Sistemas Informáticos

Curso 2006-07

Planificador y asignador de tareas hardware sobre FPGAs

Esther Montero Lannegrand

Dirigido por:

Prof. Hortensia Mecha López

Dpto. de Arquitectura de Computadores y Automática

Facultad de Informática

Universidad Complutense de Madrid

TABLA DE CONTENIDOS

TABLA DE CONTENIDOS **2**

AUTORIZACIÓN **4**

RESUMEN DEL PROYECTO **5**

ABSTRACT **6**

OBJETIVO DEL PROYECTO **7**

CAPÍTULO 1: ENTORNO DE TRABAJO **8**

1.1 FIELD PROGRAMMABE GATE ARRAY (FPGA) 8
1.1.1 Aplicaciones *9*
1.1.2 Arquitectura *9*
1.1.3 Metodología de diseño *12*
1.2 XUP 12
1.3 VIRTEX II PRO 14
1.4 ANALIZADOR LÓGICO: 16702B LOGIC ANALYSIS SYSTEM 16
1.5 LENGUAJES DE DESCRIPCIÓN HARDWARE 18
1.5.1 VHDL *18*
1.5.2 Verilog *18*
1.6 MEMORIA DDR SDRAM 19
1.6.1 Operaciones *24*
1.6.1.1 Activación de banco / fila 24
1.6.1.2 Lecturas 24
1.6.1.3 Escrituras 25
1.6.2 Inicialización *25*

CAPÍTULO 2: HERRAMIENTAS SOFTWARE **27**

2.1 XILINX ISE 27
2.1.1 Creación de un proyecto *29*
2.1.2 Cuadro de Procesos y flujo de diseño *30*
2.2 MODELSIM 32
2.3 CHIPSCOPE PRO TOOLS 33
2.3.1 Chipscope Pro Core Inserter *33*
2.3.2 Chipscope Pro Analyzer *36*
2.4 MIG 007 38
2.5 IMPACT 43

CAPÍTULO 3: CONTROLADOR MEMORIA DDR SDRAM **44**

3.1 GENERACIÓN DEL CONTROLADOR CON LA HERRAMIENTA MEMORY INTERFACE GENERADOR 44
3.2 COMPROBACIÓN DEL FUNCIONAMIENTO DEL BANCO DE PRUEBAS Y PRIMERA CORRECCIÓN 48
3.3 SEGUNDA CORRECCIÓN 50
3.4 TERCERA CORRECCIÓN 51
3.5 DISEÑO DEL CONTROLADOR SIN BANCO DE PRUEBAS 52
3.5.1 Eliminación de ddr1_test_bench_64bit_00 y modificaciones en cmd_fsm *57*

CAPÍTULO 4: PLANIFICADOR DE TAREAS **59**

4.1 DESCRIPCIÓN 60
4.1.1 Esquema *61*
4.1.2 Protocolo de planificación *63*
4.1.3 Protocolo de ubicación *64*
4.1.4 Formato de las tareas *65*
4.2 ARQUITECTURA 66
4.2.1 Unidad de planificación *67*
4.2.1.1 Entadas y salidas 67
4.2.1.2 Unidad de control 68
4.2.1.3 Ruta de datos 70
4.2.2 Unidad de ejecución *74*
4.2.3 Cola fifo *75*

4.2.3.1Esquema 75
4.2.3.2 Primera fase de diseño 77
4.2.3.3 Segunda fase de diseño 77
4.2.3.4 Tercera fase de diseño 80
4.2.3.5 Cuarta fase de diseño 80
4.2.3.6 Quinta fase de diseño 81
4.2.3.7 Simulaciones 83
4.3 SIMULACIÓN 85

CAPÍTULO 5: DESARROLLO DEL PROYECTO 91

CAPÍTULO 6: CONCLUSIONES 93

APÉNDICES 94

A. CONTROLADOR VGA 94
A.1 Descripción *94*
A.1.1 Esquema 96
B. BIBLIOGRAFÍA 99
C. GLOSARIO 101
D. ÍNDICE DE FIGURAS 104
E. ÍNDICE DE TABLAS 108
F. PALABRAS CLAVE 109

AUTORIZACIÓN

La autora de este proyecto, Esther Montero Lannegrand, autoriza mediante este texto a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos no comerciales, y mencionando explícitamente a su autora tanto en la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo. Esther Montero Lannegrand.

