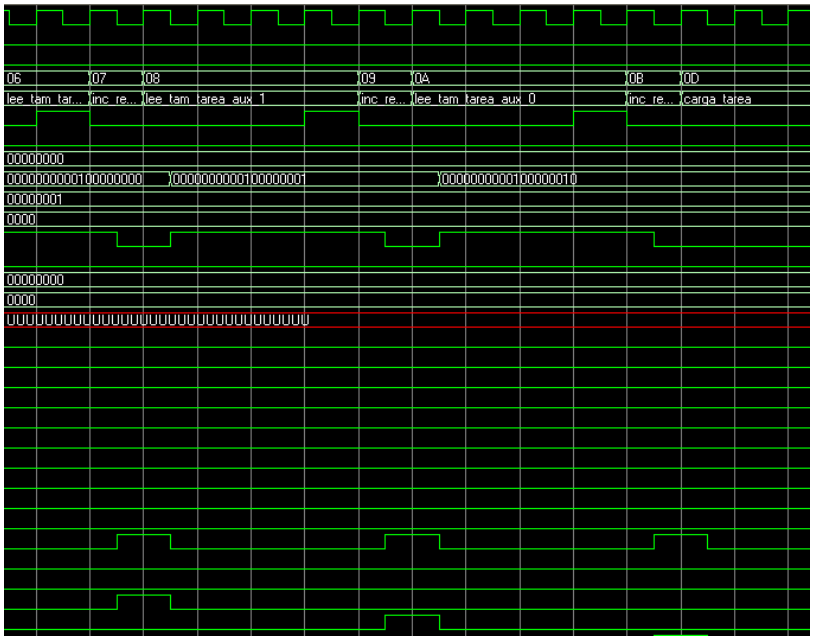
Tabla 38: Valores de las salidas del planificador

Estado	contador_tareas_ejec	fin_tareas	rst_inicio_cola	ent_inicio_cola	ld_inicio_cola	ent_contador_tareas_	ld_contador_tareas_e	rst_contador_tareas_c	ent_dir_mem	ld_dir_mem	ent_id	ld_id	ent_id_aux	ld_id_aux	ent_tam_tarea_aux_2	ld_tam_tarea_aux_2	ent_tam_tarea_aux_1	ld_tam_tarea_aux_1	ent_tam_tarea_aux_0	ld_tam_tarea_aux_0	ld_id_tarea_3	ent_id_tarea_3	ent_id_tarea_2	ld_id_tarea_2	ent_id_tarea_1	ld_id_tarea_1	ent_id_tarea_0	ld_id_tarea_0	enable_mem	escritura	direccion_memoria
inicio	-	-	1	-	0	-	0	1	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
espera_hueco_y_tarea	-	-	0	-	0	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
lee_memoria_id	-	-	0	-	0	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	1	0	-
carga_registro_id	-	-	0	-	0	-	0	0	"000000000 001111111"	1	dato_ leido_ memoria	1	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
lee_memoria_id_aux	-	-	0	-	0	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	1	dir_ mem
inc_reg_dir_mem_1_2	-	-	0	-	0	-	0	0	dir_mem+'1'	1	-	0	dato_ leido_ memoria	1	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
lee_tam_tarea_aux_2	-	-	0	-	0	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	1	0	dir_ mem
inc_reg_dir_mem_1_1	-	-	0	-	0	-	0	0	dir_mem+'1'	1	-	0	-	0	dato_ leido_ memoria	1	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
lee_tam_tarea_aux_1	-	-	0	-	0	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	1	0	dir_ mem
inc_reg_dir_mem_1_0	-	-	0	-	0	-	0	0	dir_mem+'1'	1	-	0	-	0	-	0	dato_ leido_ memoria	1	-	0	0	-	-	0	-	0	-	0	0	0	-
lee_tam_tarea_aux_0	-	-	0	-	0	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	dir_ mem
inc_reg_dir_mem_2	-	-	0	-	0	-	0	0	ent_dir_mem+'1'	1	-	0	-	0	-	0	-	0	dato_ leido_ memoria	1	0	-	-	0	-	0	-	0	0	0	-
inc_reg_dir_mem_3	-	-	0	-	0	-	0	0	dir_mem+(tam_tarea_aux_2 &tam_tarea_aux_1 &tam_tarea_aux_0)	1	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
carga_tarea	-	-	0	-	0	-	0	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
inc_contador_ tareas_ejecucion	"0---"	-	0	inicio_ cola+'1'	1	1&contador_tareas_ ejecucion(2 downto 0)	1	0	-	0	-	0	-	0	-	0	-	0	-	0	1	id	-	0	-	0	-	0	0	0	-
inc_contador_ tareas_ejecucion	"-0--"	-	0	inicio_ cola+'1'	1	contador_tareas_eje cucion(3)&'1'&conta dor_ tareas_ejecucion (1 downto 0)	1	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	id	1	-	0	-	0	0	0	-
inc_contador_ tareas_ejecucion	"--0-"	-	0	inicio_ cola+'1'	1	contador_tareas_eje cucion(3downto2)&'1'& contador_tareas _ejecucion(0)	1	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	id	1	-	0	0	0	-
inc_contador_ tareas_ejecucion	"---0"	-	0	inicio_ cola+'1'	1	contador_tareas_eje cucion(3downto1) & '1'	1	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	id	1	0	0	-
quitar_tarea	-	"1---"	0	-	0	0'&contador_ tareas_ejecucion (2 downto 0)	1	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
quitar_tarea	-	"-1--"	0	-	0	contador_tareas_ ejecucion(3)&'0'& contador_tareas_ ejecucion(1 downto 0)	1	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
quitar_tarea	-	"--1-"	0	-	0	contador_tareas_ ejecucion(3 downto 2) &'0'&contador_tareas _ejecucion(0)	1	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-
quitar_tarea	-	"---1"	0	-	0	contador_tareas_ ejecucion(3downto1) &'0'	1	0	-	0	-	0	-	0	-	0	-	0	-	0	0	-	-	0	-	0	-	0	0	0	-

Simulaciones

Ahora vamos a realizar algunas simulaciones para mostrar el funcionamiento del módulo.





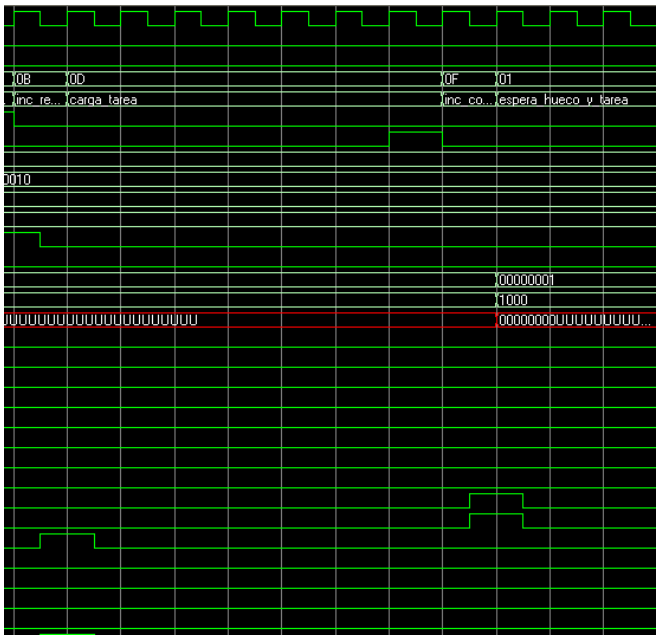


Figura 94: Simulación del planificador

Esta simulación muestra el proceso completo desde que se detecta que hay una tarea disponible en la cola y un hueco para poder ser lanzada. Desde ese punto se busca la tarea en memoria y se carga en la FPGA actualizando los punteros de la cola y el resto de registros de configuración.

5.4. Drivers de Entrada/Salida

A continuación vamos a comentar el funcionamiento de los drivers de entrada/salida que hemos utilizado.

Básicamente han sido dos y son los responsables de recoger la información del usuario y visualizar los resultados.

Existe un módulo encargado de la lectura de los datos introducidos por teclado por parte del usuario, este módulo es el *comprobador_tecla*.

El otro módulo es el que permite visualizar caracteres en una pantalla. Este controlador es el CHAR_MODE_SVGA_CTRL y es usado tanto para mostrar lo que el usuario ha escrito como para informarle de errores, del resultado de las operaciones realizadas, estado del sistema, etc.

5.4.1. Teclado

Descripción

Este módulo se encarga de detectar las pulsaciones de teclado. En el caso de producirse una pulsación, se activará la señal *pulsacion* y en el bus *tecla* se pondrá el código de teclado de la tecla pulsada.

Esquema

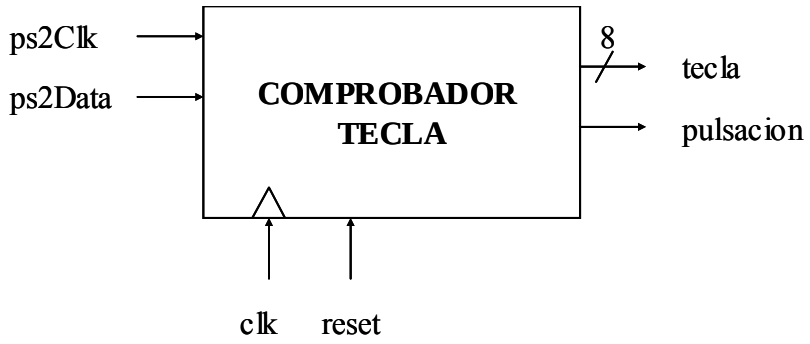


Figura 95: Comprobador de tecla

Entradas/Salidas

Tabla 39: Entradas del comprobador de tecla

Entradas	Descripción
reset	- Señal de reset del componente
clk	- Señal de reloj del controlador
ps2Clk	- Señal del reloj que viene del teclado
ps2Data	- Señal de datos que viene del teclado

Tabla 40: Salidas del comprobador de tecla

Salidas	Descripción
tecla [7:0]	- Señal que contiene el código de teclado de la última tecla pulsada
pulsación	- Señal que indica si se ha realizado o no una pulsación

Ruta de datos

La representación esquemática de la ruta de datos de este módulo es la siguiente:

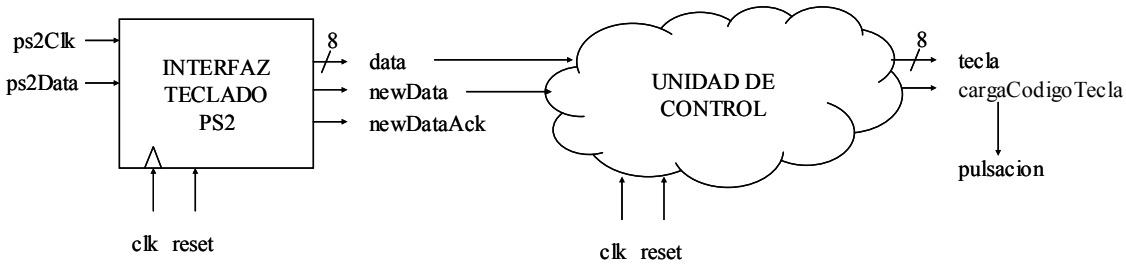


Figura 96: Ruta de datos del comprobador de tecla

El funcionamiento de este módulo se basa en el uso del interfaz de teclado PS/2 que contiene. Este módulo se encarga de unir los datos que llegan en serie por el puerto ps2Data y transformarlos a un código de teclado.

Una vez obtenido el código de teclado podemos conocer el valor ASCII de la tecla pulsada mediante el uso de un módulo auxiliar que hemos generado. Este módulo es el *conversor_tecla* y su esquema es el siguiente:

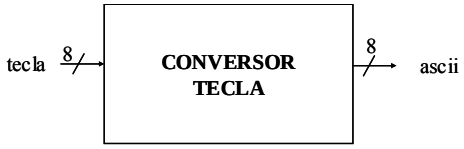
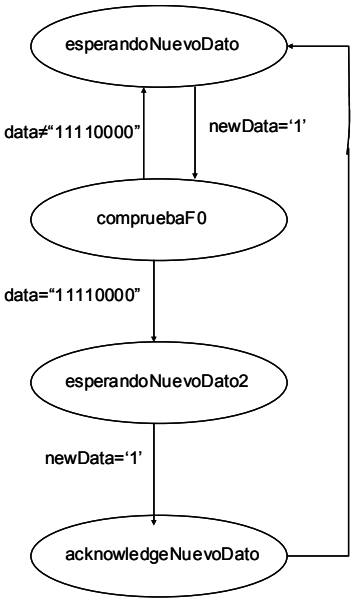


Figura 97: Conversor de tecla

Unidad de control

La máquina de estados correspondiente con el controlador del módulo es la siguiente:



La máquina de estados básicamente lo que hace es que en función del tipo de código que le llegue, si es un código de liberación de tecla no activará la señal de pulsación y sino si que la activa estableciendo el valor del código de teclado en la salida tecla.

Figura 98: Diagrama de transición de estados del comprobador de tecla

A partir de la máquina de estados obtenemos las siguientes tablas de transición de estados y de salidas:

Transición de estados

Tabla 41: Transición de estados del comprobador de tecla

ESTADO	newData	data	Estado Siguiente
esperandoNuevoDato	1	-	compruebaF0
compruebaF0	-	11110000	esperrandoNuevoDato2
compruebaF0	-	≠11110000	esperrandoNuevoDato
esperandoNuevoDato2	1	-	acknowledgeNuevoDato
acknowledgeNuevoDato	-	-	esperandoNuevoDato

Salidas

Tabla 42: Valores de las salidas del comprobador de tecla

ESTADO	newDataAck	cargaCodigoTecla	tecla
esperandoNuevoDato	0	0	-
compruebaF0	1	0	-
esperandoNuevoDato2	0	0	0
acknowledgeNuevoDato	1	1	data

5.4.2. VGA

Descripción

Este es el módulo permite controlar una pantalla VGA en modo texto. Es el que utilizamos para mostrar las órdenes que introduce el usuario y para mostrar también el resultado de la ejecución de las órdenes. Permite utilizar diferentes frecuencias de refresco a diferentes resoluciones modificando únicamente un archivo de propiedades. Además su control es realmente sencillo ya que únicamente hay que indicarle la posición y el código ASCII que se desea escribir en cada momento, encargándose él internamente de la gestión de la memoria de refresco.

Esquema

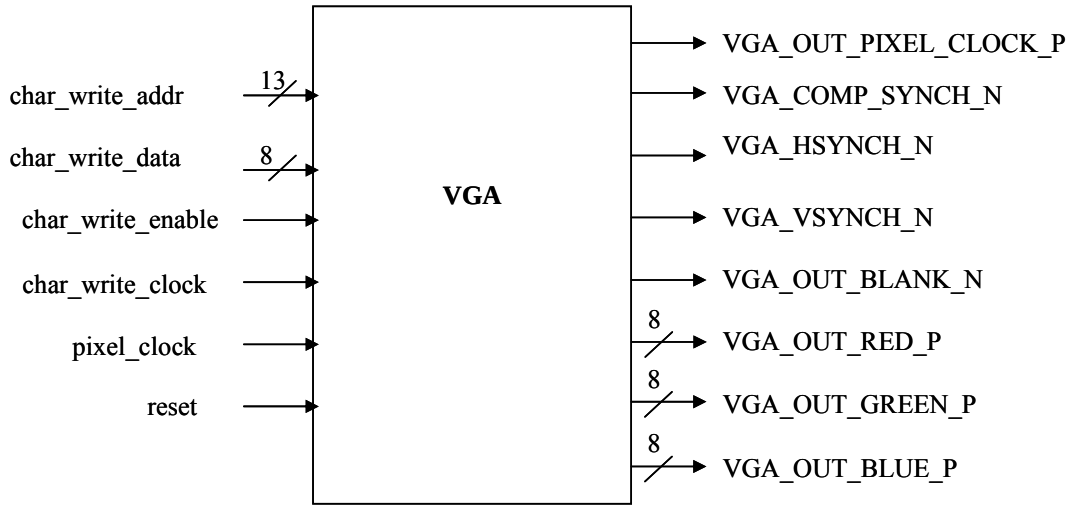


Figura 99: Controlador de VGA

Entradas/Salidas

Tabla 43: Entradas del controlador VGA

Entradas	Descripción
char_write_addr [12:0]	- Señal que contiene
char_write_data [7:0]	- Señal que tiene los datos que se desean escribir en la posición actual
char_write_enable	- Señal de capacitación de escritura. Para escribir un carácter nuevo debe estar activa esta señal
char_write_clock	- Señal de reloj de escritura
pixel_clock	- Señal de reloj de refresco de pixels
reset	- Señal de reset del componente

Tabla 44: Salidas del controlador VGA

Salidas	Descripción
VGA_OUT_PIXEL_CLOCK_P	- Señal de reloj de refresco del monitor
VGA_COMP_SYNCH_N	- Señal de comp_synch
VGA_HSYNCH_N	- Señal de hsynch
VGA_VSYNCH_N	- Señal de vsynch
VGA_OUT_BLANK_N	- Señal de blanking
VGA_OUT_RED_P [7:0]	- Señal de intensidad de color rojo
VGA_OUT_GREEN_P [7:0]	- Señal de intensidad de color verde
VGA_OUT_BLUE_P [7:0]	- Señal de intensidad de color azul

5.5. Generador de señales de reloj

Descripción

Mediante este módulo que contiene dos DCMs podemos generar una serie de señales de reloj a partir de los relojes propios de la FPGA. Entre los relojes que se generan se encuentra algunos cuya frecuencia es divisor de la frecuencia del reloj de entrada del sistema, otros tienen la misma frecuencia pero se encuentran desfasados 90, 180 o 270 grados con respecto a él.

Los relojes generados por este módulo los utilizamos principalmente para alimentar los distintos componentes del sistema tanto módulos creados por nosotros como el controlador de VGA, etc.

Esquema

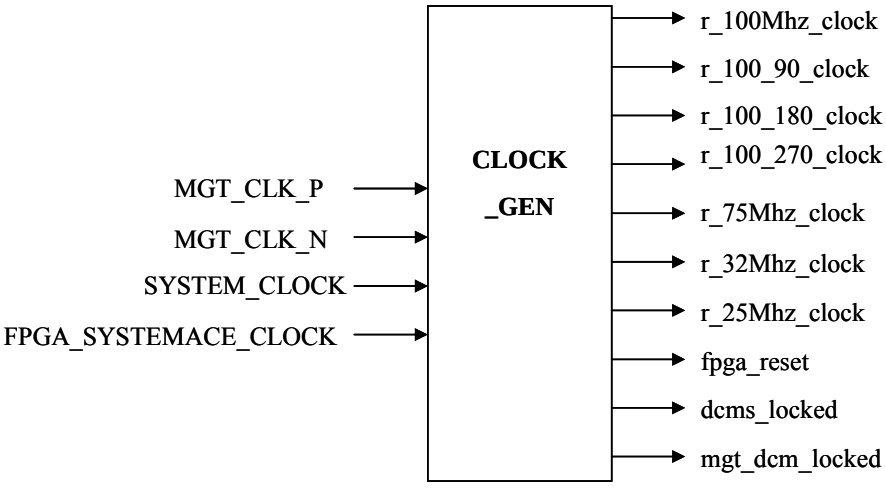


Figura 100: Generador de señales de reloj

Entradas/Salidas

Tabla 45: Entradas del generador de señales de reloj

Entradas	Descripción
MGT_CLK_N	- Señal de reloj generada por la FPGA de 75MHz. Esta y la siguiente son señales LVDS diferenciales
MGT_CLK_P	- Señal de reloj generada por la FPGA de 75MHz. Esta y la anterior son señales LVDS diferenciales
SYSTEM_CLOCK	- Señal de reloj generada por la FPGA de 100 MHz
FPGA_SYSTEMACE_CLOCK	- Señal de reloj generada por la FPGA de 32 MHz

Tabla 46: Salidas del generador de señales de reloj

Salidas	Descripción
r_100MHz_clock	- Señal de reloj de 100 MHz generada por el DCM
r_100_90_clock	- Señal de reloj de 100 MHz desfasada 90º generada por el DCM
r_100_180_clock	- Señal de reloj de 100 MHz desfasada 180º generada por el DCM
r_100_270_clock	- Señal de reloj de 100 MHz desfasada 270º generada por el DCM
r_75MHz_clock	- Señal de reloj de 75 MHz generada por el DCM
r_32MHz_clock	- Señal de reloj de 32 MHz generada por el DCM
r_25MHz_clock	- Señal de reloj de 25 MHz generada por el DCM
reset	- Señal de reset de los DCM
dcms_locked	- Señal que indica si los dos DCM que contiene el componente están ya preparados
mgt_dcm_locked	- Señal que indica si el DCM que utiliza el reloj de 75 MHz ya está preparado

5.6. Ejemplos de funcionamiento

Vamos a mostrar el funcionamiento del sistema mediante una serie de ejemplos de las órdenes permitidas.

En estos ejemplos vamos a realizar todos los pasos que se siguen desde que el usuario introduce la orden y ésta finaliza devolviendo el resultado.

Ejemplo de operación de memoria

Para realizar una operación de memoria como por ejemplo la escritura de un byte sobre una determinada dirección el usuario deberá introducir la orden:

CARGAR 00000000 4A

Si lo que se desea es cargar un conjunto de bytes consecutivos a partir de una determinada dirección se utiliza la siguiente instrucción:

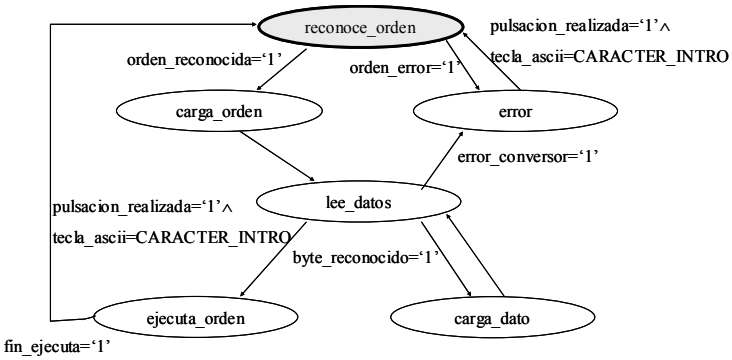
CARGARN 00000000 03 41 5B F6

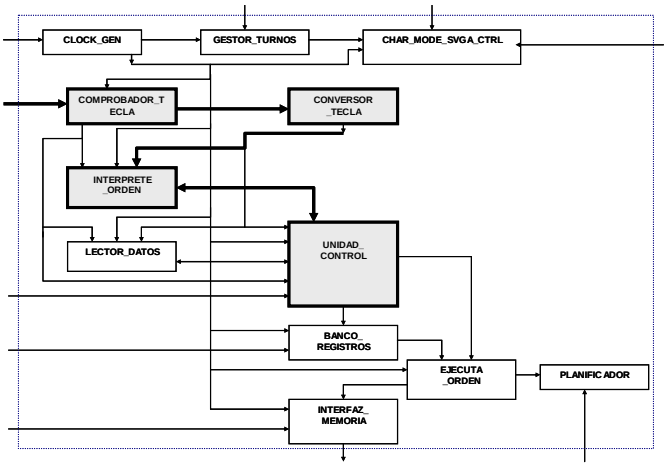
Para realizar la lectura de los datos contenidos en una posición de memoria hay que escribir:

LEER 00000000

Con esta orden se leerán los datos contenidos en la dirección 0 de la memoria y se mostrarán por pantalla.

Vamos a ver paso por paso el funcionamiento del sistema y los módulos que son utilizados en cada uno de los casos:

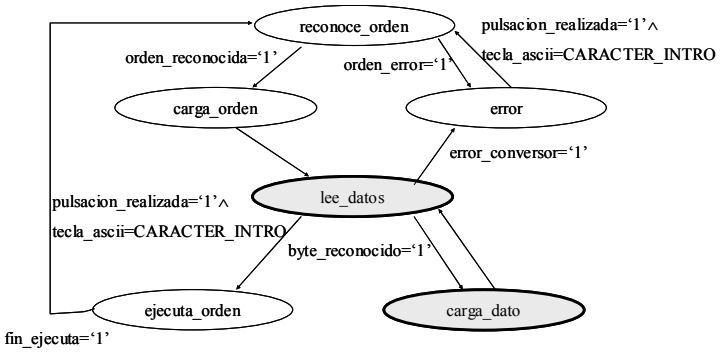


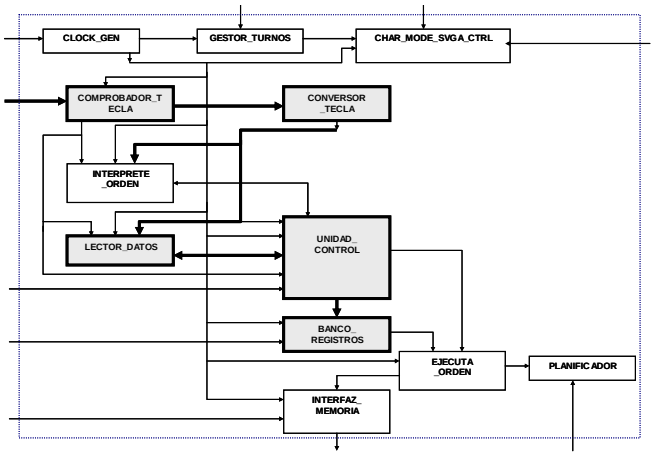


El primer paso para la realización de la lectura y de cualquier otra operación consiste primero en leer y reconocer la orden que se tiene que ejecutar.

Para ello el controlador hace uso de tres módulos según se muestra en el dibujo. Estos módulos son el que se encarga de leer cada una de las teclas pulsadas y obtener su código de teclado, este código de teclado es propagado hasta un módulo conversor a código ASCII. Una vez convertido los caracteres llegan hasta el módulo que interpreta las órdenes.

Una vez en este módulo se procederá al reconocimiento de la orden concreta informando del resultado a la unidad de control que cederá el testigo de ejecución al siguiente módulo.



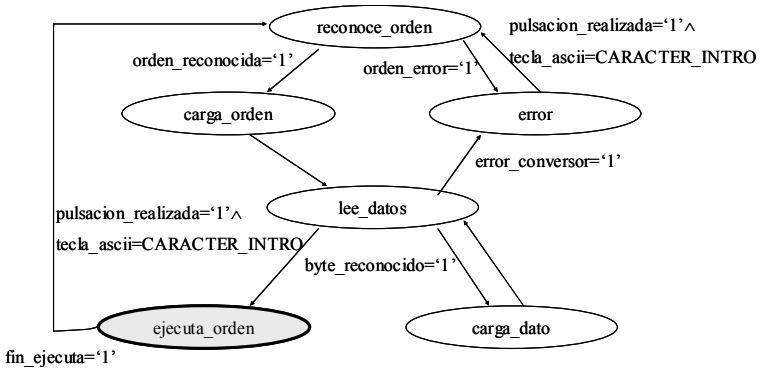


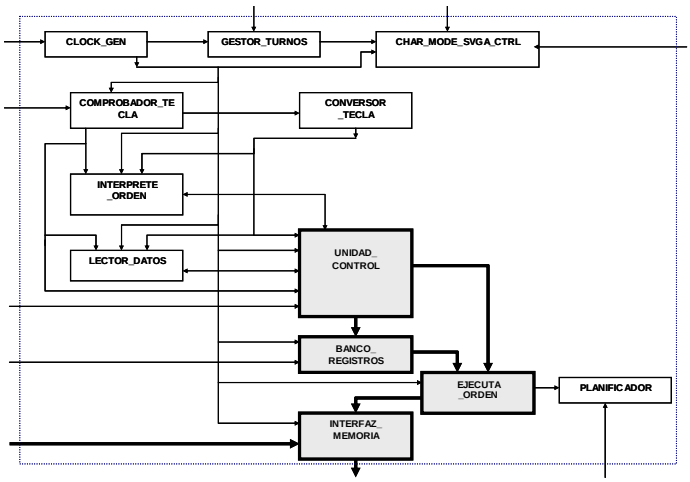
Previamente a la llegada al estado encargado de leer los datos, se ha pasado por un estado en el cual se ha almacenado el valor de la orden leída.

En el estado que lee los datos de la orden, la unidad de control capacita al igual que antes los módulos detectores y conversores de la tecla pulsada, un módulo que se encarga de gestionar la llegada de los datos, convirtiendo cada dos caracteres hexadecimales en un solo byte.

Una vez generado este byte es propagado hasta el banco de registros donde se van almacenando los bytes leidos en registros consecutivos.

Cuando el usuario termina de introducir los datos y pulsa la tecla intro el controlador pasa al estado en el cual se ejecuta la orden introducida por el usuario.





Por último se procede a ejecutar la orden introducida. Dicha ejecución es llevada a cabo por el módulo ejecutor y él se encarga de la comunicación con la memoria por medio del controlador al que suministra los datos necesarios para que se pueda llevar a cabo la operación.

Cuando finaliza la ejecución de la orden, la unidad de control vuelve al estado inicial de reconocimiento en el que está preparado para recibir a la siguiente orden.

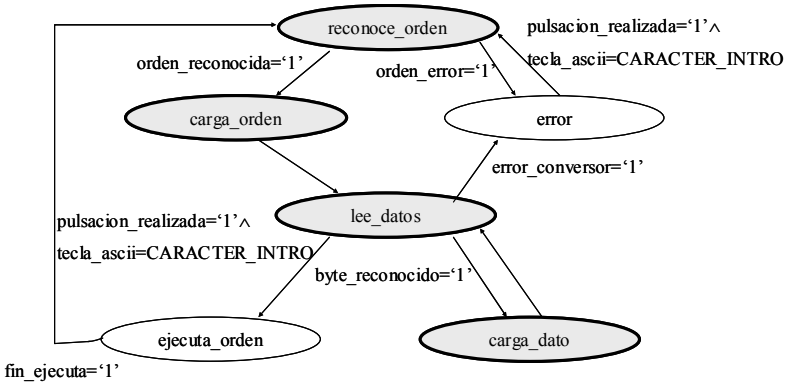
Con estos pasos que hemos mostrado hemos completado un ciclo completo de ejecución de una operación de memoria. El sistema mostrará en pantalla el resultado de la lectura o bien indicará si la escritura se ha realizado con éxito.

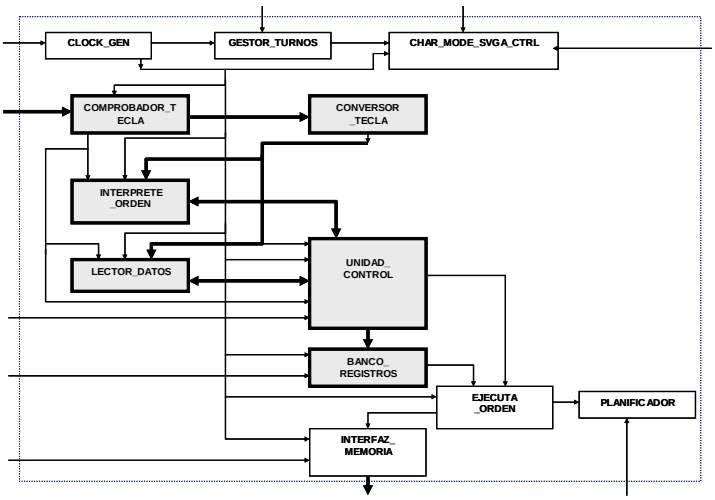
Ejemplo de lanzamiento de tareas

El usuario también puede realizar operaciones de lanzamiento de tareas sobre la FPGA. Para realizar un lanzamiento debe usar la siguiente orden indicando el identificador de la tarea que desea lanzar:

LANZAR 06

A continuación vamos a ver al igual que en el caso de las instrucciones de memoria, que pasos hay que seguir para poder completarla:

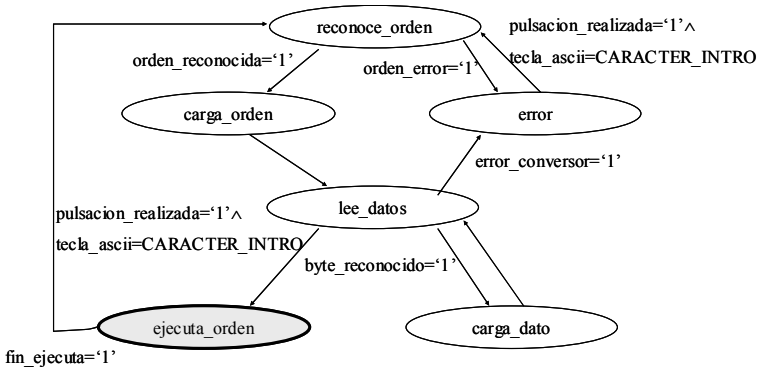




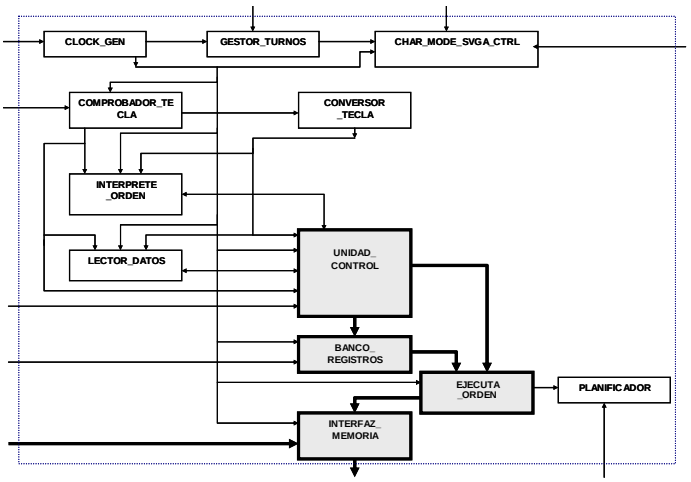
Los dos primeros pasos son comunes al caso de las operaciones de memoria, es decir, el sistema reconoce una orden a partir de las pulsaciones de teclado, la almacena y después pasa a leer sus datos.

Una vez hecho esto pasará a ejecutar la orden de lanzar y en este punto difiere de las operaciones de memoria.

En este momento la unidad de control pasa el testigo al ejecutor que se encarga procesar la orden.



En este estado de ejecución es cuando entra en funcionamiento el planificador, pero no el módulo en si, sino que el ejecutor es el encargado de llevar a la cola de tareas el identificador de la tarea introducida por el usuario.



El resto de funcionalidades del planificador se encuentran activas siempre. No es necesario que la unidad de control se encuentre en el estado de ejecutar sino que en cualquier momento puede comprobar que puede lanzar una nueva tarea a ejecución, como que puede eliminar una que esté cargada y que ya haya finalizado.

Con estos ejemplos podemos ver como el funcionamiento del sistema es bastante sencillo y además nos permite ver los componentes que realmente se utilizan durante la ejecución de una orden.

Capítulo 5: Instrucciones de uso

La forma de utilizar el sistema es realmente sencilla. El usuario sólo tendrá que cargarlo en la FPGA y comenzar a utilizarlo. Para ello deberá conectar la FPGA a un ordenador mediante el cable de conexión correspondiente.

Una vez hecho esto el usuario tendrá que utilizar algún tipo de herramienta, el iMPACT por ejemplo, que le permita cargar archivos de mapas de bits sobre la FPGA. Mediante esta herramienta cargará el archivo .BIT que representa al sistema.

Cuando ya esté cargado el sistema el usuario ya puede utilizarlo ya que no tiene que preocuparse de ningún tipo de configuración manual ni nada de eso.

En su utilización deberá introducir órdenes por teclado siguiendo la siguiente sintaxis:

- Para realizar la lectura de un byte de una dirección de memoria el usuario debe escribir:
    - *LEER DIRECCION*
        - Donde *DIRECCION* es una secuencia de 8 dígitos hexadecimales que representan una dirección de 32 bits.

  Un ejemplo de orden de lectura podría ser:

  LEER 00001234

- Para realizar la carga un byte a una dirección de memoria el usuario debe escribir:
    - *CARGAR DIRECCION DATO*
        - Donde *DATO* es el dato que se escribirá de un byte formado por 2 dígitos hexadecimales.
        - *DIRECCION* es la direccion formada por 8 dígitos hexadecimales en la que se escribe el dato.

  Un ejemplo de escritura podría ser:

  CARGAR 0012ABCD C6

- Para la realización de una carga múltiple de *n* bytes consecutivos a partir de una dirección de memoria el usuario debe escribir:
    - *CARGARN DIRECCION NUMERO_DATOS DATO1 DATO2 …DATOn*

- Donde *NUMERO_DATOS* es el número total de datos que el usuario quiere cargar en la memoria. Es un valor de un byte formado por 2 dígitos hexadecimales.

- *DATO1 DATO2 … DATOn* son los *n* datos que se escribirán en memoria. El valor de *n* viene dado por el valor establecido en *NUMERO_DATOS*. Cada dato va separado del resto por un espacio y es de un byte representado por 2 dígitos hexadecimales.

- *DIRECCION* es la dirección a partir de la cual se escriben los *n* datos de manera consecutiva.

Un ejemplo de escritura múltiple podría ser:

CARGARN 00112233 03 25 7F 88

- Para lanzar a ejecución una nueva tarea el usuario tiene que escribir:

  - *LANZAR IDENTIFICADOR_TAREA*

    - Donde *IDENTIFICADOR_TAREA* es un número de 8 bits representado por dos caracteres hexadecimales que sirve para distinguir las tareas que están almacenadas en memoria y cargar sólo aquella que desee el usuario.