RESUMEN DEL PROYECTO

Las mejoras tecnológicas alcanzadas para el hardware reconfigurable junto con la demanda de flexibilidad y velocidad de las aplicaciones hacen que los dispositivos reconfigurables en tiempo de ejecución se conviertan en una pieza interesante a la hora de diseñar sistemas de computación.

Este proyecto consiste en el desarrollo de un controlador de memoria DDR SDRAM y un planificador que se integrarán en un entorno de ejecución multitarea sobre dispositivos dinámicamente reconfigurables.

El área de la FPGA, el dispositivo reconfigurable utilizado, se divide en dos partes. La primera estará destinada a la implementación de todo el sistema y la segunda se denomina área de ejecución.

El área de ejecución estará dividida en cuatro particiones de diferente tamaño. Cada una de las particiones podrá ser reconfigurada, en tiempo de ejecución, independientemente del resto del dispositivo mediante la carga de un bitstream, es lo que se conoce como reconfiguración parcial.

El planificador decidirá que tarea se ejecutará en cada partición y el momento en que se lanza a ejecución. El algoritmo de planificación busca el máximo aprovechamiento del área de ejecución por lo que sigue una política de mejor ajuste.

En caso de que una tarea no se pueda ejecutar en una de las particiones, bien porque tiene mayor tamaño o bien por no cumplir las restricciones temporales, se podrá ejecutar en más de una.

El mapa de bits a cargar en cada una de las particiones se encontrará un una memoria DDR SDRAM al que se accederá mediante el controlador diseñado.

Todos los resultados se podrán visualizar a través de un monitor.

ABSTRACT

Recent technology improvements on the field of Reconfigurable HW, together with the increasing demands for flexibility and speed coming from the latest user applications make reconfigurable devices an interesting element in the design of the state-of-the-art computing systems.

This project consists on the development of a DDR SDRAM memory controller and a HW task-scheduler, and the integration of both modules into a multitasking execution framework targeted for reconfigurable devices.

The area of the FPGA, the reconfigurable device, is splitted into two different parts. The first one will contain the implementation of the complete system, while the second one is called “execution area”.

The execution area is splitted into four partitions of different sizes. Each of them can be reconfigured at runtime independently from the others by loading a bitstream. This is called partial reconfiguration.

The scheduler assigns one or several partitions to each task, and decides also when the task will be executed. The algorithm used to allocate the tasks is “best-fit”, always looking for the best area utilization.

In the cases when one partition is not enough to execute a task (by not meeting timing constrains or by being too big in size), more than one partition could be assigned to one single task.

The bitmap to be loaded in each partition will be stored into a DDR SDRAM memory, accessed through our custom-designed controller.

All the results can be visualized in a standard computer display.

OBJETIVO DEL PROYECTO

El objetivo del proyecto es la ampliación sistema operativo para la ejecución de tareas sobre dispositivos dinámicamente reconfigurables.

Este entorno es íntegramente hardware y está formado por los módulos que se indican en la figura 1.