Un ejemplo de lanzamiento de tarea podría ser:

LANZAR 27

El resultado devuelto por el sistema será un mensaje que indique *CORRECTO* si la orden se ha realizado bien, un mensaje que indica *ERROR* si se ha producido un error durante el proceso y únicamente en el caso de las lecturas se mostrará el dato leído por pantalla.

Si el usuario introdujese mal alguno de los datos necesarios para ejecutar una orden se produciría un error y el sistema le informaría. Por ejemplo si el usuario escribiese:

LEER 0000112K

Tras pulsar la última letra *K* el sistema pasaría a un estado de error informando al usuario ya que la letra *K* no corresponde con ninguno de los caracteres hexadecimales permitidos.

En la consola de usuario se habrá producido un salto de línea o no en función de si tras la letra *K* pulsó la tecla intro. Si la pulsó, entonces deberá pulsarla nuevamente para poder introducir una nueva orden, si no la pulsó, entonces deberá pulsarla para completar la captura de la orden, a pesar de estar incorrecta, y después volverla a pulsar al igual que en el caso anterior.

Si se introduce una orden mal el usuario puede seguir escribiendo hasta completarla pero desde que se detecte el error el sistema ya informaría de que lo que se ha escrito no se puede ejecutar por haberse producido un error.

Así son también ejemplos de ejecuciones incorrectas:

LEE 12345678, ya que *LEE* no es una orden reconocida.
CARGAR00000000 25, ya que no se introdujo espacio después del nombre de la orden.

En el caso de que en alguna de las órdenes no se hubiesen introducido datos, el sistema tomará cualquier posible basura que pueda haber en el banco de registros por lo que el comportamiento puede no estar determinado.

CARGAR 346790AA

Esta orden realizaría una carga sobre esa dirección pero no se le ha indicado el byte que tiene que cargar por lo que el sistema cargará la posible basura que encuentre.

Capítulo 6: Conclusiones

Con el desarrollo de este proyecto hemos aprendido mucho sobre el funcionamiento de las FPGAs, las ventajas que ofrecen y la forma de trabajar con este tipo de dispositivos.

Por el entorno en que se ha desarrollado el proyecto hemos profundizado en nuestros conocimientos de VHDL, aprendido a manejar nuevas herramientas de simulación y de carga de mapa de bits en la FPGA, ModelSim e iMPACT respectivamente, y un nuevo entorno de desarrollo como es el ISE.

A lo largo de la realización del proyecto nos hemos encontrado con varios problemas, entre los que podemos destacar el cambio de placas a la mitad del proyecto, la falta de recursos al tener que compartir tanto las placas como el lugar de trabajo que no siempre estaba disponible, el no poder realizar apenas trabajo en casa y el desconocimiento del manejo de la memoria DDR SDRAM. De todos estos problemas el más importante y que nos causó un retraso bastante significativo fue el del cambio de placa ya que supuso tener que adaptar todo lo que teníamos para que funcionara en la nueva placa, además de tener que aprender a manejar la memoria DDR SDRAM.

La fase de ejecución del proyecto que más tiempo nos ha llevado y que nos ha resultado más complicada ha sido, como ya hemos comentado, la de la utilización de la memoria DDR SDRAM debido a la complejidad del interfaz necesario para manejarla.

Nuestra meta del proyecto no ha podido ser alcanzada, por tanto el proyecto no está cerrado, y son muchas las futuras líneas de trabajo abiertas. Desde el punto de vista del diseño también se podrían realizar algunas mejoras, por lo que en un futuro podría ser ampliado.

Apéndices

A. Referencias

[R1] Documento: *Arquitectura Hardware para el seguimiento de múltiples imágenes basada en FPGA*. Enlace: http://www.aedie.org

[R2] Documento: *Introduction to FPGA Devices & Tools*. Enlace: http://atc.inf-cr.uclm.es

[R3] Artículo: *Xilinx ships PowerPC with RapidIO interface an lastest version of Xilinx system generator for DSP tool compilant with The MathWorks release 13*. Enlace: http://www.hoise.com

[R4] Documento: *Auto-Reconfiguración sobre FPGAs*. Enlace: http://www.escet.urjc.es

[R5] Documento: *XUPV2P User Guide*. Enlace: http://www.xilinx.org

[R6] Enlace: http://www.home.agilent.com

[R7] Documento: *Agilent Technologies 16700B and 16702B Logic Analysis Systems*. Enlace: http://cp.literature.agilent.com

[R8] Documento: *Double Data Rate (DDR) SDRAM*. Enlace: http://download.micron.com/pdf/datasheets/dram/ddr/512MBDDRx4x8x16.pdf

[R9] Documento: *Xilinx MIG 007 User Guide*. Enlace: http://www.xilinx.com

[R10] Documento: *Synthesizable 400 Mb/s DDR SDRAM Controller*. Enlace: http://www.eetasia.com/ARTICLES/2004DEC/A/2004DEC09_MPR_AN.PDF

[R11] Documento: *Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs*. Enlace: http://www.xilinx.com/bvdocs/appnotes/xapp462.pdf

[R12] Documento: *Virtex-II Digital Clock Manager*. Enlace: http://www.xilinx.com/products/virtex/techtopic/vtt010.pdf

[R13] Documento: *System ACE CompactFlash Solution*. Enlace: http://www-mtl.mit.edu/Courses/6.111/labkit/datasheets/XCCACE.pdf

B. Glosario de términos

ABEL (Advanced Boolean Expression Language): lenguaje de descripción de hardware y un conjunto de herramientas de diseño para programar dispositivos lógicos programables (PLDs).

Almacenamiento no volátil: capacidad de almacenar de forma que el contenido no se pierda cuando se interrumpa la corriente eléctrica que la alimenta.

ASCII (American Standard Code for Information Interchange): código de caracteres basado en el alfabeto latino tal como se usa en inglés moderno y en otras lenguas occidentales.

Bitstream: es una representación binaria de un diseño implementado de FPGA. El bitstream es generado por las herramientas de generación de Xilinx (BitGen y Makebits) y esta denotado por la extensión ".bit".

Bus: sistema que transfiere datos y que puede conectar mediante lógica varios periféricos utilizando el mismo conjunto de cables.

Byte: 8 bits.

Chip: es un circuito integrado en que se encuentran todos o casi todos los componentes para que un ordenador pueda realizar una función concreta.

Circuito integrado de aplicación específica (ASIC): circuito integrado hecho a la medida para un uso en particular.

CLB (configurable logic blocks): elementos funcionales para implementar la lógica de usuario, que a su vez están formado por elementos hardware programables.

Cola (FIFO): tipo de estructura de datos en la que el primer elemento que entra es el primero que sale.

Controlador de dispositivo (driver): programa informático que permite al sistema operativo interactuar con un periférico.

CPLD (complex programmable logic devices): dispositivo electrónico digital programable que esta formado por un cojunto de PLD.

DCM (digital clock manager): módulo que se encarga de gestionar las señales de reloj.

DIMM (Dual In-line Memory Module): módulo de memoria RAM utilizado en ordenadores personales. Se trata de un pequeño circuito impreso que contiene chips de memoria y se conecta directamente en ranuras de la placa base.

DLL (delay-locked loop): modulo que permite el desplazamiento de fase, espejo de relojes, multiplicar, dividir y sincronización de relojes externos.

Ethernet: tecnología de redes de computadoras de área local (LANs) basada en tramas de datos.
Fichero NCD (Native Circuit Description), el cual representa la descripción física del circuito de entrada pero aplicada al dispositivo especifico sobre el que se desea trabajar.

Fichero NGD (Native Generic Database), que describe el diseño lógico del circuito digital reducido a primitivas de Xilinx.

FPGA (Field programmable gate array): dispositivos electrónicos digitales programables de muy alta densidad.

Hardware dinámicamente reconfigurable: hardware que ofrece la posibilidad de darle cualquier tipo de funcionalidad deseada por el usuario.

IEEE (The Institute of Electrical and Electronics Engineers): asociación técnico-profesional mundial dedicada a la estandarización, entre otras cosas.

Interfaz: conjunto de comandos y métodos que permiten la intercomunicación de un programa con cualquier otro o elemento interno o externo.

IOB (input /output blocks): interfaz con los terminales del dispositivo.

ISE (Integrated Software Environment): herramienta de diseño de circuitos profesional que nos va a permitir entre otras funciones, la realización de esquemáticos y su posterior simulación, o mediante el uso de VHDL la implementación de circuitos digitales.

Kernel: parte fundamental o núcleo de un sistema operativo.

Leds (Ligh Emitting Diode): diodo emisor de luz, dispositivo semiconductor que emite luz de un solo color.

LUT (look up table): tabla de búsqueda que forma parte de los CLBs.

Mapa de bits: representación binaria en la cual un bit o conjunto de bits corresponde a alguna parte de un objeto, un bit puede ser la representación de cualquier cosa.

Memoria Compact Flash: memoria que permite que múltiples posiciones de memoria sean escritas o borradas en una misma operación de programación mediante impulsos eléctricos.

Memoria DDR (Double Data Rate): memoria de doble tasa de transferencia de datos que esta formada por memoria SDRAM.

Memoria DRAM (Dynamic Random Acces Memory): memoria volátil que necesita un refresco periódico para que se conserve el contenido.

Memoria EEPROM (electrically-erasable programmable read-only memory): memoria ROM que puede ser programada, borrada y reprogramado eléctricamente.

Memoria PROM (Programmable Read-Only Memory): memoria digital que puede ser programada una sola vez.

Memoria RAM (Random Access Memory) : memoria volátil en la que se puede tanto leer como escribir información que se utiliza normalmente como memoria temporal para almacenar resultados intermedios y datos similares no permanentes.

Memoria SDRAM (Synchronous Dynamic Random Access Memory): este tipo de memoria se conecta al reloj del sistema y está diseñada para ser capaz de leer o escribir a un ciclo de reloj por acceso.

Memoria SRAM (Static Random Access Memory): memoria volátil que no necesita refresco.

Multiplexores: dispositivo que selecciona como salida una de sus entradas en función de una señal de control

Pin: cada uno de los contactos terminales de un conector o componente electrónico.

Placas de trabajo o de prototipado: tarjeta de circuitos impresos sobre la que se dispone una FPGA, que sirve como medio de conexión entre el microprocesador, la FPGA, circuitos electrónicos de soporte, ranuras para conectar parte o toda la RAM del sistema, y para la conexión de dispositivos externos en general.

PS/2 (Personal System/2): forma de conectar dispositivos externos al PC, especialmente pensado para el ratón y el teclado.

Pushbutton: pulsador.

Readback: capacidad de realizar configuraciones dinámicas y de leer la configuración del dispositivo mientras funciona.

Reconfiguración parcial: técnica que permite cargar varias tareas sobre una misma FPGA, en cualquier momento y sin que tengan porqué interferir al funcionamiento del resto.

Semiconductor: sustancia que se comporta como conductor o como aislante dependiendo del campo eléctrico en el que se encuentre.

Skew: tiempo medio entre dos señales sincronizadas.

Switches: interruptor.

UCF (user constraint file): archivo en el que se establecen las restricciones del diseño lógico.

USB (Universal Serial Bus): interfaz de un estándar para conectar dispositivos a un ordenador personal.

Verilog: lenguaje de descripción de hardware usado para modelar sistemas electrónicos.

VGA (Video Graphics Array): una norma de visualización de gráficos para ordenadores.

VHDL (VHSIC Hardware Description Languaje): lenguaje usado para diseñar circuitos digitales.

C. Índice de figuras

Figura	Página
Figura 1: Esquema general del sistema	7
Figura 2: Ejecución de la carga inicial	8
Figura 3: Ejecución del lanzamiento de una tarea	9
Figura 4: Estructura interna de una FPGA	11
Figura 5: Estructura de un CLB	11
Figura 6: Estructura de una LUT	11
Figura 7: Estructura de interconexión de FPGA	12
Figura 8: Placa de extensión de Spartan 2	14
Figura 9: Arquitectura interna de Virtex	15
Figura 10: FPGA Spartan 2, placa de prototipado XSA-100	17
Figura 11: Foto Virtex II Multimedia	18
Figura 12: Foto Virtex II Pro sobre placa XUP	18
Figura 13: Conexión Virtex II con puertas	19
Figura 14: Foto de 16702B	20
Figura 15: Algunas características del 16702B	20
Figura 16: Muestra del comportamiento del sistema dado por la serie 16700	21
Figura 17: Muestra de un análisis de la forma de onda de señales	21
Figura 18: Esquema general de un módulo DDR SDRAM	27
Figura 19: Conteido del Mode Register	28
Figura 20: Orden de los elementos de una ráfaga	29
Figura 21: Contenido del Extended Mode Register	30
Figura 22: Lista de comandos	32
Figura 23: Activación de una fila de un banco de memoria	33
Figura 24: Comando de lectura	34
Figura 25: Ejecución de un comando de lectura	35
Figura 26: Ejecución de comandos de lectura consecutivos	35
Figura 27: Ejecución de un comando de lectura seguido de un comando BURST TERMINATE y de un comando PRECHARGE	36
Figura 28: Ejecución de un comando de lectura seguido de un comando de escritura	36
Figura 29: Comando de escritura	37
Figura 30: Ejecución de un comando de escritura	37
Figura 31: Ejecución de comandos de escritura consecutivos	38
Figura 32: Ejecución de un comando de escritura seguido de un comando de lectura	38
Figura 33: Ejecución de un comando de escritura seguido de un comando de precarga	39
Figura 34: Jerarquía de módulos del controlador de memoria	40
Figura 35: Interconexión de los módulos del controlador	41
Figura 36: Diagrama de estados del controlador	42
Figura 37: Esquema general del controlador de memoria	43
Figura 38: Inicialización de la memoria	46
Figura 39: Ejecución de una operación de escritura	47
Figura 40: Ejecución de una operación de lectura	48

Figura 41: Esquema general de un módulo DCM	50
Figura 42: Relojes de salida del DCM	50
Figura 43: Realimentación de relojes en un DCM	53
Figura 44: Componentes de un DCM	54
Figura 45: Pasos del proceso de diseño de circuitos digitales	57
Figura 46: Ventana principal del Project Navigator	57
Figura 47: Creación de nuevo proyecto (paso 1)	58
Figura 48: Creación de nuevo proyecto (paso 2)	59
Figura 49: Creación de nuevo proyecto (paso 3)	59
Figura 50: Creación de nuevo proyecto (paso 4)	60
Figura 51: Creación de nuevo proyecto (paso 5)	60
Figura 52: Proceso de síntesis	61
Figura 53: Proceso de implementación	62
Figura 54: Proceso de generación del .bit	62
Figura 55: Compilación de un fichero fuente VHDL	63
Figura 56: Simulación de un fichero VHDL	63
Figura 57: Árbol de componentes de la simulación	63
Figura 58: Resultado de la simulación	64
Figura 59: Ventana principal de la aplicación iMPACT	65
Figura 60: Ventana principal de la aplicación MIG	66
Figura 61: Modo editor de pines	67
Figura 62: Ejemplo de uso del MIG (paso 1)	68
Figura 63: Ejemplo de uso del MIG (paso 2)	69
Figura 64: Esquema general de nuestro sistema	85
Figura 65: Componente principal del sistema	86
Figura 66: Ruta de datos del sistema	90
Figura 67: Unidad de control	92
Figura 68: Diagrama de estados de la unidad de control	94
Figura 69: Simulación de la unidad de control	96
Figura 70: Intérprete de órdenes	97
Figura 71: Ruta de datos del intérprete de órdenes	99
Figura 72: Diagrama de transición de estados del intérprete órden	100
Figura 73: Simulación del intérprete de órdenes	104
Figura 74: Lector de datos	105
Figura 75: Ruta de datos del lector de datos	107
Figura 76: Banco de registros	107
Figura 77: Diagrama de transición de estados del lector de datos	108
Figura 78: Simulación del lector de datos	110
Figura 79: Ejecutor de órdenes	111
Figura 80: Ruta de datos del ejecutor de órdenes	113
Figura 81: Diagrama de transición de estados del ejecutor de órdenes	114
Figura 82: Simulación del ejecutor de órdenes (1)	118
Figura 83: Simulación del ejecutor de órdenes (2)	119
Figura 84: Interfaz de memoria	120
Figura 85: Ruta de datos del interfaz de memoria	123
Figura 86: Diagrama de transición de estados del interfaz de memoria (1)	124
Figura 87: Diagrama de transición de estados del interfaz de memoria (2)	126
Figura 88: Simulación del interfaz de memoria	129

Figura 89: Planificador	130
Figura 90: Ruta de datos del planificador	132
Figura 91: Estructura de la cola de tareas en la memoria	133
Figura 92: Estructura de la memoria	134
Figura 93: Diagrama de transición de estados del planificador	135
Figura 94: Simulación del planificador	139
Figura 95: Comprobador de tecla	141
Figura 96: Ruta de datos del comprobador de tecla	142
Figura 97: Conversor de tecla	142
Figura 98: Diagrama de transición de estados del comprobador de tecla	142
Figura 99: Controlador de VGA	144
Figura 100: Generador de señales de reloj	146

D. Índice de tablas

Tabla	Página
Tabla 1: Entradas de un módulo DDR SDRAM	27
Tabla 2: Entrada/Salida de un módulo DDR SDRAM	27
Tabla 3: Entrada/Salida del controlador de memoria	43
Tabla 4: Salidas del controlador a la memoria	44
Tabla 5: Entradas del controlador de memoria	44
Tabla 6: Salidas del controlador de memoria	45
Tabla 7: Entradas de un módulo DCM	51
Tabla 8: Salidas de un módulo DCM	51
Tabla 9: Atributos de un módulo DCM	52
Tabla 10: Entradas del componente principal	86
Tabla 11: Salidas del componente principal	87
Tabla 12: Señales de entrada/salida del componente principal	87
Tabla 13: Entradas de la unidad de control	93
Tabla 14: Salidas de la unidad de control	93
Tabla 15: Transición de estados de la unidad de control	95
Tabla 16: Valores de las salidas de la unidad de control	95
Tabla 17: Entradas del intérprete de órdenes	98
Tabla 18: Salidas del intérprete de órdenes	98
Tabla 19: Transición de estados del intérprete de órdenes	102
Tabla 20: Valores de las salidas del interprete de órdenes	103
Tabla 21: Entradas del lector de datos	106
Tabla 22: Salidas del lector de datos	106
Tabla 23: Transición de estados del lector de datos	109
Tabla 24: Valores de las salidas del lector de datos	109
Tabla 25: Entradas del ejecutor de órdenes	112
Tabla 26: Salidas del ejecutor de órdenes	112
Tabla 27: Transición de estados del ejecutor de órdenes	116
Tabla 28: Valores de las salidas del ejecutor de órdenes	117
Tabla 29: Entradas del interfaz de memoria	121
Tabla 30: Salidas del interfaz de memoria	121
Tabla 31: Señales de entrada/salida del interfaz de memoria	122
Tabla 32: Transición de estados del interfaz de memoria (1)	127
Tabla 33: Transición de estados del interfaz de memoria (2)	127
Tabla 34: Valores de las salidas del interfaz de memoria	128
Tabla 35: Entradas del planificador	131
Tabla 36: Salidas del planificador	131
Tabla 37: Transición de estados del planificador	136
Tabla 38: Valores de las salidas del planificador	137
Tabla 39: Entradas del comprobador de tecla	141
Tabla 40: Salidas del comprobador de tecla	141
Tabla 41: Transición de estados del comprobador tecla	143
Tabla 42: Valores de las salidas del comprobador tecla	143
Tabla 43: Entradas del controlador VGA	145
Tabla 44: Salidas del controlador VGA	145

Tabla 45: Entradas del generador de señales de reloj	147
Tabla 46: Salidas del generador de señales de reloj	147

D. Palabras clave

- FPGA
- VHDL
- Reconfiguración
- DDR
- Xilinx
- Virtex
- VGA
- DCM

F. Código del proyecto

Módulo principal (ordenes)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use constantes.all;
use work.parameter_64bit_00.all;

entity ordenes is
    port(
                l_clk90:out std_logic;
                l_user_data_valid:out std_logic;
                l_load_dato1:out std_logic;
                l_load_dato2:out std_logic;

                switch: in STD_LOGIC;
                reset: in STD_LOGIC;
                reset_memoria:in std_logic;

                MGT_CLK_P:in std_logic;--// 75MHz LVDS DIFFERENTIAL CLOCK FOR SATA
                MGT_CLK_N:in std_logic;
                SYSTEM_CLOCK_IN:in std_logic;--// 100MHz LVTTL SYSTEM CLOCK
                FPGA_SYSTEMACE_CLOCK:in std_logic;--// 32MHz SYSTEMACE CLOCK

                KBD_CLOCK:IN STD_LOGIC;
                KBD_DATA:IN STD_LOGIC;

                VGA_VSYNCH:out std_logic;
                VGA_HSYNCH:out std_logic;
                VGA_OUT_BLANK_Z:out std_logic;
                VGA_COMP_SYNCH:out std_logic;
                VGA_OUT_PIXEL_CLOCK:out std_logic;
                VGA_OUT_RED:out std_logic_vector(7 downto 0);
                VGA_OUT_GREEN:out std_logic_vector(7 downto 0);

 rst_dqs_div_in:in std_logic;
rst_dqs_div_out:out std_logic;
ddr_dqs               : inout std_logic_vector(7 downto 0);
 ddr_dq                : inout std_logic_vector(63 downto 0):= (OTHERS => 'Z');
 ddr_cke1               : out std_logic;
 ddr_cke2               : out std_logic;
 ddr_csb1               : out std_logic;
 ddr_csb2               : out std_logic;
 ddr_rasb              : out std_logic;
 ddr_casb              : out std_logic;
 ddr_web               : out std_logic;
 ddr_dm                : out std_logic_vector(((mask_width/2)-1) downto 0);
 ddr_ba                : out std_logic_vector((bank_address_p-1) downto 0);
 ddr_address           : out std_logic_vector((row_address_p-1) downto 0);
 ddr1_clk0             : out std_logic;
 ddr1_clk0b            : out std_logic;
 ddr1_clk1             : out std_logic;
 ddr1_clk1b            : out std_logic;
 ddr1_clk2             : out std_logic;
 ddr1_clk2b            : out std_logic
       );
end ordenes;

architecture ordenes_arch of ordenes is
attribute syn_keep: boolean;

component gestor_turnos
       generic(bits:natural:=2);
       port(
               reset:in std_logic;
               clk:in std_logic;
               enableDAC:out std_logic;
               turno:out std_logic_vector(bits-1 downto 0)
               );
end component;

component comprobador_tecla
port(
```

```
      rst: IN std_logic;                        --Reset del sistema
      clk: IN std_logic;                        --Reloj del sistema
      ps2Clk: IN std_logic;                     --Reloj del teclado
      ps2Data: IN std_logic;                    --Datos del teclado
      tecla: OUT std_logic_vector (7 downto 0); --Codigo de la tecla pulsada
      pulsacion: OUT std_logic                  --Indica que se ha pulsado una tecla
(activo a alta)
  );
end component;

component CLOCK_GEN
port(
       MGT_CLK_P:in std_logic;--// 75MHz LVDS DIFFERENTIAL CLOCK FOR SATA
       MGT_CLK_N:in std_logic;
       SYSTEM_CLOCK:in std_logic;--// 100MHz LVTTL SYSTEM CLOCK
       FPGA_SYSTEMACE_CLOCK:in std_logic;--// 32MHz SYSTEMACE CLOCK
       r_100MHz_clock:out std_logic;--// buffered SYSTEM_CLOCK
       r_100_90_clock:out std_logic;
       r_100_180_clock:out std_logic;
       r_100_270_clock:out std_logic;
       r_75MHz_clock:out std_logic;--// buffered MGT_CLK
       r_32MHz_clock:out std_logic;--// 1/3 * SYSTEM_CLOCK
       r_25MHz_clock:out std_logic;--// buffered FPGA_SYSTEMACE_CLOCK
       reset:out std_logic;--// reset asserted when DCMs are NOT LOCKED
       dcms_locked:out std_logic;
       mgt_dcm_locked:out std_logic
);
end component;

component ConversorTecla
    port (
        tecla: in STD_LOGIC_VECTOR (7 downto 0);
        ascii: out STD_LOGIC_VECTOR (7 downto 0)
    );
end component;

component  CHAR_MODE_SVGA_CTRL is
port(
       VGA_OUT_PIXEL_CLOCK_P:out std_logic; -- Reloj de control de pixeles
       VGA_COMP_SYNCH_N:out std_logic; -- Sincronizador
       VGA_OUT_BLANK_N:out std_logic; -- Blanking
       VGA_HSYNCH_N:out std_logic; -- Sincronizador horizontal
       VGA_VSYNCH_N:out std_logic; -- Sincronizador vertical
       VGA_OUT_RED_P:out std_logic_vector (7 downto 0); -- Salida para el color rojo
       VGA_OUT_GREEN_P:out std_logic_vector (7 downto 0); -- Salida para el color verde
       VGA_OUT_BLUE_P:out std_logic_vector (7 downto 0); -- Salida para el color azul
       char_write_addr:in std_logic_vector (12 downto 0); -- Entrada de la direccion de
escritura del caracter
       char_write_data:in std_logic_vector (7 downto 0); -- Codigo ASCII del caracter a
escribir
       char_write_enable:in std_logic; -- Senal de enable de la escritura de caracter
       char_write_clock:in std_logic; -- Reloj del componente
       pixel_clock:in std_logic; -- Reloj de control de pixeles
       reset:in std_logic -- Senal de reset
);
end component;

component interprete_orden
    Port ( tecla_ascii : in std_logic_vector(7 downto 0);
           reset : in std_logic;
           pulsacion_realizada : in std_logic;
           clk : in std_logic;
           tipo_orden : out std_logic_vector(7 downto 0);
           orden_reconocida : out std_logic;
           orden_error : out std_logic);
end component;

component lector_datos
    Port (     enable:in std_logic;
                          reset:in std_logic;
                      clk:in std_logic;
                      pulsacion_realizada : in std_logic;
           tecla_ascii : in std_logic_vector(7 downto 0);
           byte : out std_logic_vector(7 downto 0);
           byte_reconocido : out std_logic;
           error_conversor : out std_logic
                      );
```

```
end component;

component unidad_control
    port (
        reset: in STD_LOGIC;
        clk: in STD_LOGIC;
        orden_reconocida: in STD_LOGIC;
                orden_error:in std_logic;
        tecla_ascii: in STD_LOGIC_VECTOR(7 downto 0);
        pulsacion_realizada: in STD_LOGIC;
        error_conversor: in STD_LOGIC;
        fin_ejecuta: in STD_LOGIC;
        byte_reconocido: in STD_LOGIC;
        reset_interprete: out STD_LOGIC;
        reset_lector: out STD_LOGIC;
                enable_lector:out std_logic;
        load_orden: out STD_LOGIC;
        load_br: out STD_LOGIC;
        numero_registro: out STD_LOGIC_VECTOR (7 downto 0);
                tipo_numero_registro: out std_logic;
        enable_ejecuta: out STD_LOGIC;
                reset_ejecuta: out STD_LOGIC;
                error_unidad_control:out std_logic;
                stado:out std_logic_vector(2 downto 0)
    );
end component;

component banco_registros
    Port ( clk : in std_logic;
           reset : in std_logic;
           datos_in : in std_logic_vector(7 downto 0);
           numero_registro : in std_logic_vector(7 downto 0);
           load_reg : in std_logic;
           datos_out : out std_logic_vector(7 downto 0);
                       contenido:out std_logic_vector(39 downto 0)
                       );
end component;

component ejecuta_orden
    Port ( tipo_orden : in std_logic_vector(7 downto 0);
           enable_ejecuta : in std_logic;
           clk : in std_logic;
           reset : in std_logic;
                       dato_leido_registro: in std_logic_vector(7 downto 0);
--                     dato_leido_memoria:in std_logic_vector(7 downto 0);
                       direccion_memoria : out std_logic_vector(31 downto 0);
--                     escribe_memoria: out std_logic;
--                     lee_memoria:out std_logic;
                            escritura:out std_logic;
                       numero_registro: out std_logic_vector(7 downto 0);
                       fin_ejecuta:out std_logic;
                       load_dato_leido_memoria:out std_logic;
                       reg_direccion_out:out std_logic_vector(31 downto 0);
                       stado:out std_logic_vector(4 downto 0);
                            enable_mem:out std_logic;
                            fin_memoria:in std_logic;
                            fin_cola:out std_logic_vector(7 downto 0);
                            memoria_preparada:in std_logic
                       );
end component;

component interfaz_memoria
       port(
               l_user_data_valid:out std_logic;
               l_load_dato1:out std_logic;
               l_load_dato2:out std_logic;


               veces:out std_logic_vector(3 downto 0);

               memoria_preparada:out std_logic;
               dcms_locked:in std_logic;
               SYSTEM_CLOCK:in std_logic;
               reset : in std_logic;

               escritura : in std_logic;
               dir_mem : in std_logic_vector(22 downto 0);
```

```
                dato_in_mem1 : in std_logic_vector(127 downto 0);
                dato_in_mem2 : in std_logic_vector(127 downto 0);
                enable_mem : in std_logic;
                fin_memoria : out std_logic;
                dato_out_mem1 : out std_logic_vector(127 downto 0);
                dato_out_mem2 : out std_logic_vector(127 downto 0);

                dato_out_mem_antes : out std_logic_vector(127 downto 0);
                dato_out_mem_despues : out std_logic_vector(127 downto 0);

rst_dqs_div_in:in std_logic;--no especificado (no usado)
rst_dqs_div_out:out std_logic;--no especificado (no usado)
 ddr_dqs               : inout std_logic_vector(7 downto 0);
 ddr_dq                : inout std_logic_vector(63 downto 0):= (OTHERS => 'Z');
 ddr_cke1               : out std_logic;
 ddr_cke2               : out std_logic;
--NO SE SABE CUAL ES
ddr_csb1               : out std_logic;
ddr_csb2               : out std_logic;
 ddr_rasb              : out std_logic;
 ddr_casb              : out std_logic;
 ddr_web               : out std_logic;
 ddr_dm                : out std_logic_vector(((mask_width/2)-1) downto 0);
 ddr_ba                : out std_logic_vector((bank_address_p-1) downto 0);
 ddr_address           : out std_logic_vector((row_address_p-1) downto 0);
 ddr_clk0             : out std_logic;
 ddr_clk0b            : out std_logic;
 ddr_clk1             : out std_logic;
 ddr_clk1b            : out std_logic;
 ddr_clk2             : out std_logic;
 ddr_clk2b            : out std_logic;

                stado:out std_logic_vector(4 downto 0);
                stadold:out std_logic_vector(1 downto 0)
        );
end component;

component buffer_escritura
        generic(longitud_buffer:natural:=8;longitud_direccion:natural:=12);
        port(
                clk:in std_logic;
                reset:in std_logic;
                enable:in std_logic;
                datos:in std_logic_vector(longitud_buffer-1 downto 0);
                suma_posicion:in std_logic_vector(longitud_direccion-1 downto 0);
                byte_actual:out std_logic_vector(7 downto 0);
                posicion_actual:out std_logic_vector(longitud_direccion-1 downto 0);
                auto_reset:in std_logic
        );
end component;


component hex2ascii
        port(
                        hex:in std_logic_vector(3 downto 0);
                        ascii:out std_logic_vector(7 downto 0)
        );
end component;

component planificador
        port (
                clk:in std_logic;
                reset: in std_logic;
                enable:in std_logic;

                fin_memoria: in std_logic;
                fin_carga_tarea: in std_logic;
                dato_leido_memoria:in std_logic_vector(7 downto 0);

                direccion_memoria:out std_logic_vector(18 downto 0);
                fin_cola:in std_logic_vector(7 downto 0);
                fin_tareas:in std_logic_vector(3 downto 0);
                enable_mem:out std_logic;
                escritura:out std_logic;
                z_inicio_cola:out std_logic_vector(7 downto 0);
                z_ejecutadas:out std_logic_vector(3 downto 0);
                stado:out std_logic_vector(4 downto 0);
```

```
                z_id_tareas:out std_logic_vector(31 downto 0)
        );
end component;

component interfaz_compactflash
        port(
                clk:in std_logic;
                reset:in std_logic;
                dato_in:in std_logic_vector(15 downto 0);
                direccion:in std_logic_vector(6 downto 0);
                dato_out:out std_logic_vector(15 downto 0);
                escritura:in std_logic;
                fin_compactflash:out std_logic;
                enable:in std_logic;
                stado:out std_logic_vector(2 downto 0);

                MPD_P:inout std_logic_vector(15 downto 0);
                MPA_P:out std_logic_vector(6 downto 0);
                MPCE_N:out std_logic;
                MPWE_N:out std_logic;
                MPOE_N:out std_logic;
                MPIRQ_P:in std_logic;
                MPBRDY_P:in std_logic
        );
end component;

component IBUFG
  port ( I : in std_logic;
         O : out std_logic);
 end component;

--Reset del sistema
signal nreset:std_logic;
--Reloj del sistema
signal reloj_sistema:std_logic;

--Senyal que indica si se ha pulsado una tecla
signal pulsacion_realizada:std_logic;
--Senyal que contiene el codigo de teclado de la tecla pulsada
signal tecla_pulsada:std_logic_vector(7 downto 0);
--Senyal que contiene el codigo ascii de la tecla pulsada
signal tecla_ascii:std_logic_vector(7 downto 0);

--Componentes de video
signal arrayred: std_logic_vector (7 downto 0);
signal arraygreen: std_logic_vector (7 downto 0);
signal arrayblue: std_logic_vector (7 downto 0);
--Pixel clock de la VGA
signal pixelclock:std_logic;
--Direccion de escritura del caracter
signal direccion_escritura:std_logic_vector(12 downto 0);
--Datos que se van a escribir correspondientes con el codigo ascii del caracter que se
desa escribir
signal datos_escritura:std_logic_vector(7 downto 0);
--Capacitacion de escritura de la VGA
signal enableDAC:std_logic;


signal r_100MHz_clock:std_logic;--// buffered SYSTEM_CLOCK
signal r_100_90_clock:std_logic;
signal r_100_180_clock:std_logic;
signal r_100_270_clock:std_logic;
signal r_75MHz_clock:std_logic;--// buffered MGT_CLK
signal r_32MHz_clock:std_logic;--// 1/3 * SYSTEM_CLOCK
signal r_25MHz_clock:std_logic;--// buffered FPGA_SYSTEMACE_CLOCK
signal fpga_reset:std_logic;--// reset asserted when DCMs are NOT LOCKED
signal dcms_locked:std_logic;

signal prueba1,prueba2,prueba3:std_logic_vector(7 downto 0);
signal turno:std_logic_vector(7 downto 0);

signal tipo_orden:std_logic_vector(7 downto 0);
signal orden_reconocida:std_logic;
signal orden_error:std_logic;
signal reset_componentes:std_logic;

signal byte_dato_leido:std_logic_vector(7 downto 0);
```

```
signal byte_dato_reconocido,error_dato_leido:std_logic;

signal
reset_interprete,reset_lector,fin_ejecuta,enable_ejecuta,reset_ejecuta,load_br,load_orde
n,tipo_numero_registro:std_logic;
signal numero_registro_uc,numero_registro_eorden,numero_registro:std_logic_vector(7
downto 0);

signal dato_registro_out:std_logic_vector(7 downto 0);
signal dato_memoria_in,dato_memoria_out:std_logic_vector(31 downto 0);

signal direccion_memoria:std_logic_vector(31 downto 0);
signal escribe_memoria:std_logic;
signal vcc,gnd:std_logic;
signal dato_out_mem,dato_in_mem:std_logic_vector(7 downto 0);

signal direccion_orden:std_logic_vector(12 downto 0);
signal stado_ld:std_logic_vector(1 downto 0);
signal stado_uc:std_logic_vector(2 downto 0);
signal enable_ld:std_logic;

signal registro_orden:std_logic_vector(7 downto 0);

signal linea_muestra_orden:std_logic_vector(7 downto 0);

signal stado_eorden:std_logic_vector(4 downto 0);

signal load_dato_leido_memoria:std_logic;

signal dato_leido_memoria:std_logic_vector(7 downto 0);

signal contenido_regs:std_logic_vector(39 downto 0);

signal prueba:std_logic_vector(7 downto 0);

signal lee_memoria:std_logic;

signal escribe_memoria2,lee_memoria2:std_logic;
signal direccion_memoria2:std_logic_vector(31 downto 0);
signal dato_memoria_out2,dato_memoria_in2:std_logic_vector(31 downto 0);

type estados2 is
(inicial,prepara_carga,carga0,carga1,carga2,carga3,prepara_lee,lee0,lee1,lee2,lee3);
signal estado2:estados2;

signal reg_direccion_out:std_logic_vector(31 downto 0);

signal enable_buffer_escritura1,reset_buffer_escritura1:std_logic;
signal enable_buffer_escritura2,reset_buffer_escritura2:std_logic;
signal enable_buffer_escritura3,reset_buffer_escritura3:std_logic;
signal enable_buffer_escritura4,reset_buffer_escritura4:std_logic;
signal enable_buffer_escritura5,reset_buffer_escritura5:std_logic;
signal enable_buffer_escritura6,reset_buffer_escritura6:std_logic;
signal datos_buffer_escritura1:std_logic_vector(63 downto 0);
signal datos_buffer_escritura2:std_logic_vector(39 downto 0);
signal suma_posicion1:std_logic_vector(12 downto 0);
signal suma_posicion2:std_logic_vector(12 downto 0);
signal suma_posicion3:std_logic_vector(12 downto 0);
signal suma_posicion4:std_logic_vector(12 downto 0);
signal suma_posicion5:std_logic_vector(12 downto 0);
signal suma_posicion6:std_logic_vector(12 downto 0);
signal byte_actual_buffer_escritura1:std_logic_vector(7 downto 0);
signal byte_actual_buffer_escritura2:std_logic_vector(7 downto 0);
signal byte_actual_buffer_escritura3:std_logic_vector(7 downto 0);
signal byte_actual_buffer_escritura4:std_logic_vector(7 downto 0);
signal byte_actual_buffer_escritura5:std_logic_vector(7 downto 0);
signal byte_actual_buffer_escritura6:std_logic_vector(7 downto 0);
signal posicion_actual_buffer_escritura1:std_logic_vector(12 downto 0);
signal posicion_actual_buffer_escritura2:std_logic_vector(12 downto 0);
signal posicion_actual_buffer_escritura3:std_logic_vector(12 downto 0);
signal posicion_actual_buffer_escritura4:std_logic_vector(12 downto 0);
signal posicion_actual_buffer_escritura5:std_logic_vector(12 downto 0);
signal posicion_actual_buffer_escritura6:std_logic_vector(12 downto 0);