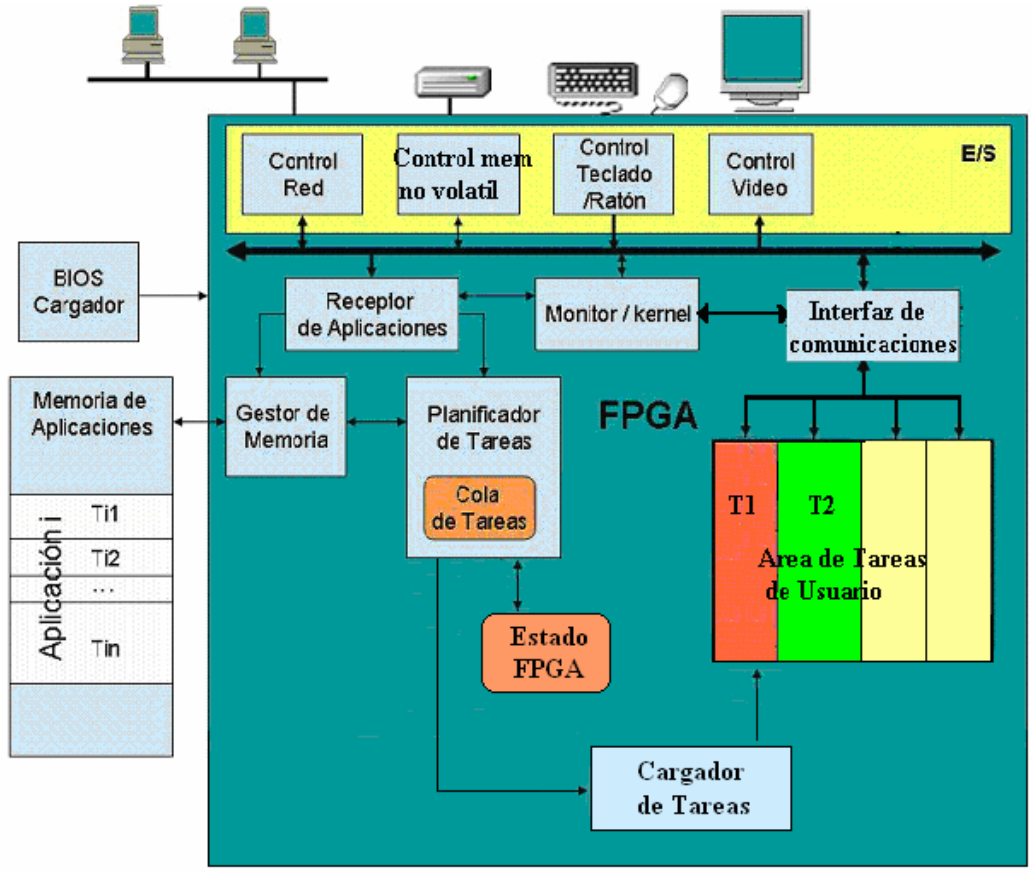


Figura 1: Esquema general del sistema

Dada la complejidad del sistema global, en este proyecto nos concentraremos en los siguientes módulos:

- Controlador de memoria DDR SDRAM: el cometido de este módulo es la comunicación del dispositivo de memoria volátil y el entorno. La comunicación a realizar será de entrada/salida.

- Planificador de tareas: con este módulo el sistema gestionará las tareas recibidas y decidirá en que parte del área de ejecución se ejecutarán en función de unas restricciones espaciales y temporales. El planificador considerará el área de tareas de usuario dividida en cuatro partes con igual longitud y diferente ancho.

- Controlador monitor VGA: a través de este controlador se mostrará por pantalla información sobre las tareas que se realicen.

CAPÍTULO 1: ENTORNO DE TRABAJO

Los módulos desarrollados en este proyecto están diseñados para funcionar sobre un dispositivo dinámicamente reconfigurable, en concreto sobre una FPGA de la familia Virtex II Pro empotrada en la placa de desarrollo XUP Virtex II Pro Development System.

Los lenguajes de descripción hardware utilizados han sido VHDL, en la implementación del controlador de memoria y el planificador, y Verilog, para el controlador VGA.

Para verificar el sistema se ha utilizado el analizador lógico 16702B de Agilent.

En cuanto al software, se utilizado Xilinx ISE para el diseño y síntesis, Modelsim para realizar simulaciones, Chipscope para verificar el sistema e IMPACT para configurar la FPGA. Todas estas herramientas se explican en detalle en el capítulo dos.

1.1 Field Programmabe Gate Array (FPGA)

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

Las FPGAs son circuitos electrónicos prefabricados reconfigurables que contienen bloques de lógica cuya funcionalidad e interconexión se puede programar. Esto nos permite crear nuevos circuitos que se comportarán como nosotros queramos. Son típicamente volátiles.

Las FPGAs aparecieron como una evolución de los CPLDs (Complex Programmable Logic Device o dispositivo lógico complejo programable). Las ventajas que tienen frente a éstos son su mayor densidad de puertas, la flexibilidad de su arquitectura y que habitualmente contienen funciones de alto nivel, como sumadores o multiplicadores, y bloques de memoria empotrados en la matriz de interconexionado.

La ventaja de la FPGA es que al diseñar un sistema no es necesaria su fabricación, sino que se puede realizar programando correctamente el dispositivo por lo que los costes se reducen.

Una FPGA es programable a nivel hardware aunque actualmente existen dispositivos con varios procesadores capaces de ejecutar programas especificados con lenguajes de alto nivel. Debido a esto proporciona las ventajas de un procesador de propósito general y un circuito especializado.

La utilización de FPGAs se está extendiendo en todos los ámbitos del diseño de sistemas digitales debido a que los costes y el tiempo de desarrollo son mucho menores. En muchos casos ni siquiera se procede a la posterior fabricación del circuito final sino que se utiliza directamente la FPGA.

Otro dispositivo empleado para el diseño de sistemas es el ASIC (Application-Specific Integrated Circuit). Son circuitos integrados para aplicaciones específicas, es decir, están hechos a la medida para un uso en particular, no para propósito general. La diferencia fundamental entre FPGA y ASIC reside en el nivel de personalización de la aplicación. Mientras que a nivel de ASIC el diseñador interviene en varias etapas del

proceso tecnológico, a nivel FPGA solamente se interviene en la configuración del sistema, de ahí que el diseño y depuración sea más inmediato y sencillo con una FPGA. Una desventaja de ASIC consiste en que si se produce un error durante la etapa de diseño del sistema no se puede corregir tras la fabricación, lo que conlleva rediseñar y volver a fabricar nuevamente con el consiguiente aumento del gasto y tiempo necesario [1] [2].

1.1.1 Aplicaciones

Debido a la flexibilidad, capacidad de procesado y a la rápida implementación de sistemas sobre FPGA se emplea habitualmente en:

- Procesamiento de señales digitales.
- Módulos de comunicación.
- Sistemas aeronáuticos.
- Formación en el diseño de sistemas hardware.
- Simulación y depuración en el diseño de microprocesadores y microcontroladores.

1.1.2 Arquitectura

La arquitectura básica de una FPGA consiste en una matriz de CLBs (Configurable Logic Block o bloques lógicos programables) y canales de comunicación.