signal digito_leido1,digito_leido2:std_logic_vector(7 downto 0);

signal error_unidad_control:std_logic;
```

```
signal escritura,fin_memoria,enable_mem:std_logic;

signal fin_cola:std_logic_vector(7 downto 0);

signal enable_planificador,fin_carga_tarea:std_logic;
signal fin_tareas,z_ejecutadas:std_logic_vector(3 downto 0);
signal z_inicio_cola:std_logic_vector(7 downto 0);
signal stado_planif:std_logic_vector(4 downto 0);
signal direccion_memoria_planif:std_logic_vector(18 downto 0);
signal enable_mem_planif,fin_memoria_planif,escritura_planif:std_logic;
signal z_id_tareas:std_logic_vector(31 downto 0);
signal
enable_compactflash,escritura_compactflash,load_dato_leido_compactflash:std_logic;
signal direccion_compactflash:std_logic_vector(6 downto 0);

signal dato_in_compactflash:std_logic_vector(15 downto 0);
signal dato_out_compactflash:std_logic_vector(15 downto 0);
signal fin_compactflash:std_logic;
signal dato_leido_compactflash:std_logic_vector(15 downto 0);
signal MPCE_N_aux,MPWE_N_aux,MPOE_N_aux:std_logic;
signal stado_compactflash:std_logic_vector(2 downto 0);

signal dato_in_mem1,dato_in_mem2,dato_out_mem1,dato_out_mem2:std_logic_vector(127 downto
0);
signal stado_im:std_logic_vector(4 downto 0);
signal stadold_im:std_logic_vector(1 downto 0);

signal MPD_P:std_logic_vector(15 downto 0);
signal MPA_P:std_logic_vector(6 downto 0);
signal MPCE_N:std_logic;
signal MPWE_N:std_logic;
signal MPOE_N:std_logic;
signal MPIRQ_P:std_logic;
signal MPBRDY_P:std_logic;
signal memory_54mhz:std_logic;

signal ddr_csb,ddr_cke:std_logic;

signal reloj_ddr:std_logic;
--signal dcms_locked2:std_logic;
signal dcms_locked3:std_logic;
signal dato_out_mem_antes,dato_out_mem_despues:std_logic_vector(127 downto 0);

signal memoria_preparada:std_logic;
signal veces:std_logic_vector(3 downto 0);

signal SYSTEM_CLOCK:std_logic;

begin



        ibufsystemclock:      IBUFG port map(
                                      I => SYSTEM_CLOCK_IN,
                                      O => SYSTEM_CLOCK
                              );

        pixelclock<=r_32MHz_clock;
        reloj_sistema<=pixelclock;

        nreset<=not reset;

        --Modulo conversor de codigo de teclado a ASCII
        convTecla:ConversorTecla port map (tecla_pulsada,tecla_ascii);

        --Modulo controlador del teclado
        Interfaz_de_entrada:comprobador_tecla port map
(nreset,reloj_sistema,KBD_CLOCK,KBD_DATA,tecla_pulsada,pulsacion_realizada);

        --Modulo generador de senyales de reloj
        interfaz_clock: CLOCK_GEN port
map(MGT_CLK_P,MGT_CLK_N,SYSTEM_CLOCK,FPGA_SYSTEMACE_CLOCK,r_100MHz_clock,r_100_90_clock,
r_100_180_clock,r_100_270_clock,r_75MHz_clock,r_32MHz_clock,r_25MHz_clock,fpga_reset,dcm
s_locked,dcms_locked3);
```

```
        --Modulo controlador de la VGA
        interfaz_DAC: CHAR_MODE_SVGA_CTRL port map
(VGA_OUT_PIXEL_CLOCK,VGA_COMP_SYNCH,VGA_OUT_BLANK_Z,VGA_HSYNCH,VGA_VSYNCH,prueba1,prueba
2,prueba3,direccion_escritura,datos_escritura,enableDAC,reloj_sistema,pixelclock,reset);

        --Modulo encargado de gestionar los turnos de escritura en pantalla
        gt: gestor_turnos generic map(8) port
map(reset_componentes,reloj_sistema,enableDAC,turno);

        --Salidas de las componentes de video
        VGA_OUT_RED<=arrayred;
        VGA_OUT_GREEN<=arraygreen;

        arrayred<=prueba1;
        arraygreen<=prueba2;
        arrayblue<=prueba3;

        as2hex1: hex2ascii port map(dato_leido_memoria(7 downto 4),digito_leido1);
        as2hex2: hex2ascii port map(dato_leido_memoria(3 downto 0),digito_leido2);


        reset_componentes<=switch;


        iorden: interprete_orden port
map(tecla_ascii,reset_interprete,pulsacion_realizada,

        reloj_sistema,tipo_orden,orden_reconocida,orden_error);

        ldatos: lector_datos port
map(enable_ld,reset_lector,reloj_sistema,pulsacion_realizada,tecla_ascii,byte_dato_leido
,byte_dato_reconocido,error_dato_leido);

        ucontrol: unidad_control port
map(reset_componentes,reloj_sistema,orden_reconocida,orden_error,tecla_ascii,

        pulsacion_realizada,error_dato_leido,fin_ejecuta,

        byte_dato_reconocido,reset_interprete,reset_lector,enable_ld,

        load_orden,load_br,numero_registro_uc,tipo_numero_registro,enable_ejecuta,reset_e
jecuta,error_unidad_control,stado_uc);

        registro_orden<=
                                                tipo_orden when load_orden='1' else
                                                "00000011" when
error_unidad_control='1' else
                                                registro_orden;

        eorden: ejecuta_orden port
map(registro_orden,enable_ejecuta,reloj_sistema,reset_ejecuta,dato_registro_out,direccio
n_memoria,escritura,numero_registro_eorden,fin_ejecuta,load_dato_leido_memoria,reg_direc
cion_out,stado_eorden,enable_mem,fin_memoria,fin_cola,memoria_preparada);

--      dato_leido_memoria<=dato_memoria_out(31 downto 24) when
load_dato_leido_memoria='1' else dato_leido_memoria;
        dato_leido_memoria<=dato_out_mem when load_dato_leido_memoria='1' else
dato_leido_memoria;

        bregs: banco_registros port
map(reloj_sistema,reset_componentes,byte_dato_leido,numero_registro,load_br,dato_registr
o_out,contenido_regs);

        --contenido_regs(31 downto 0)<=reg_direccion_out;

        numero_registro<=numero_registro_uc when tipo_numero_registro='0' else
                                            numero_registro_eorden;

        dato_in_mem<=dato_registro_out;


        vcc<='1';
        gnd<='0';
```

```
--      dato_in_mem1<=dato_in_mem &
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000";


        dato_in_mem1<=dato_in_mem & dato_in_mem & dato_in_mem & dato_in_mem & dato_in_mem
& dato_in_mem & dato_in_mem & dato_in_mem &
"01000001010000010100000101000001010000010100000101000001";
        dato_in_mem2<="01000010010000100100001001000010010000100100001001000010"
& "01000011010000110100001101000011010000110100001101000011";

process(pixelclock)
begin
        if(pixelclock='1' and pixelclock'event) then
                reloj_ddr<=not reloj_ddr;
        end if;
end process;

l_clk90<=r_100_90_clock;


intMem: interfaz_memoria port map(
        l_user_data_valid,
        l_load_dato1,
        l_load_dato2,

        veces,
        memoria_preparada,
        dcms_locked3,
        SYSTEM_CLOCK,
                reset_memoria,

                escritura,
                direccion_memoria(22 downto 0),
                dato_in_mem1,
                dato_in_mem2,
                enable_mem,
                fin_memoria,
                dato_out_mem1,
                dato_out_mem2,

                dato_out_mem_antes,
                dato_out_mem_despues,

rst_dqs_div_in,
rst_dqs_div_out,
 ddr_dqs,
 ddr_dq,
 ddr_cke1,
 ddr_cke2,
 ddr_csb1,
 ddr_csb2,
 ddr_rasb,
 ddr_casb,
 ddr_web,
 ddr_dm,
 ddr_ba,
 ddr_address,
 ddr1_clk0,
 ddr1_clk0b,
 ddr1_clk1,
 ddr1_clk1b,
 ddr1_clk2,
 ddr1_clk2b,

                stado_im,
                stadold_im
        );

        dato_in_compactflash<=dato_registro_out&"00000000";

        dato_leido_compactflash<=dato_out_compactflash when
load_dato_leido_compactflash='1' else dato_leido_compactflash;

intcompactflash:
        interfaz_compactflash port map(
                reloj_sistema,
```

```
        reset_componentes,
        dato_in_compactflash,
        direccion_compactflash,
        dato_out_compactflash,
        escritura_compactflash,
        fin_compactflash,
        enable_compactflash,
        stado_compactflash,

        MPD_P,
        MPA_P,
        MPCE_N_aux,
        MPWE_N_aux,
        MPOE_N_aux,
        MPIRQ_P,
        MPBRDY_P
);

MPCE_N<=MPCE_N_aux;
MPWE_N<=MPWE_N_aux;
MPOE_N<=MPOE_N_aux;

enable_planificador<='1';
fin_carga_tarea<='1';
fin_tareas<="0000";
fin_memoria_planif<='1';
escritura_planif<='0';

planif: planificador port map(
        reloj_sistema,
        reset_componentes,
        enable_planificador,
        fin_memoria_planif,
        fin_carga_tarea,
        dato_leido_memoria,
        direccion_memoria_planif,
        fin_cola,
        fin_tareas,
        enable_mem_planif,
        escritura_planif,
        z_inicio_cola,
        z_ejecutadas,
        stado_planif,
        z_id_tareas
);


process(reset_componentes,reloj_sistema)
begin
        if(reset='1') then
                estado2<=inicial;
        elsif(reloj_sistema='1' and reloj_sistema'event) then
                case estado2 is
                        when inicial=>
                                estado2<=prepara_carga;
                                escribe_memoria2<='0';
                                lee_memoria2<='0';

direccion_memoria2<="00000000000000000000000000000100";

dato_memoria_in2<="00010001001000100011001101000100";
                        when prepara_carga=>
                                estado2<=carga0;
                                escribe_memoria2<='1';
                                lee_memoria2<='0';

direccion_memoria2<="00000000000000000000000000000100";

dato_memoria_in2<="00010001001000100011001101000100";
                        when carga0=>
                                estado2<=carga1;
                                escribe_memoria2<='1';
                                lee_memoria2<='0';

direccion_memoria2<="00000000000000000000000000000100";

dato_memoria_in2<="00010001001000100011001101000100";
```

```
                        when carga1=>
                                estado2<=carga2;
                                escribe_memoria2<='1';
                                lee_memoria2<='0';

        direccion_memoria2<="00000000000000000000000000000100";

        dato_memoria_in2<="00010001001000100011001101000100";
                        when carga2=>
                                estado2<=carga3;
                                escribe_memoria2<='1';
                                lee_memoria2<='0';

        direccion_memoria2<="00000000000000000000000000000100";

        dato_memoria_in2<="00010001001000100011001101000100";
                        when carga3=>
                                estado2<=prepara_lee;
                                escribe_memoria2<='1';
                                lee_memoria2<='0';

        direccion_memoria2<="00000000000000000000000000000100";
                        when prepara_lee=>
                                estado2<=lee0;
                                escribe_memoria2<='0';
                                lee_memoria2<='1';

        direccion_memoria2<="00000000000000000000000000000100";
                        when lee0=>
                                estado2<=lee1;
                                escribe_memoria2<='0';
                                lee_memoria2<='1';

        direccion_memoria2<="00000000000000000000000000000100";
                        when lee1=>
                                estado2<=lee2;
                                escribe_memoria2<='0';
                                lee_memoria2<='1';

        direccion_memoria2<="00000000000000000000000000000100";
                        when lee2=>
                                estado2<=lee3;
                                escribe_memoria2<='0';
                                lee_memoria2<='1';

        direccion_memoria2<="00000000000000000000000000000100";
                        when lee3=>
                                estado2<=lee3;
                                escribe_memoria2<='0';
                                lee_memoria2<='1';

        direccion_memoria2<="00000000000000000000000000000100";
                                --dato_leido_memoria<=dato_memoria_out2(31 downto
24);
                                --contenido_regs(31 downto 0)<=dato_memoria_out2;
                end case;
        end if;
        end process;


        process(reset_componentes,reloj_sistema,pulsacion_realizada)
        begin
                if(reset_componentes='1') then
                        direccion_orden<="0001001011000";
                        linea_muestra_orden<="00000111";
                elsif(reloj_sistema='1' and reloj_sistema'event) then
                        if(pulsacion_realizada='1') then
                                if(tecla_ascii=CARACTER_INTRO) then

        direccion_orden<=linea_muestra_orden*"0000001100100";
                                        linea_muestra_orden<=linea_muestra_orden+'1';
                                else
                                        direccion_orden<=direccion_orden+'1';
                                end if;
                        end if;
                end if;
        end process;
```

```
end ordenes_arch;
```

Unidad de control

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use constantes.all;

entity unidad_control is
    port (
        reset: in STD_LOGIC;
        clk: in STD_LOGIC;
        orden_reconocida: in STD_LOGIC;
        orden_error:in std_logic;
        tecla_ascii: in STD_LOGIC_VECTOR(7 downto 0);
        pulsacion_realizada: in STD_LOGIC;
        error_conversor: in STD_LOGIC;
        fin_ejecuta: in STD_LOGIC;
        byte_reconocido: in STD_LOGIC;
        reset_interprete: out STD_LOGIC;
        reset_lector: out STD_LOGIC;
        enable_lector:out std_logic;
        load_orden: out STD_LOGIC;
        load_br: out STD_LOGIC;
        numero_registro: out STD_LOGIC_VECTOR (7 downto 0);
        tipo_numero_registro: out std_logic;
        enable_ejecuta: out STD_LOGIC;
        reset_ejecuta:out std_logic;
        error_unidad_control:out std_logic;
        stado:out std_logic_vector(2 downto 0)
    );
end unidad_control;

architecture unidad_control_arch of unidad_control is

type estados is(reconoce_orden,carga_orden,lee_datos,carga_dato,ejecuta_orden,error);
signal estado:estados;
signal numero_registro_aux: std_logic_vector (7 downto 0);

begin

        process(clk,reset,pulsacion_realizada,byte_reconocido,fin_ejecuta)
        begin
                if(reset='1') then
                        estado<=reconoce_orden;
                elsif(clk='1' and clk'event) then
                        case estado is
                                when reconoce_orden=>
                                        if(orden_error='1') then
                                                estado<=error;
                                        elsif(orden_reconocida='1') then
                                                estado<=carga_orden;
                                        end if;
                                when carga_orden=>
                                        estado<=lee_datos;
                                when lee_datos=>
                                        if(error_conversor='1') then
                                                estado<=error;
                                        elsif(byte_reconocido='1') then
                                                estado<=carga_dato;
                                        elsif(pulsacion_realizada='1' and
                                                        tecla_ascii=CARACTER_INTRO) then
                                                estado<=ejecuta_orden;
                                        end if;
                                when carga_dato=>
                                        estado<=lee_datos;
                                when ejecuta_orden=>
                                        if(fin_ejecuta='1') then
                                                estado<=reconoce_orden;
                                        end if;
                                when error=>
                                        if(pulsacion_realizada='1' and
                                                        tecla_ascii=CARACTER_INTRO) then
                                                estado<=reconoce_orden;
```

```
                                end if;
                        end case;
                end if;
        end process;

        process (clk,estado,tecla_ascii,pulsacion_realizada,byte_reconocido,fin_ejecuta)
        begin
                if(clk='1' and clk'event) then
                        reset_interprete<='0';
                        reset_lector<='0';
                        load_orden<='0';
                        load_br<='0';
                        enable_ejecuta<='0';
                        tipo_numero_registro<='0';
                        enable_lector<='0';
                        error_unidad_control<='0';
                        case estado is
                                when reconoce_orden=>
                                        reset_lector<='1';
                                        error_unidad_control<='0';
                                when carga_orden=>
                                        load_orden<='1';
                                        enable_lector<='1';
                                        numero_registro_aux<=(others=>'0');
                                        error_unidad_control<='0';
                                when lee_datos=>
                                        if(pulsacion_realizada='1' and
                                                        tecla_ascii=CARACTER_INTRO) then
                                                reset_ejecuta<='0';
                                                enable_ejecuta<='1';
                                                tipo_numero_registro<='1';
                                        else
                                                reset_ejecuta<='1';
                                                enable_ejecuta<='0';
                                                tipo_numero_registro<='0';
                                        end if;
                                        error_unidad_control<='0';
                                when carga_dato=>
                                        load_br<='1';
                                        numero_registro_aux<=numero_registro_aux+'1';
                                when ejecuta_orden=>
                                        tipo_numero_registro<='1';
                                        reset_interprete<='1';
                                        error_unidad_control<='0';
                                when error=>
                                        reset_interprete<='1';
                                        error_unidad_control<='1';
                        end case;
                        numero_registro<=numero_registro_aux;
                end if;
        end process;

        stado<="000" when estado=reconoce_orden else
                                "001" when estado=lee_datos else
                                "010" when estado=ejecuta_orden else
                                "111";
end unidad_control_arch;
```

Intérprete de órdenes

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

entity interprete_orden is
    Port ( tecla_ascii : in std_logic_vector(7 downto 0);
           reset : in std_logic;
           pulsacion_realizada : in std_logic;
           clk : in std_logic;
           tipo_orden : out std_logic_vector(7 downto 0);
           orden_reconocida : out std_logic;
           orden_error : out std_logic);
end interprete_orden;

architecture arch of interprete_orden is

--DECLARACION DE COMPONENTES

component registro
       generic(longitud:natural:=5);
       port(
              ent:in std_logic_vector(longitud-1 downto 0);
              clk,carga_r,reset:in std_logic;
              sr:out std_logic_vector(longitud-1 downto 0)
       );
end component;