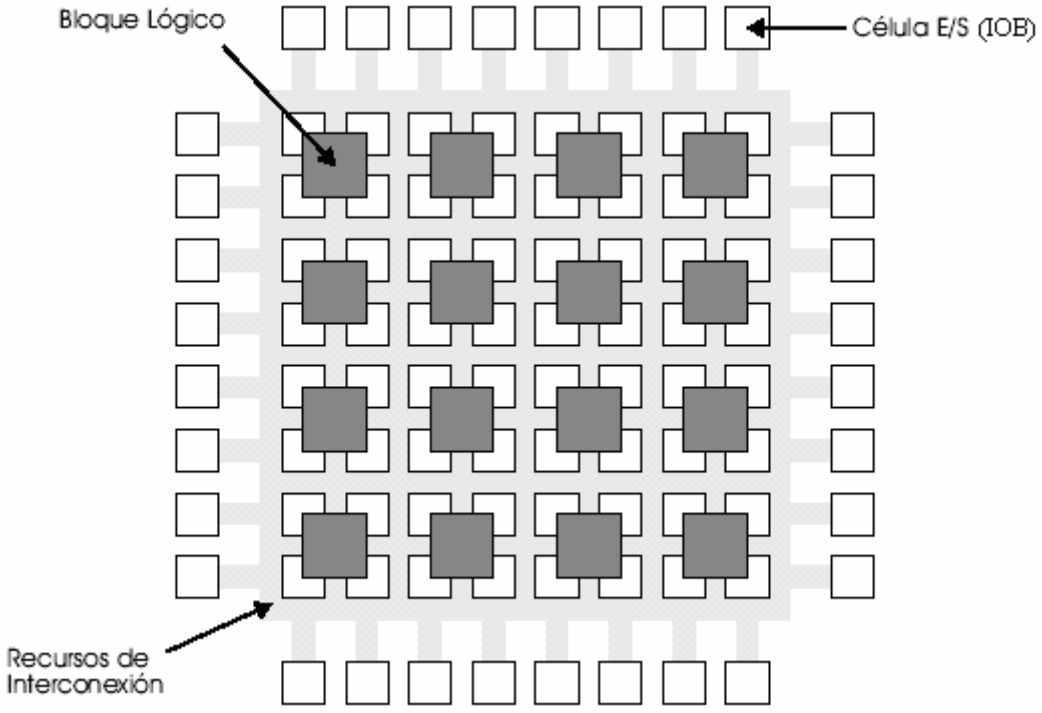


Figura 2: Estructura interna de una FPGA [3]

Los componentes básicos, se muestran en la figura 2, son:

- IOB o células de entrada salida. Comunican el dispositivo con el exterior y están situados rodeando la matriz de bloques lógicos, próximos a las patillas del chip. Se puede programar su forma de trabajo para que se comporten como puertos de entrada, salida o entrada-salida.

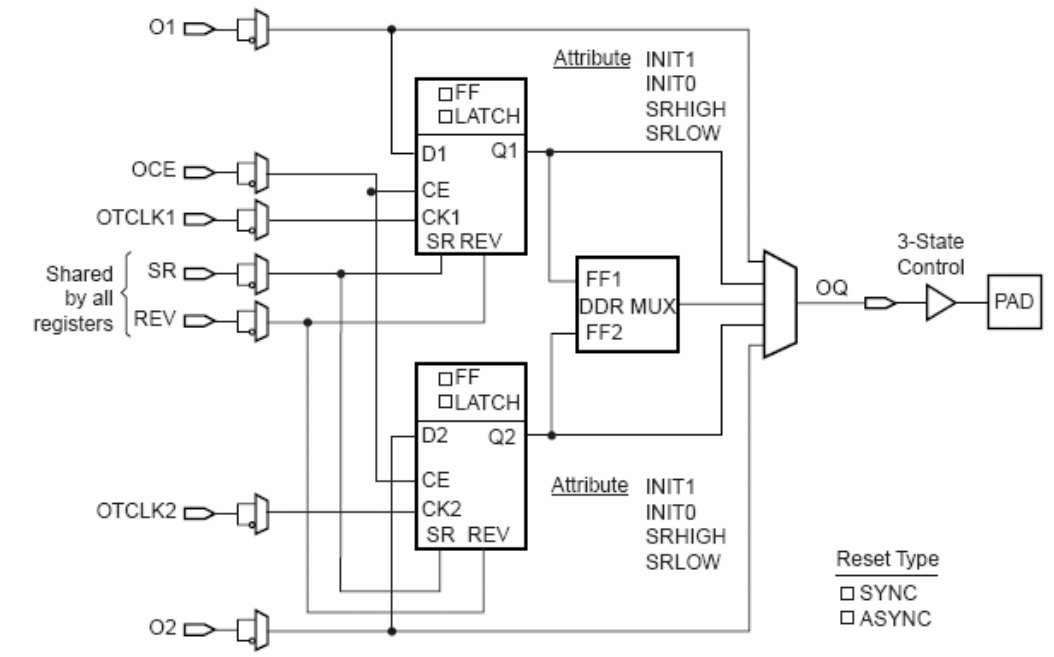


Figura 3: Estructura de un IOB (Xilinx V2P) [4]

- Bloques lógicos configurables. Su contenido y estructura puede variar su complejidad, yendo desde una simple puerta lógica a un CBL. Suelen incluir biestables para facilitar la implementación de circuitos secuenciales.

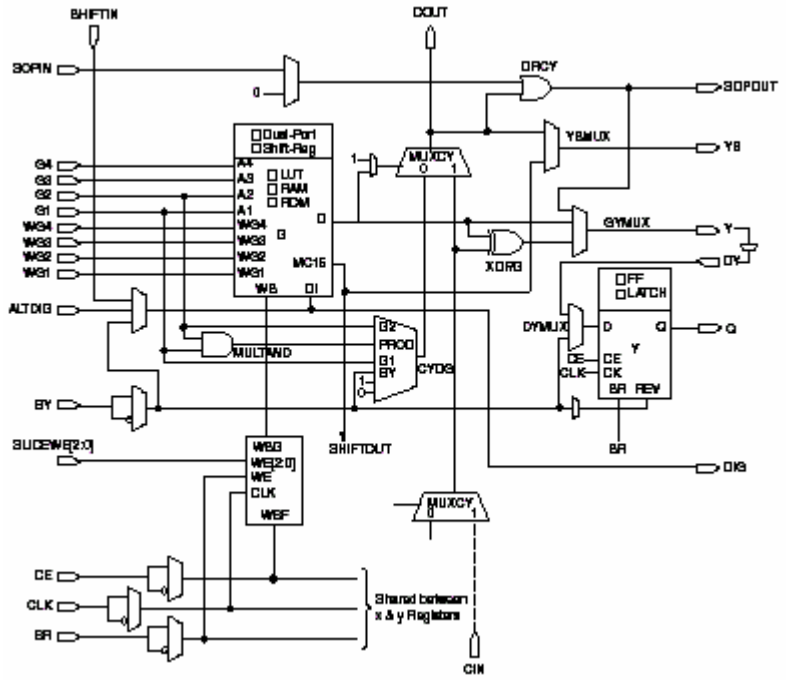


Figura 4: Estructura de un CLB (Xilinx V2P) [4]

Un bloque lógico que suele contener es una tabla de funciones lógicas de cuatro entradas (LUT).

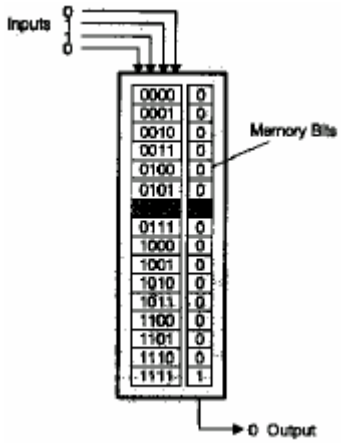


Figura 5: Estructura de una LUT [5]

- Bloques de interconexión. Está formado por un canal de rutado vertical y otro horizontal a los que se puede conectar cualquier CLB próximo mediante un punto de interconexión. En los cruces de los dos canales se encuentras las matrices de conmutación.

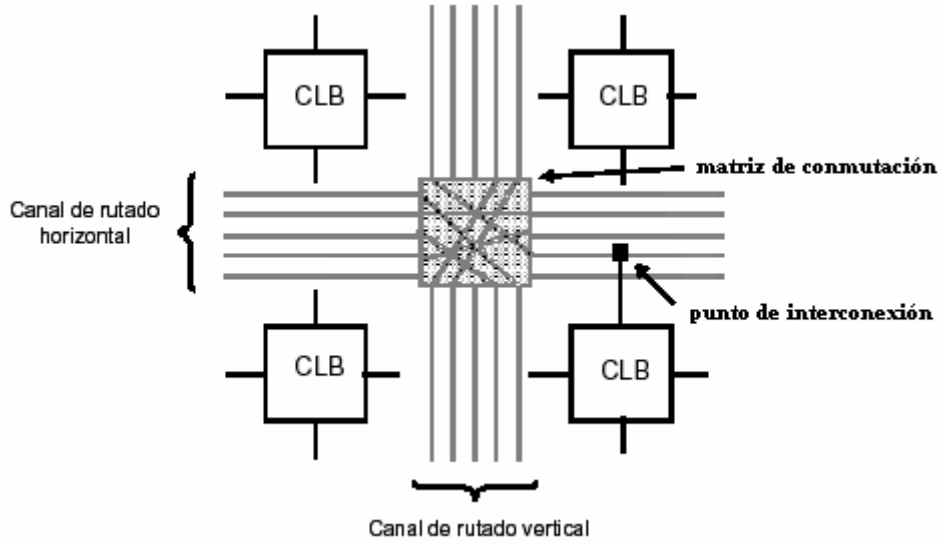


Figura 6: Estructura de interconexión en la FPGA [5]

- Memoria de configuración. Se carga con el mapa de bits para configurar los bloques y conectarlos. Se reescribirá cada vez que se programe el dispositivo.