--DECLARACION DE SEÑALES

type estados is (inicio,error,reconocida,
              cargar_0,cargar_1,cargar_2,cargar_3,cargar_4,cargar_5,
              leer_0,leer_1,leer_2,leer_3,
              cargarN,
              lanzar_1,lanzar_2,lanzar_3,lanzar_4,lanzar_5,
              leer_cf_0,leer_cf_1,cargar_cf_0,cargar_cf_1);
signal estado:estados;

signal ld_tipo_orden,rst_tipo_orden:std_logic;
signal ld_orden_reconocida,rst_orden_reconocida:std_logic;
signal ld_orden_error,rst_orden_error:std_logic;
signal ent_tipo_orden:std_logic_vector(7 downto 0);
signal ent_orden_reconocida,ent_orden_error:std_logic_vector(0 downto 0);
signal z_orden_reconocida,z_orden_error:std_logic_vector(0 downto 0);

begin

       reg_tipo_orden: registro generic map(8) port
                     map(ent_tipo_orden,clk,ld_tipo_orden,rst_tipo_orden,tipo_orden);
       reg_orden_reconocida: registro generic map(1) port
                     map(ent_orden_reconocida,clk,ld_orden_reconocida,
                            rst_orden_reconocida,z_orden_reconocida);
       reg_orden_error: registro generic map(1) port
                     map(ent_orden_error,clk,ld_orden_error,
                            rst_orden_error,z_orden_error);

       orden_reconocida<=z_orden_reconocida(0);
       orden_error<=z_orden_error(0);

       process(reset,clk,tecla_ascii,pulsacion_realizada,estado)
       begin

              --SALIDAS

              rst_tipo_orden<='0';
              rst_orden_reconocida<='0';
              rst_orden_error<='0';

              ld_tipo_orden<='0';
              ld_orden_reconocida<='0';
              ld_orden_error<='0';
```

```
case estado is
	when inicio =>
			rst_orden_reconocida<='1';
			rst_orden_error<='1';
	when cargar_5 =>
			ent_tipo_orden<=ORDEN_CARGAR;
			ld_tipo_orden<='1';
	when cargarN =>
			ent_tipo_orden<=ORDEN_CARGARN;
			ld_tipo_orden<='1';
	when leer_3 =>
			ent_tipo_orden<=ORDEN_LEER;
			ld_tipo_orden<='1';
	when lanzar_5 =>
			ent_tipo_orden<=ORDEN_LANZAR;
			ld_tipo_orden<='1';
	when leer_cf_1 =>
			ent_tipo_orden<=ORDEN_LEERCF;
			ld_tipo_orden<='1';
	when cargar_cf_1 =>
			ent_tipo_orden<=ORDEN_CARGARCF;
			ld_tipo_orden<='1';
	when error =>
			ld_orden_error<='1';
			ent_orden_error<="1";
	when reconocida =>
			ld_orden_reconocida<='1';
			ent_orden_reconocida<="1";
	when others =>
			null;
end case;

--TRANSICION DE ESTADOS

if (reset='1') then
	estado<=inicio;
elsif (clk='1' and clk'event) then
	if(pulsacion_realizada='1') then
		case estado is
			when inicio =>
				if (tecla_ascii=LETRA_C) then
					estado<=cargar_0;
				elsif (tecla_ascii=LETRA_L) then
					estado<=leer_0;
				else
					estado<=error;
				end if;
			when cargar_0 =>
				if (tecla_ascii=LETRA_A) then
					estado<=cargar_1;
				else
					estado<=error;
				end if;
			when cargar_1 =>
				if (tecla_ascii=LETRA_R) then
					estado<=cargar_2;
				else
					estado<=error;
				end if;
			when cargar_2 =>
				if (tecla_ascii=LETRA_G) then
					estado<=cargar_3;
				else
					estado<=error;
				end if;
			when cargar_3 =>
				if (tecla_ascii=LETRA_A) then
					estado<=cargar_4;
				else
					estado<=error;
				end if;
			when cargar_4 =>
				if (tecla_ascii=LETRA_R) then
					estado<=cargar_5;
				else
```

```
                estado<=error;
        end if;
when cargar_5 =>
        if (tecla_ascii=CARACTER_ESPACIO) then
                estado<=reconocida;
        elsif (tecla_ascii=LETRA_N) then
                estado<=cargarN;
        elsif (tecla_ascii=LETRA_C) then
                estado<=cargar_cf_0;
        else
                estado<=error;
        end if;
when cargarN =>
        if (tecla_ascii=CARACTER_ESPACIO) then
                estado<=reconocida;
        else
                estado<=error;
        end if;
when leer_0 =>
        if (tecla_ascii=LETRA_E) then
                estado<=leer_1;
        elsif (tecla_ascii=LETRA_A) then
                estado<=lanzar_1;
        else
                estado<=error;
        end if;
when leer_1 =>
        if (tecla_ascii=LETRA_E) then
                estado<=leer_2;
        else
                estado<=error;
        end if;
when leer_2 =>
        if (tecla_ascii=LETRA_R) then
                estado<=leer_3;
        else
                estado<=error;
        end if;
when leer_3 =>
        if (tecla_ascii=CARACTER_ESPACIO) then
                estado<=reconocida;
        elsif (tecla_ascii=LETRA_C) then
                estado<=leer_cf_0;
        else
                estado<=error;
        end if;
when lanzar_1 =>
        if (tecla_ascii=LETRA_N) then
                estado<=lanzar_2;
        else
                estado<=error;
        end if;
when lanzar_2 =>
        if (tecla_ascii=LETRA_Z) then
                estado<=lanzar_3;
        else
                estado<=error;
        end if;
when lanzar_3 =>
        if (tecla_ascii=LETRA_A) then
                estado<=lanzar_4;
        else
                estado<=error;
        end if;
when lanzar_4 =>
        if (tecla_ascii=LETRA_R) then
                estado<=lanzar_5;
        else
                estado<=error;
        end if;
when lanzar_5 =>
        if (tecla_ascii=CARACTER_ESPACIO) then
                estado<=reconocida;
        else
                estado<=error;
        end if;
when leer_cf_0 =>
```

```
                                        if (tecla_ascii=LETRA_F) then
                                                estado<=leer_cf_1;
                                        else
                                                estado<=error;
                                        end if;
                                when leer_cf_1 =>
                                        if (tecla_ascii=CARACTER_ESPACIO) then
                                                estado<=reconocida;
                                        else
                                                estado<=error;
                                        end if;
                                when cargar_cf_0 =>
                                        if (tecla_ascii=LETRA_F) then
                                                estado<=cargar_cf_1;
                                        else
                                                estado<=error;
                                        end if;
                                when cargar_cf_1 =>
                                        if (tecla_ascii=CARACTER_ESPACIO) then
                                                estado<=reconocida;
                                        else
                                                estado<=error;
                                        end if;
                                when others =>
                                        estado<=estado;
                        end case;
                end if;
        end if;
    end process;
end arch;
```

Lector de datos

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

entity lector_datos is
    Port ( enable:in std_logic;
         reset:in std_logic;
         clk:in std_logic;
         pulsacion_realizada : in std_logic;
         tecla_ascii : in std_logic_vector(7 downto 0);
         byte : out std_logic_vector(7 downto 0);
         byte_reconocido : out std_logic;
         error_conversor : out std_logic
          );
end lector_datos;

architecture arch of lector_datos is

--DECLARACION DE COMPONENTES

component ascii2hex
    port (
         tecla_ascii: in STD_LOGIC_VECTOR (7 downto 0);
         hex: out STD_LOGIC_VECTOR (3 downto 0);
         error_conversor: out STD_LOGIC
    );
end component;

component registro
        generic(longitud:natural:=5);
        port(
                ent:in std_logic_vector(longitud-1 downto 0);
                clk,carga_r,reset:in std_logic;
                sr:out std_logic_vector(longitud-1 downto 0)
        );
end component;



--DECLARACION DE SEÑALES

type estados is (parado,inicio,estado1,estado2,estado_error);
signal estado:estados;

signal hex1,hex2:std_logic_vector(3 downto 0);
signal error1,error2:std_logic;

signal ent_caracter1,ent_caracter2,caracter1,caracter2:std_logic_vector(7 downto 0);
signal ld_caracter1,rst_caracter1,ld_caracter2,rst_caracter2:std_logic;

begin

        reg_caracter1: registro generic map(8) port
                map(ent_caracter1,clk,ld_caracter1,rst_caracter1,caracter1);
        reg_caracter2: registro generic map(8) port
                map(ent_caracter2,clk,ld_caracter2,rst_caracter2,caracter2);

        h1: ascii2hex port map(caracter1,hex1,error1);
        h2: ascii2hex port map(caracter2,hex2,error2);

        byte<=hex1&hex2;

        process(reset,clk,tecla_ascii,pulsacion_realizada)
        begin

                --SALIDAS

                rst_caracter1<='0';
                rst_caracter2<='0';

                ld_caracter1<='0';
                ld_caracter2<='0';
```

```
		byte_reconocido<='0';
		error_conversor<='0';

		case estado is
			when parado=>
					null;
			when inicio=>
				if(pulsacion_realizada='1' and
							tecla_ascii/=CARACTER_ESPACIO) then
					ent_caracter1<=tecla_ascii;
					ld_caracter1<='1';
				end if;
			when estado1=>
				if(pulsacion_realizada='1' and
							tecla_ascii/=CARACTER_ESPACIO) then
					ent_caracter2<=tecla_ascii;
					ld_caracter2<='1';
				end if;
			when estado2=>
				if(error2='0') then
					byte_reconocido<='1';
				end if;
			when estado_error=>
				error_conversor<='1';
		end case;

		--TRANSICION DE ESTADOS

		if (reset='1') then
			estado<=parado;
		elsif (clk='1' and clk'event) then
			case estado is
				when parado =>
					if(enable='1') then
						estado<=inicio;
					end if;
				when inicio =>
					if(pulsacion_realizada='1' and
							tecla_ascii/=CARACTER_ESPACIO) then
						estado<=estado1;
					end if;
				when estado1 =>

					if(pulsacion_realizada='1' and
							tecla_ascii/=CARACTER_ESPACIO) then
						if(error1='0') then
							estado<=estado2;
						else
							estado<=estado_error;
						end if;
					end if;
				when estado2 =>
					if(error2='0') then
						estado<=inicio;
					else
						estado<=estado_error;
					end if;
				when others => estado<=estado;
			end case;
		end if;
	end process;
end arch;
```

Registro

```
library IEEE;
use IEEE.std_logic_1164.all;

entity registro is
	generic(longitud:natural:=5);
	port(
		ent:in std_logic_vector(longitud-1 downto 0);
		clk,carga_r,reset:in std_logic;
		sr:out std_logic_vector(longitud-1 downto 0)
	);
end registro;

architecture comportamiento of registro is
begin
    process(clk,reset)
    begin
	if(reset='1') then
		sr<=(others=>'0');
	elsif(clk='1' and clk'event) then
		if(carga_r='1') then
			sr<=ent;
		end if;
	end if;
    end process;
end comportamiento;
```

Ascii2hex

```
library IEEE;
use IEEE.std_logic_1164.all;
use constantes.all;

entity ascii2hex is
    port (
        tecla_ascii: in STD_LOGIC_VECTOR (7 downto 0);
        hex: out STD_LOGIC_VECTOR (3 downto 0);
        error_conversor: out STD_LOGIC
    );
end ascii2hex;

architecture ascii2hex_arch of ascii2hex is

signal hex_aux:std_logic_vector(4 downto 0);
begin

	hex_aux<= "00000" when tecla_ascii=NUMERO_0 else
			“00001" when tecla_ascii=NUMERO_1 else
			"00010" when tecla_ascii=NUMERO_2 else
			"00011" when tecla_ascii=NUMERO_3 else
			"00100" when tecla_ascii=NUMERO_4 else
			"00101" when tecla_ascii=NUMERO_5 else
			"00110" when tecla_ascii=NUMERO_6 else
			"00111" when tecla_ascii=NUMERO_7 else
			"01000" when tecla_ascii=NUMERO_8 else
			"01001" when tecla_ascii=NUMERO_9 else
			"01010" when tecla_ascii=LETRA_A else
			"01011" when tecla_ascii=LETRA_B else
			"01100" when tecla_ascii=LETRA_C else
			"01101" when tecla_ascii=LETRA_D else
			"01110" when tecla_ascii=LETRA_E else
			"01111" when tecla_ascii=LETRA_F else
			"11111";

	hex<=hex_aux(3 downto 0);
	error_conversor<=hex_aux(4);

end ascii2hex_arch;
```

Ejecutor de órdenes

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--use constantes.ALL;
--PARA SIMULAR comentar la anterior y descomentar la siguiente
--Ademas descomentar las constantes de abajo con los tipos de ordenes
--use IEEE.constantes.ALL;

entity ejecuta_orden is
	Port (tipo_orden : in std_logic_vector(7 downto 0);
		enable_ejecuta : in std_logic;
		clk : in std_logic;
		reset : in std_logic;
		dato_leido_registro: in std_logic_vector(7 downto 0);
		direccion_memoria : out std_logic_vector(31 downto 0);
		escritura:out std_logic;
		numero_registro: out std_logic_vector(7 downto 0);
		fin_ejecuta:out std_logic;
		load_dato_leido_memoria:out std_logic;
		reg_direccion_out:out std_logic_vector(31 downto 0);
		stado:out std_logic_vector (4 downto 0);
		enable_mem:out std_logic;
		fin_memoria:in std_logic;
		fin_cola:out std_logic_vector(7 downto 0);
		memoria_preparada:in std_logic
);
end ejecuta_orden;

architecture arch of ejecuta_orden is

--PARA SIMULAR
constant ORDEN_CARGAR:std_logic_vector(7 downto 0):="00000000";
constant ORDEN_LEER:std_logic_vector(7 downto 0):="00000001";
constant ORDEN_CARGARN:std_logic_vector(7 downto 0):="00000010";
constant ORDEN_ERROR:std_logic_vector(7 downto 0):="00000011";
constant ORDEN_LANZAR:std_logic_vector(7 downto 0):="00000100";
constant ORDEN_LEERCF:std_logic_vector(7 downto 0):="00000101";
constant ORDEN_CARGARCF:std_logic_vector(7 downto 0):="00000110";

--DECLARACION DE COMPONENTES

component registro
	generic(longitud:natural:=5);
	port(
		ent:in std_logic_vector(longitud-1 downto 0);
		clk,carga_r,reset:in std_logic;
		sr:out std_logic_vector(longitud-1 downto 0)
	);
end component;



constant CICLOS_ESPERA_MEMORIA:std_logic_vector(7 downto 0):="00000100";

type estados is (inicio,lee_dir3,lee_dir2,lee_dir1,lee_dir0,carga_regdir,
                                            cargar0,cargar1,
                                            leer0,leer1,
                                            cargarN0,cargarN1,cargarN2,cargarN3,fin,
                                            cargar00,leer00,cargarN00,
                                            lanzar1,lanzar2,lanzar3,lanzar4);
signal estado:estados;
signal numero_registro_aux,numero_palabras,reg_dato: std_logic_vector (7 downto 0);

signal
load_reg_dir3,load_reg_dir2,load_reg_dir1,load_reg_dir0,ld_fin_cola,rst_fin_cola:std_log
ic;

signal ent_fin_cola,z_fin_cola: std_logic_vector (7 downto 0);

signal ent_dir3,dir3,ent_dir2,dir2,ent_dir1,dir1,ent_dir0,dir0:std_logic_vector(7 downto
0);
signal ent_numero_registro:std_logic_vector(7 downto 0);
```

```
signal ent_contador_palabras,contador_palabras,ent_numero_palabras:std_logic_vector(7
downto 0);
signal ent_direccion,direccion:std_logic_vector(31 downto 0);

signal ld_dir3,ld_dir2,ld_dir1,ld_dir0,ld_direccion:std_logic;
signal ld_contador_palabras,ld_numero_registro,ld_numero_palabras:std_logic;
signal rst_contador_palabras,rst_numero_registro:std_logic;

begin

	reg_fin_cola: registro generic map(8) port
map(ent_fin_cola,clk,ld_fin_cola,rst_fin_cola,z_fin_cola);

	reg_dir3: registro generic map(8) port map(ent_dir3,clk,ld_dir3,reset,dir3);
	reg_dir2: registro generic map(8) port map(ent_dir2,clk,ld_dir2,reset,dir2);
	reg_dir1: registro generic map(8) port map(ent_dir1,clk,ld_dir1,reset,dir1);
	reg_dir0: registro generic map(8) port map(ent_dir0,clk,ld_dir0,reset,dir0);

	reg_direccion: registro generic map(32) port
map(ent_direccion,clk,ld_direccion,reset,direccion);

	reg_numero_registro: registro generic map(8) port
map(ent_numero_registro,clk,ld_numero_registro,rst_numero_registro,numero_registro);

	reg_contador_palabras: registro generic map(8) port
map(ent_contador_palabras,clk,ld_contador_palabras,rst_contador_palabras,contador_palabr
as);

	reg_numero_palabras: registro generic map(8) port
map(ent_numero_palabras,clk,ld_numero_palabras,reset,numero_palabras);

	process(clk,reset,enable_ejecuta)
	begin

		ld_dir3<='0';
		ld_dir2<='0';
		ld_dir1<='0';
		ld_dir0<='0';
		ld_direccion<='0';
		ld_numero_registro<='0';
		rst_numero_registro<='0';
		rst_fin_cola<='0';
		ld_fin_cola<='0';
		escritura<='0';
		enable_mem<='0';
		fin_ejecuta<='0';
		load_dato_leido_memoria<='0';
		ld_contador_palabras<='0';
		rst_contador_palabras<='0';
		ld_numero_palabras<='0';


		--SALIDAS
		case estado is
			when inicio => rst_numero_registro<='1';
			when lee_dir3=>
				ld_dir3<='1';
				ent_dir3<=dato_leido_registro;
				ld_numero_registro<='1';
				ent_numero_registro<="00000001";
			when lee_dir2=>
				ld_dir2<='1';
				ent_dir2<=dato_leido_registro;
				ld_numero_registro<='1';
				ent_numero_registro<="00000010";
			when lee_dir1=>
				ld_dir1<='1';
				ent_dir1<=dato_leido_registro;
				ld_numero_registro<='1';
				ent_numero_registro<="00000011";
			when lee_dir0=>
				ld_dir0<='1';
				ent_dir0<=dato_leido_registro;
				ld_numero_registro<='1';
				ent_numero_registro<="00000100";
			when carga_regdir=>
				ld_direccion<='1';
```

```
                ent_direccion<=dir3 & dir2 & dir1 & dir0;
                ld_numero_registro<='1';
                if (tipo_orden=ORDEN_CARGAR) then
                        ent_numero_registro<="00000100";
                elsif (tipo_orden=ORDEN_LANZAR) then
                        ent_numero_registro<="00000000";
                else
                        ent_numero_registro<="00000100";

                end if;
        when cargar0 =>
                null;
        when cargar1=>
                        escritura<='1';
                        enable_mem<='1';
        when leer00=>
                enable_mem<='1';
        when leer0=>
                enable_mem<='1';
        when leer1=>
                load_dato_leido_memoria<='1';
        when cargarN0=>
                ld_numero_registro<='1';
                ent_numero_registro<="00000101";
                rst_contador_palabras<='1';
                ld_numero_palabras<='1';
                ent_numero_palabras<=dato_leido_registro;
        when cargarN1=>
                null;
        when cargarN2=>
                escritura<='1';
                enable_mem<='1';
                if(fin_memoria='1') then
                        ld_contador_palabras<='1';
                        ent_contador_palabras<=contador_palabras+'1';
                end if;

        when cargarN3=>
                ld_numero_registro<='1';
                ent_numero_registro<="00000101"+contador_palabras;
        when lanzar1=>
                ld_numero_registro<='1';
                ent_numero_registro<="00000000";
                escritura<='1';
        when lanzar2=>
                escritura<='1';
                enable_mem<='1';
        when lanzar3=>
                escritura<='1';
                enable_mem<='1';
        when lanzar4=>
                ld_fin_cola<='1';
                ent_fin_cola<=z_fin_cola+'1';
        when fin=>
                fin_ejecuta<='1';
        when others=>
                null;

end case;

if(clk='1' and clk'event) then
        case estado is
                when cargar00=>
                        direccion_memoria<=direccion;
                when leer00=>
                        direccion_memoria<=direccion;
                when cargarN1=>
                        direccion_memoria<=direccion+contador_palabras;
                when lanzar1=>
                        direccion_memoria<="00000000000000000000000000"
                                                                & z_fin_cola;
                when others=>
                        null;
        end case;
end if;

--TRANSICION DE ESTADOS
```

```
	if(reset='1') then
		estado<=inicio;
	elsif(clk='1' and clk'event) then
		case estado is
			when inicio=>  if (enable_ejecuta='1') then
						estado<=lee_dir3;
					end if;
			when lee_dir3=>		estado<=lee_dir2;
			when lee_dir2=>		estado<=lee_dir1;
			when lee_dir1=>		estado<=lee_dir0;

			when lee_dir0=>		estado<=carga_regdir;
			when carga_regdir =>  if (tipo_orden=ORDEN_CARGAR) then
						estado<=cargar00;
					elsif (tipo_orden=ORDEN_LEER) then
						estado<=leer00;
					elsif (tipo_orden=ORDEN_LANZAR) then
						estado<=lanzar1;
					else
						estado<=cargarN00;
					end if;
			when cargar00=>		estado<=cargar0;
			when cargar0=>		estado<=cargar1;
			when cargar1=> if(fin_memoria='1') then
						estado<=fin;
					end if;
			when leer00=>
				if(memoria_preparada='1') then
					estado<=leer0;
				end if;
			when leer0=>   if(fin_memoria='1') then
						estado<=leer1;
					end if;
			when leer1=>		estado<=fin;
			when cargarN00=>	estado<=cargarN0;

			when cargarN0=>		estado<=cargarN1;
			when cargarN1=>		estado<=cargarN2;
			when cargarN2=>		if(fin_memoria='1') then
						estado<=cargarN3;
					end if;
			when cargarN3=>
				if(contador_palabras=numero_palabras) then
					estado<=fin;
				else
					estado<=cargarN1;
				end if;
			when lanzar1=> estado<=lanzar2;

			when lanzar2=> estado<=lanzar3;
			when lanzar3=> if (fin_memoria='1') then
						estado<=lanzar4;
					end if;
			when lanzar4=> estado<=fin;
			when fin=>     estado<=estado;
		end case;
	end if;
end process;

stado <= "00000" when estado=inicio else
			"00001" when estado=lee_dir3 else
			"00010" when estado=lee_dir2 else
			"00011" when estado=lee_dir1 else
			"00100" when estado=lee_dir0 else
			"00101" when estado=carga_regdir else
			"00110" when estado=cargar0 else
			"00111" when estado=cargar1 else
			"01000" when estado=leer0 else
			"01001" when estado=leer1 else
			"01010" when estado=cargarN0 else
			"01011" when estado=cargarN1 else
			"01100" when estado=cargarN2 else
			"01101" when estado=cargarN3 else
			"01110" when estado=fin else
			"01111" when estado=cargar00 else
			"10000" when estado=leer00 else
			"10001" when estado=cargarN00 else
```

```
                              "10010" when estado=lanzar1 else
                              "10011" when estado=lanzar2 else
                              "10100" when estado=lanzar3 else
                              "10101";

       reg_direccion_out<=contador_palabras&"00000000"&numero_palabras&"00000000";
       fin_cola<=z_fin_cola;

end arch;
```

Interfaz de memoria

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.parameter_64bit_00.all;

entity interfaz_memoria is
	port(
		memoria_preparada:out std_logic;
		dcms_locked:in std_logic;
		SYSTEM_CLOCK:in std_logic;
		reset:in std_logic;
		escritura:in std_logic;
		dir_mem:in std_logic_vector(22 downto 0);
		dato_in_mem1:in std_logic_vector(127 downto 0);
		dato_in_mem2:in std_logic_vector(127 downto 0);
		enable_mem:in std_logic;
		fin_memoria:out std_logic;
		dato_out_mem1:out std_logic_vector(127 downto 0);
		dato_out_mem2:out std_logic_vector(127 downto 0);
		dato_out_mem_antes:out std_logic_vector(127 downto 0);
		dato_out_mem_despues:out std_logic_vector(127 downto 0);
		rst_dqs_div_in:in std_logic;
		rst_dqs_div_out:out std_logic;
		ddr_dqs:inout std_logic_vector(7 downto 0);
		ddr_dq:inout std_logic_vector(63 downto 0):= (OTHERS => 'Z');
		ddr_cke1:out std_logic;
		ddr_cke2:out std_logic;
		ddr_csb1:out std_logic;
		ddr_csb2:out std_logic;
		ddr_rasb:out std_logic;
		ddr_casb:out std_logic;
		ddr_web:out std_logic;
		ddr_dm:out std_logic_vector(((mask_width/2)-1) downto 0);
		ddr_ba:out std_logic_vector((bank_address_p-1) downto 0);
		ddr_address:out std_logic_vector((row_address_p-1) downto 0);
		ddr_clk0:out std_logic;
		ddr_clk0b:out std_logic;
		ddr_clk1:out std_logic;
		ddr_clk1b:out std_logic;
		ddr_clk2:out std_logic;
		ddr_clk2b:out std_logic;
		stado:out std_logic_vector(4 downto 0);
		stadold:out std_logic_vector(1 downto 0)
	);
end interfaz_memoria;

architecture arch of interfaz_memoria is

--DECLARACION DE COMPONENTES

component registro
	generic(longitud:natural:=5);
	port(
		ent:in std_logic_vector(longitud-1 downto 0);
		clk,carga_r,reset:in std_logic;
		sr:out std_logic_vector(longitud-1 downto 0)
	);
end component;

component mem_interface_top
port(
	wait_200us2:out std_logic;
	dip1: in STD_LOGIC;
	dip2: in std_logic;
	dip3: in std_logic;
	reset_in: in STD_LOGIC;
	SYSTEM_CLOCK: in STD_LOGIC;
	clk_int_1: out STD_LOGIC;
	clk90_int_1: out STD_LOGIC;
	clk180_1: out STD_LOGIC;
	clk270_1: out STD_LOGIC;
	sys_rst_1: out STD_LOGIC;
	sys_rst90_1: out STD_LOGIC;
```

```
        sys_rst180_1: out STD_LOGIC;
        sys_rst270_1: out STD_LOGIC;
        cntrl0_rst_dqs_div_in    : in std_logic;
        cntrl0_rst_dqs_div_out   : out std_logic;
        cntrl0_ddr1_casb         : out STD_LOGIC;
        cntrl0_ddr1_cke1          : out STD_LOGIC;
        cntrl0_ddr1_cke2          : out STD_LOGIC;
        cntrl0_ddr1_clk0  : out STD_LOGIC;
        cntrl0_ddr1_clk0b   : out STD_LOGIC;
        cntrl0_ddr1_clk1  : out STD_LOGIC;
        cntrl0_ddr1_clk1b   : out STD_LOGIC;
        cntrl0_ddr1_clk2  : out STD_LOGIC;
        cntrl0_ddr1_clk2b   : out STD_LOGIC;
       cntrl0_burst_done   : in std_logic;
       cntrl0_user_input_address  : in std_logic_vector(22 downto 0);
       cntrl0_user_bank_address   : in std_logic_vector(1 downto 0);
       cntrl0_user_input_data   : in std_logic_vector(127 downto 0);
       cntrl0_user_data_mask   : in std_logic_vector(15 downto 0);
       cntrl0_auto_ref_req     : out STD_LOGIC;
       cntrl0_ar_done          : out STD_LOGIC;
       cntrl0_user_cmd_ack     : out STD_LOGIC;
       cntrl0_user_data_valid  : out STD_LOGIC;
       cntrl0_user_output_data : out STD_LOGIC_vector(127 downto 0);
       cntrl0_init_val         : out STD_LOGIC;
       cntrl0_ddr1_csb1          : out STD_LOGIC;
       cntrl0_ddr1_csb2          : out STD_LOGIC;
       cntrl0_ddr1_rasb         : out STD_LOGIC;
       cntrl0_ddr1_web          : out STD_LOGIC;
        cntrl0_user_config_register      : in STD_LOGIC_VECTOR (9 downto 0);
       cntrl0_ddr1_address      : out STD_LOGIC_VECTOR(12 downto 0);
       cntrl0_ddr1_ba           : out STD_LOGIC_VECTOR(1 downto 0);
       cntrl0_user_command_register   : in std_logic_vector(2 downto 0);
       cntrl0_ddr1_dm           : out STD_LOGIC_VECTOR(7 downto 0);
       cntrl0_ddr1_dqs          : inout STD_LOGIC_VECTOR(7 downto 0);
       cntrl0_ddr1_dq           : inout STD_LOGIC_VECTOR(63 downto 0)
       );
end component;

--DECLARACION DE SEÑALES

type estados_generales is
(inicio,ini_1,ini_2,ini_3,ini_4,espera_comando,leer0,leer1,leer2,leer3,leer4,leer5,leer6
,leer7,cargar0,cargar1,cargar2,cargar3,cargar4,cargar5,cargar6,cargar7,espera_auto_refre
sco);
signal estado_general:estados_generales;

type estados_lectura_datos is (espera_datos,leer_datos1,leer_datos2,espera_fin_leer);
signal estado_lectura_datos:estados_lectura_datos;

signal contador:std_logic_vector(2 downto 0);
signal reset_contador,cuenta:std_logic;

signal ld_entradas,rst_entradas,ld_salidas,rst_salidas:std_logic;
signal z_escritura,lectura:std_logic;
signal z_dato_in_mem,ent_dato_out_mem:std_logic_vector(7 downto 0);
signal z_dir_mem:std_logic_vector(18 downto 0);

signal activa_mem:std_logic;
signal gnd,vcc:std_logic;

signal m_ent_dato_out_mem,m_z_dato_in_mem:std_logic_vector(31 downto 0);
signal dip1:std_logic;
signal dip2:std_logic;
signal auto_ref_req:std_logic;
signal dip3:std_logic;
signal reset_in:std_logic;
signal user_input_data:std_logic_vector(127 downto 0);
signal user_data_mask:std_logic_vector(((mask_width)-1) downto 0);
signal user_output_data:std_logic_vector(127 downto 0):=(OTHERS => 'Z');
signal user_data_valid:std_logic;
signal user_input_address:std_logic_vector(((row_address_p + column_address_p)-1) downto
0);
signal user_bank_address:std_logic_vector((bank_address_p-1) downto 0);
signal user_config_register:std_logic_vector(9 downto 0);
signal user_command_register:std_logic_vector(2 downto 0);
signal user_cmd_ack:std_logic;
signal burst_done:std_logic;
```

```
signal init_val:std_logic;
signal ar_done:std_logic;
signal sys_rst:std_logic;
signal sys_rst90:std_logic;
signal sys_rst180:std_logic;
signal sys_rst270:std_logic;
signal ld_reg_dato1,ld_reg_dato2:std_logic;
signal ent_reg_dato1,sal_reg_dato1,ent_reg_dato2,sal_reg_dato2:std_logic_vector(127
downto 0);
signal ent_fin_lectura,ld_fin_lectura,fin_lectura:std_logic;
signal ddr_clk0c,ddr_clk0bc:std_logic;
signal wait_200us:std_logic;

signal clk_int_1,clk90_int_1,clk180_1,clk270_1:std_logic;
signal clk0,clk90,clk180:std_logic;

signal ld_veces:std_logic;
signal ent_veces,z_veces:std_logic_vector(3 downto 0);

signal ld_reg_dato_antes,ld_reg_dato_despues:std_logic;
signal sal_reg_dato_antes,sal_reg_dato_despues:std_logic_vector(127 downto 0);

begin

        veces<=z_veces;
        l_user_data_valid<=user_data_valid;
        l_load_dato1<=ld_reg_dato1;
        l_load_dato2<=ld_reg_dato2;

        clk0<=clk_int_1;
        clk90<=clk90_int_1;
        clk180<=clk180_1;

        gnd<='0';
        vcc<='1';

        --Señales que tienen un valor fijo
        dip1<='0';
        dip3<='0';
        dip2<='0';

        reset_in<=not reset;

        ddr_clk0<=ddr_clk0c;
        ddr_clk0b<=ddr_clk0bc;

        user_data_mask<="0000000000000000";
        user_bank_address<="10";

        mbr: mem_interface_top
                port map(
                        wait_200us,
                        dip1,
                        dip2,
                        dip3,
                        reset_in,
                        SYSTEM_CLOCK,
          clk_int_1,
          clk90_int_1,
          clk180_1,
          clk270_1,
                        sys_rst,
                        sys_rst90,
                        sys_rst180,
                        sys_rst270,
                        rst_dqs_div_in,
                        rst_dqs_div_out,
                        ddr_casb,
                        ddr_cke1,
                        ddr_cke2,
                        ddr_clk0c,
                        ddr_clk0bc,
                        ddr_clk1,
                        ddr_clk1b,
                        ddr_clk2,
                        ddr_clk2b,
                        burst_done,
```

```
                user_input_address,
                user_bank_address,
                user_input_data,
                user_data_mask,
                auto_ref_req,
                ar_done,
                user_cmd_ack,
                user_data_valid,
                user_output_data,
                init_val,
                ddr_csb1,
                ddr_csb2,
                ddr_rasb,
                ddr_web,
                user_config_register,
                ddr_address,
                ddr_ba,
                user_command_register,
                ddr_dm,
                ddr_dqs,
                ddr_dq
        );

reg_fin_lectura:
process(clk90,reset)
begin
        if(reset='1') then
                fin_lectura<='0';
        elsif(clk90='1' and clk90'event) then
                if(ld_fin_lectura='1') then
                        fin_lectura<=ent_fin_lectura;
                end if;
        end if;
end process;

reg_dato1: registro generic map(128) port
                map(user_output_data,clk90,ld_reg_dato1,reset,sal_reg_dato1);
reg_dato2: registro generic map(128) port
                map(user_output_data,clk90,ld_reg_dato2,reset,sal_reg_dato2);

reg_dato_antes: registro generic map(128) port
        map(user_output_data,clk90,ld_reg_dato_antes,reset,sal_reg_dato_antes);
reg_dato_despues: registro generic map(128) port
      map(user_output_data,clk90,ld_reg_dato_despues,reset,sal_reg_dato_despues);

reg_veces: registro generic map(4) port
                                    map(ent_veces,clk0,ld_veces,reset,z_veces);

dato_out_mem1<=sal_reg_dato1;
dato_out_mem2<=sal_reg_dato2;

dato_out_mem_antes<=sal_reg_dato_antes;
dato_out_mem_despues<=sal_reg_dato_despues;

contador_ciclos:
process(clk0,reset_contador)
begin
        if(reset_contador='1') then
                contador<="000";
        elsif(clk0='1' and clk0'event) then
                contador<=contador+'1';
        end if;
end process;

salidas_sincronas_clk0:
process(clk0,reset,enable_mem,dir_mem,estado_general)
begin
        if(clk0='1' and clk0'event) then
                case estado_general is
                        when ini_1=>
                                user_config_register<="1000110010";
                        when leer0=>
                                user_input_address<=dir_mem;
                        when cargar0=>
                                user_input_address<=dir_mem;
                        when others=>
                                null;
```

```
                end case;
        end if;
end process;

memoria_preparada<='1' when
                estado_general=espera_comando and auto_ref_req='0' else '0';

salidas_moore:
process(estado_general)
begin
        reset_contador<='0';
        burst_done<='0';
        fin_memoria<='0';
        ld_veces<='0';
        case estado_general is
                when espera_comando=>
                        null;
                when leer0=>
                        ld_veces<='1';
                        ent_veces<=z_veces+'1';
                when leer2=>
                        reset_contador<='1';
                when leer4=>
                        burst_done<='1';
                when leer5=>
                        burst_done<='1';
                when leer7=>
                        fin_memoria<='1';
                when cargar0=>
                        ld_veces<='1';
                        ent_veces<=z_veces+'1';
                when cargar2=>
                        reset_contador<='1';
                when cargar6=>
                        burst_done<='1';
                when cargar7=>
                        burst_done<='1';
                        fin_memoria<='1';
                when others=>
                        null;
        end case;
end process;

salidas_sincronas_clk90:
process(clk90,reset,enable_mem,dir_mem,estado_general)
begin
        if(clk90='1' and clk90'event) then
                case estado_general is
                        when cargar3=>
                                user_input_data<=dato_in_mem1;
                        when cargar4=>
                                user_input_data<=dato_in_mem2;
                        when others=>
                                null;
                end case;
        end if;

end process;

salidas_sincronas_clk180:
process(clk180,reset,enable_mem,dir_mem,estado_general)
begin
        if(clk180='1' and clk180'event) then
                case estado_general is
                        when ini_3=>
                                user_command_register<="010";
                        when ini_4=>
                                user_command_register<="000";
                        when espera_comando=>
                                user_command_register<="000";
                        when leer1=>
                                user_command_register<="110";
                        when leer6=>
                                user_command_register<="000";
                        when cargar1=>
                                user_command_register<="100";
                        when others=>
```

```
                                null;
                        end case;
                end if;
        end process;

        transicion_estados:
        process(clk0,reset,enable_mem,dir_mem,estado_general)
        begin

                if (reset='1') then
                        estado_general<=inicio;
                elsif (clk0'event and clk0='1') then
                        case estado_general is
                                when inicio=>
                                        if(wait_200us='0') then
                                                estado_general<=ini_1;
                                        end if;
                                when ini_1=>
                                        estado_general<=ini_2;
                                when ini_2=>
                                        estado_general<=ini_3;
                                when ini_3=>
                                        estado_general<=ini_4;
                                when ini_4=>
                                        if(init_val='1') then
                                                estado_general<=espera_comando;
                                        end if;
                                when espera_comando=>
                                        if(auto_ref_req='1') then
                                                estado_general<=espera_auto_refresco;
                                        elsif(enable_mem='1' and user_cmd_ack='0') then
                                                if(escritura='1') then
                                                        estado_general<=cargar0;
                                                else
                                                        estado_general<=leer0;
                                                end if;
                                        end if;
                                when espera_auto_refresco=>
                                        if(ar_done='1') then
                                                estado_general<=espera_comando;
                                        end if;
                                when leer0=>
                                        estado_general<=leer1;
                                when leer1=>
                                        estado_general<=leer2;
                                when leer2=>
                                        if(user_cmd_ack='1') then
                                                estado_general<=leer3;
                                        end if;
                                when leer3=>
                                        if(contador="11") then
                                                estado_general<=leer4;
                                        end if;
                                when leer4=>
                                        estado_general<=leer5;
                                when leer5=>
                                        estado_general<=leer6;
                                when leer6=>
                                        if(fin_lectura='1') then
                                                estado_general<=leer7;
                                        end if;
                                when leer7=>
                                        estado_general<=espera_comando;
                                when cargar0=>
                                        estado_general<=cargar1;
                                when cargar1=>
                                        estado_general<=cargar2;
                                when cargar2=>
                                        if(user_cmd_ack='1') then
                                                estado_general<=cargar3;
                                        end if;
                                when cargar3=>
                                        estado_general<=cargar4;
                                when cargar4=>
                                        estado_general<=cargar5;
                                when cargar5=>
                                        if(contador="11") then
```

```
                                        estado_general<=cargar6;
                                end if;
                        when cargar6=>
                                estado_general<=cargar7;
                        when cargar7=>
                                estado_general<=espera_comando;
                        when others=>
                                null;
                end case;
        end if;
end process;

ld_reg_dato1<='1' when
        user_data_valid='1' and estado_lectura_datos=espera_datos else '0';

ld_reg_dato_antes<='1' when
        user_data_valid='0' and estado_lectura_datos=espera_datos else '0';

lectura_datos_memoria:
process(clk90,reset,estado_lectura_datos,user_output_data)
begin

        ld_reg_dato2<='0';
        ld_fin_lectura<='0';
        ld_reg_dato_despues<='0';
        case estado_lectura_datos is
                when espera_datos=>
                        ent_fin_lectura<='0';
                        ld_fin_lectura<='1';
                        if(user_data_valid='1') then
                        end if;
                when leer_datos1=>
                        ld_reg_dato2<='1';
                when leer_datos2=>
                        ld_reg_dato_despues<='1';
                when espera_fin_leer=>
                        ent_fin_lectura<='1';
                        ld_fin_lectura<='1';
        end case;

        if(reset='1') then
                estado_lectura_datos<=espera_datos;
        elsif(clk90='1' and clk90'event) then
                case estado_lectura_datos is
                        when espera_datos=>
                                if(user_data_valid='1') then
                                        estado_lectura_datos<=leer_datos1;
                                end if;
                        when leer_datos1=>
                                estado_lectura_datos<=leer_datos2;
                        when leer_datos2=>
                                estado_lectura_datos<=espera_fin_leer;
                        when espera_fin_leer=>
                                if(user_data_valid='0') then
                                        estado_lectura_datos<=espera_datos;
                                end if;
                end case;
        end if;
end process;



stadold<=
        "00" when estado_lectura_datos=espera_datos else
        "01" when estado_lectura_datos=leer_datos1 else
        "10";--leer_datos2

stado<=
        "00000" when estado_general=inicio else
        "00001" when estado_general=ini_1 else
        "00010" when estado_general=ini_2 else
        "00011" when estado_general=ini_3 else
        "00100" when estado_general=ini_4 else
        "00101" when estado_general=espera_comando else
        "00110" when estado_general=leer0 else
        "00111" when estado_general=leer1 else
        "01000" when estado_general=leer2 else
```

```
                "01001" when estado_general=leer3 else
                "01010" when estado_general=leer4 else
                "01011" when estado_general=leer5 else
                "01100" when estado_general=leer6 else
                "01101" when estado_general=leer7 else
                "01110" when estado_general=cargar0 else
                "01111" when estado_general=cargar1 else
                "10000" when estado_general=cargar2 else
                "10001" when estado_general=cargar3 else
                "10010" when estado_general=cargar4 else
                "10011" when estado_general=cargar5 else
                "10100" when estado_general=cargar6 else
                "10101" when estado_general=cargar7 else
                "10110";--espera_auto_refresco

end arch;
```