Además de los componentes descritos (estructura básica), en función del modelo y del fabricante, pueden aparecer nuevos módulos y presentar otra arquitectura, por ejemplo varios planos con CLBs.

1.1.3 Metodología de diseño

Los diseños se realizan programando adecuadamente los CLBs, IOBs y bloques de interconexión. Esta configuración se realiza mediante la carga del mapa de bits que describe nuestro sistema y especifica la función de cada CLB, el modo de trabajo de cada IOB y las conexiones necesarias.

Para la descripción del sistema se utilizan lenguajes HDL (Hardware Description Language) como son VHDL, Verilog o ABEL.

Cada fabricante proporciona un conjunto de herramientas para la programación y depuración de los dispositivos.

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

1.2 XUP

El dispositivo utilizado ha sido una FPGA XC2VP30, de la familia Virtex II PRO montada sobre una placa de desarrollo XUP Virtex II Pro Development System (que se ve en la figura 7).

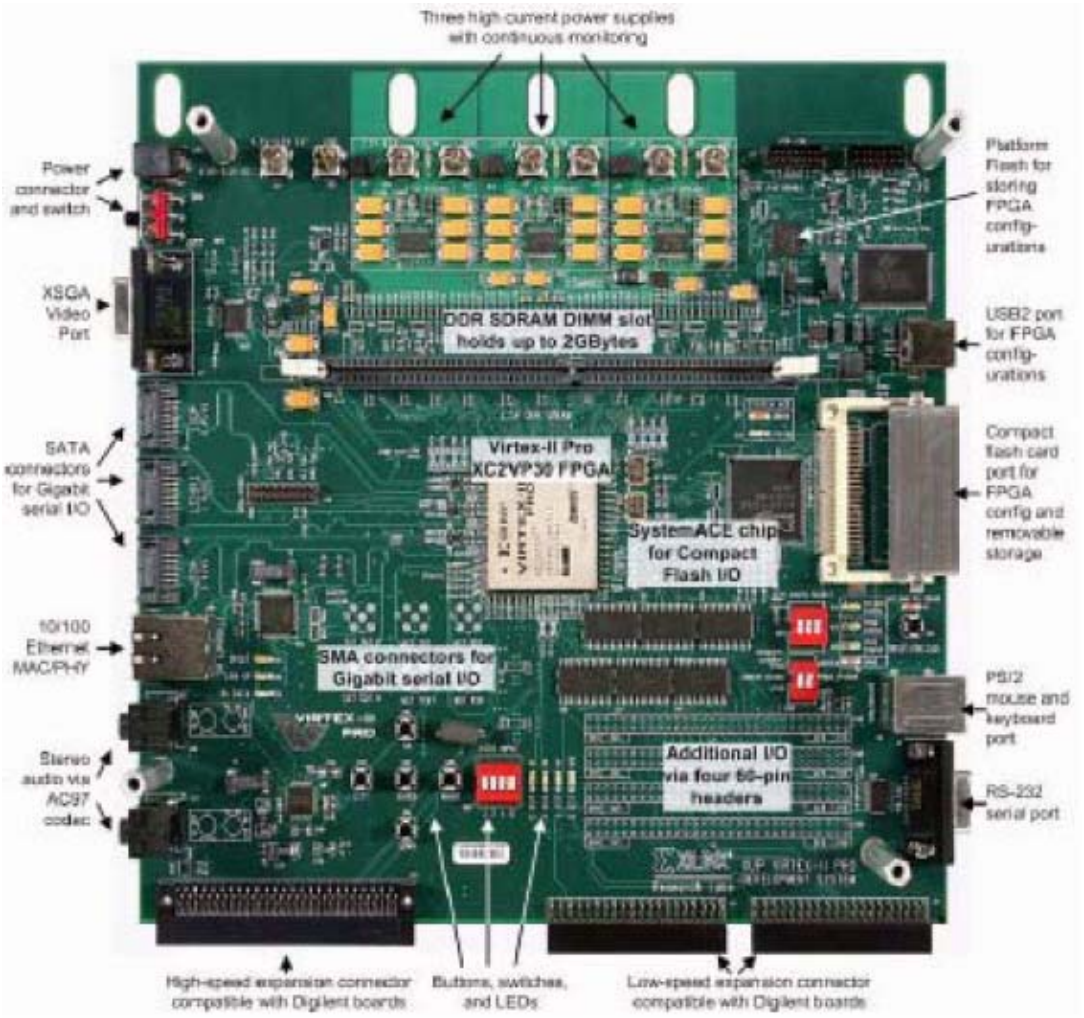


Figura 7: foto Xup [6]