Planificador

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

entity planificador is
	port (
		clk:in std_logic;
		reset: in std_logic;
		enable:in std_logic;
		fin_memoria: in std_logic;
		fin_carga_tarea: in std_logic;
		dato_leido_memoria:in std_logic_vector(7 downto 0);
		direccion_memoria:out std_logic_vector(18 downto 0);
		fin_cola:in std_logic_vector(7 downto 0);
		fin_tareas:in std_logic_vector(3 downto 0);
		enable_mem:out std_logic;
		escritura:out std_logic;
		z_inicio_cola:out std_logic_vector(7 downto 0);
		z_ejecutadas:out std_logic_vector(3 downto 0);
		stado:out std_logic_vector(4 downto 0);
		z_id_tareas:out std_logic_vector(31 downto 0)
	);
end planificador;

architecture planificador_arch of planificador is


--DECLARACION DE COMPONENTES


component registro
	generic(longitud:natural:=5);
	port(
		ent:in std_logic_vector(longitud-1 downto 0);
		clk,carga_r,reset:in std_logic;
		sr:out std_logic_vector(longitud-1 downto 0)
	);
end component;

--DECLARACION DE VARIABLES

signal
rst_contador_tareas_ejecucion,rst_inicio_cola,rst_dir_mem,rst_id,rst_id_aux,rst_tam_tare
a_aux_2,rst_tam_tarea_aux_1,rst_tam_tarea_aux_0: std_logic;
signal
ld_contador_tareas_ejecucion,ld_inicio_cola,ld_dir_mem,ld_id,ld_id_aux,ld_tam_tarea_aux_
2,ld_tam_tarea_aux_1,ld_tam_tarea_aux_0: std_logic;
signal
ent_inicio_cola,ent_id,ent_id_aux,ent_tam_tarea_aux_2,ent_tam_tarea_aux_1,ent_tam_tarea_
aux_0: std_logic_vector(7 downto 0);
signal inicio_cola,id,id_aux,tam_tarea_aux_2,tam_tarea_aux_1,tam_tarea_aux_0:
std_logic_vector(7 downto 0);
signal ent_dir_mem,dir_mem:std_logic_vector(18 downto 0);
signal ent_contador_tareas_ejecucion,contador_tareas_ejecucion:std_logic_vector(3 downto
0);
signal
ent_id_tarea_3,ent_id_tarea_2,ent_id_tarea_1,ent_id_tarea_0,id_tarea_3,id_tarea_2,id_tar
ea_1,id_tarea_0:std_logic_vector(7 downto 0);
signal
rst_id_tarea_3,rst_id_tarea_2,rst_id_tarea_1,rst_id_tarea_0,ld_id_tarea_3,ld_id_tarea_2,
ld_id_tarea_1,ld_id_tarea_0:std_logic;

type estados is
(inicio,espera_hueco_y_tarea,lee_memoria_id,carga_registro_id,lee_memoria_id_aux,inc_reg
_dir_mem_1_2,lee_tam_tarea_aux_2,inc_reg_dir_mem_1_1,lee_tam_tarea_aux_1,inc_reg_dir_mem
_1_0,lee_tam_tarea_aux_0,inc_reg_dir_mem_2,inc_reg_dir_mem_3,carga_tarea,inc_contador_ta
reas_ejecucion,quitar_tarea);
signal estado:estados;

begin

	reg_inicio_cola: registro generic map(8) port
		map(ent_inicio_cola,clk,ld_inicio_cola,rst_inicio_cola,inicio_cola);
```

```
reg_contador_tareas_ejecucion: registro generic map(4) port
       map(ent_contador_tareas_ejecucion,clk,ld_contador_tareas_ejecucion,
                     rst_contador_tareas_ejecucion,contador_tareas_ejecucion);
reg_dir_mem: registro generic map(19) port
                     map(ent_dir_mem,clk,ld_dir_mem,rst_dir_mem,dir_mem);
reg_id: registro generic map(8) port map(ent_id,clk,ld_id,rst_id,id);
reg_id_aux: registro generic map(8) port
                     map(ent_id_aux,clk,ld_id_aux,rst_id_aux,id_aux);
reg_tam_tarea_aux_2: registro generic map(8) port
       map(ent_tam_tarea_aux_2,clk,ld_tam_tarea_aux_2,rst_tam_tarea_aux_2,
                                                          tam_tarea_aux_2);
reg_tam_tarea_aux_1: registro generic map(8) port
       map(ent_tam_tarea_aux_1,clk,ld_tam_tarea_aux_1,rst_tam_tarea_aux_1,
                                                          tam_tarea_aux_1);
reg_tam_tarea_aux_0: registro generic map(8) port
       map(ent_tam_tarea_aux_0,clk,ld_tam_tarea_aux_0,rst_tam_tarea_aux_0,
                                                          tam_tarea_aux_0);
reg_id_tarea_3: registro generic map(8) port
              map(ent_id_tarea_3,clk,ld_id_tarea_3,rst_id_tarea_3,id_tarea_3);
reg_id_tarea_2: registro generic map(8) port
              map(ent_id_tarea_2,clk,ld_id_tarea_2,rst_id_tarea_2,id_tarea_2);
reg_id_tarea_1: registro generic map(8) port
              map(ent_id_tarea_1,clk,ld_id_tarea_1,rst_id_tarea_1,id_tarea_1);
reg_id_tarea_0: registro generic map(8) port
              map(ent_id_tarea_0,clk,ld_id_tarea_0,rst_id_tarea_0,id_tarea_0);

process(clk,reset)
begin
       rst_inicio_cola<='0';
       rst_contador_tareas_ejecucion<='0';
       rst_dir_mem<='0';
       rst_id<='0';
       rst_id_aux<='0';
       rst_tam_tarea_aux_2<='0';
       rst_tam_tarea_aux_1<='0';
       rst_tam_tarea_aux_0<='0';
       rst_id_tarea_3<='0';
       rst_id_tarea_2<='0';
       rst_id_tarea_1<='0';
       rst_id_tarea_0<='0';

       ld_inicio_cola<='0';
       ld_contador_tareas_ejecucion<='0';
       ld_dir_mem<='0';
       ld_id<='0';
       ld_id_aux<='0';
       ld_tam_tarea_aux_2<='0';
       ld_tam_tarea_aux_1<='0';
       ld_tam_tarea_aux_0<='0';
       ld_id_tarea_3<='0';
       ld_id_tarea_2<='0';
       ld_id_tarea_1<='0';
       ld_id_tarea_0<='0';

       enable_mem<='0';
       escritura<='0';

       case estado is
              when inicio=>
                     rst_inicio_cola<='1';
                     rst_contador_tareas_ejecucion<='1';

              when espera_hueco_y_tarea=>NULL;

              when lee_memoria_id=>
                     enable_mem<='1';

              when carga_registro_id=>
                     ld_dir_mem<='1';
                     ent_dir_mem<="0000000000011111111";
                     ld_id<='1';
                     ent_id<=dato_leido_memoria;

              when lee_memoria_id_aux=>
                     enable_mem<='1';
                     direccion_memoria<=dir_mem;
```

```
                        when inc_reg_dir_mem_1_2=>
                                ld_id_aux<='1';
                                ent_id_aux<=dato_leido_memoria;
                                ld_dir_mem<='1';
                                ent_dir_mem<=dir_mem+'1';

                        when lee_tam_tarea_aux_2=>
                                enable_mem<='1';
                                direccion_memoria<=dir_mem;

                        when inc_reg_dir_mem_1_1=>
                                ld_dir_mem<='1';
                                ent_dir_mem<=dir_mem+'1';
                                ld_tam_tarea_aux_2<='1';
                                ent_tam_tarea_aux_2<=dato_leido_memoria;

                        when lee_tam_tarea_aux_1=>
                                enable_mem<='1';
                                direccion_memoria<=dir_mem;

                        when inc_reg_dir_mem_1_0=>
                                ld_dir_mem<='1';
                                ent_dir_mem<=dir_mem+'1';
                                ld_tam_tarea_aux_1<='1';
                                ent_tam_tarea_aux_1<=dato_leido_memoria;

                        when lee_tam_tarea_aux_0=>
                                enable_mem<='1';
                                direccion_memoria<=dir_mem;

                        when inc_reg_dir_mem_2=>
                                ld_dir_mem<='1';
                                ent_dir_mem<=ent_dir_mem+'1';
                                ld_tam_tarea_aux_0<='1';
                                ent_tam_tarea_aux_0<=dato_leido_memoria;

                        when inc_reg_dir_mem_3=>
                                ld_dir_mem<='1';

        ent_dir_mem<=dir_mem+(tam_tarea_aux_2&tam_tarea_aux_1&tam_tarea_aux_0);

                        when carga_tarea=>
                                --HAY QUE CARGAR LA TAREA

                        when inc_contador_tareas_ejecucion=>
                                ld_contador_tareas_ejecucion<='1';
                                ld_inicio_cola<='1';
                                if (contador_tareas_ejecucion(3)='0') then
                                        ent_contador_tareas_ejecucion<='1' &
contador_tareas_ejecucion(2 downto 0);
                                        ld_id_tarea_3<='1';
                                        ent_id_tarea_3<=id;
                                elsif (contador_tareas_ejecucion(2)='0') then

        ent_contador_tareas_ejecucion<=contador_tareas_ejecucion(3) & '1' &
contador_tareas_ejecucion(1 downto 0);
                                        ld_id_tarea_2<='1';
                                        ent_id_tarea_2<=id;
                                elsif (contador_tareas_ejecucion(1)='0') then

        ent_contador_tareas_ejecucion<=contador_tareas_ejecucion(3 downto 2) & '1' &
contador_tareas_ejecucion(0);
                                        ld_id_tarea_1<='1';
                                        ent_id_tarea_1<=id;
                                elsif (contador_tareas_ejecucion(0)='0') then

        ent_contador_tareas_ejecucion<=contador_tareas_ejecucion(3 downto 1) & '1';
                                        ld_id_tarea_0<='1';
                                        ent_id_tarea_0<=id;
                                end if;
                                ent_inicio_cola<=inicio_cola+1;

                        when quitar_tarea=>
                                ld_contador_tareas_ejecucion<='1';
                                if (fin_tareas(3)='1') then
                                        ent_contador_tareas_ejecucion<='0' &
contador_tareas_ejecucion(2 downto 0);
```

```
                        end if;
                        if (fin_tareas(2)='1') then

        ent_contador_tareas_ejecucion<=contador_tareas_ejecucion(3) & '0' &
contador_tareas_ejecucion(1 downto 0);
                        end if;
                        if (fin_tareas(1)='1') then

        ent_contador_tareas_ejecucion<=contador_tareas_ejecucion(3 downto 2) & '0' &
contador_tareas_ejecucion(0);
                        end if;
                        if (fin_tareas(0)='1') then

        ent_contador_tareas_ejecucion<=contador_tareas_ejecucion(3 downto 1) & '0';
                        end if;

                end case;

                if (reset='1') then
                        estado <= inicio;
                elsif (clk'event and clk='1') then
                        case estado is
                                when inicio=>
                                        if (enable='1') then
                                                estado<= espera_hueco_y_tarea;
                                        end if;

                                when espera_hueco_y_tarea=>
                                        if ((contador_tareas_ejecucion/="1111")
                                                        and (inicio_cola /= fin_cola)) then
                                                estado<=lee_memoria_id;
                                        elsif (fin_tareas(3)='1' or fin_tareas(2)='1' or
                                                fin_tareas(1)='1' or fin_tareas(0)='1') then
                                                estado<=quitar_tarea;
                                        end if;


                                when lee_memoria_id=>
                                        if (fin_memoria='1') then
                                                estado<= carga_registro_id;
                                        end if;

                                when carga_registro_id=>
                                        estado<=lee_memoria_id_aux;

                                when lee_memoria_id_aux=>
                                        if (fin_memoria='1') then
                                                estado<= inc_reg_dir_mem_1_2;
                                        end if;

                                when inc_reg_dir_mem_1_2=>
                                        estado<=lee_tam_tarea_aux_2;

                                when lee_tam_tarea_aux_2=>
                                        if (fin_memoria='1') then
                                                estado<= inc_reg_dir_mem_1_1;
                                        end if;

                                when inc_reg_dir_mem_1_1=>
                                        estado<=lee_tam_tarea_aux_1;

                                when lee_tam_tarea_aux_1=>
                                        if (fin_memoria='1') then
                                                estado<= inc_reg_dir_mem_1_0;
                                        end if;

                                when inc_reg_dir_mem_1_0=>
                                        estado<=lee_tam_tarea_aux_0;

                                when lee_tam_tarea_aux_0=>
                                        if (fin_memoria='1') then
                                                estado<= inc_reg_dir_mem_2;
                                        end if;

                                when inc_reg_dir_mem_2=>
                                        if(id_aux=id) then
                                                estado<=carga_tarea;
```

```
                                        else
                                                estado<=inc_reg_dir_mem_3;

                                        end if;

                                when inc_reg_dir_mem_3=>
                                        estado<=lee_memoria_id_aux;

                                when carga_tarea=>
                                        if (fin_carga_tarea='1') then
                                                estado<=inc_contador_tareas_ejecucion;
                                        end if;

                                when inc_contador_tareas_ejecucion=>
                                        estado<=espera_hueco_y_tarea;

                                when quitar_tarea=>
                                        estado<=espera_hueco_y_tarea;

                        end case;
                end if;
        end process;

        z_inicio_cola<=inicio_cola;
        z_ejecutadas<=contador_tareas_ejecucion;
        z_id_tareas<=id_tarea_3&id_tarea_2&id_tarea_1&id_tarea_0;

        stado<=
                        "00000" when estado=inicio else
                        "00001" when estado=espera_hueco_y_tarea else
                        "00010" when estado=lee_memoria_id else
                        "00011" when estado=carga_registro_id else
                        "00100" when estado=lee_memoria_id_aux else
                        "00101" when estado=inc_reg_dir_mem_1_2 else
                        "00110" when estado=lee_tam_tarea_aux_2 else
                        "00111" when estado=inc_reg_dir_mem_1_1 else
                        "01000" when estado=lee_tam_tarea_aux_1 else
                        "01001" when estado=inc_reg_dir_mem_1_0 else
                        "01010" when estado=lee_tam_tarea_aux_0 else
                        "01011" when estado=inc_reg_dir_mem_2 else
                        "01100" when estado=inc_reg_dir_mem_3 else
                        "01101" when estado=carga_tarea else
                        "01111" when estado=inc_contador_tareas_ejecucion else
                        "10000";

end planificador_arch;
```