Aporta algunos componentes periféricos como una memoria DDR RAM (pude ser desde 64MB a 2 GB), puerto ethernet 10/100, codecs de audio y video, pins de entrada y salida para una tarjeta compact flash, 2 puertos PS/2, 4 switches, 4 leds, puerto USB, puerto XSGA video y conectores SATA entre otras cosas.

La forma en que la FPGA queda conectada en la placa de dearrollo se muestra en la figura 8.

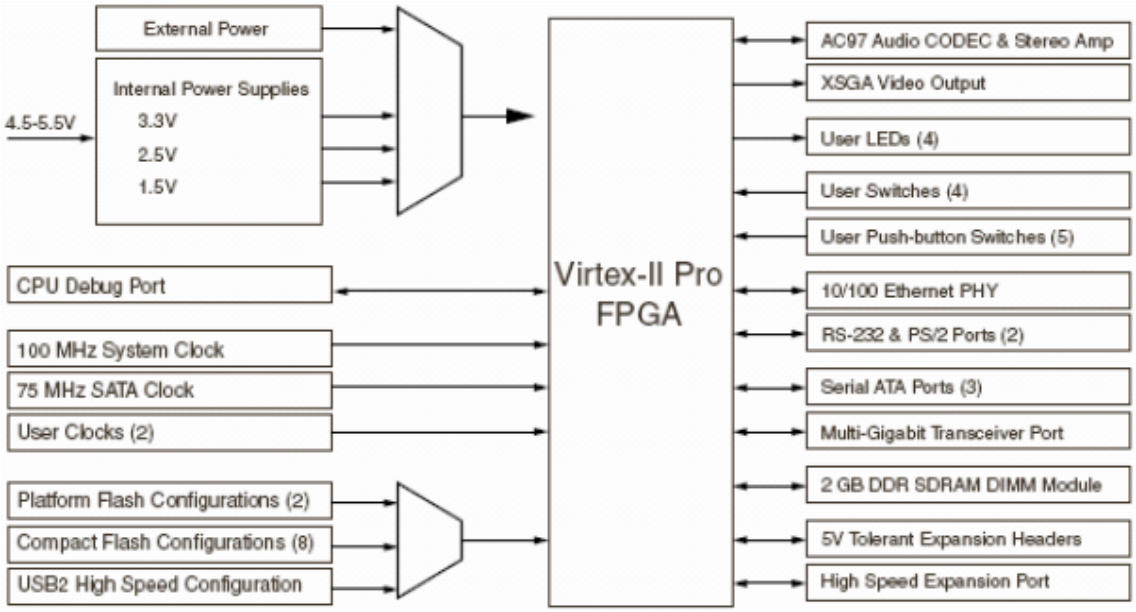


Figura 8: conexiones Virtex II Pro con la XUP [6]

1.3 Virtex II Pro

La familia Virtex II Pro son PFGAs de alto rendimiento que incorporan un núcleo de IBM PowerPC, todo integrado en el mismo chip.

Fue la primera plataforma FPGA capaz de implementar las funciones de alto rendimiento de un system-on-chip, que, anteriormente, era exclusivo de los ASICs, con la flexibilidad y los bajos costes de la lógica programable. Esta familia de Xilinx se ha utlizado para el desarrollo de arquitecturas de redes, sistemas empotrados y de procesamiento de señales. Además su arquitectura constituye la próxima generación de estándares de conectividad para el desarrollo de sistemas complejos codesarrollados en hardware y software [4]**.**

Veamos su estructura interna para entender mejor las posibilidades que ofrece este dispositivo:

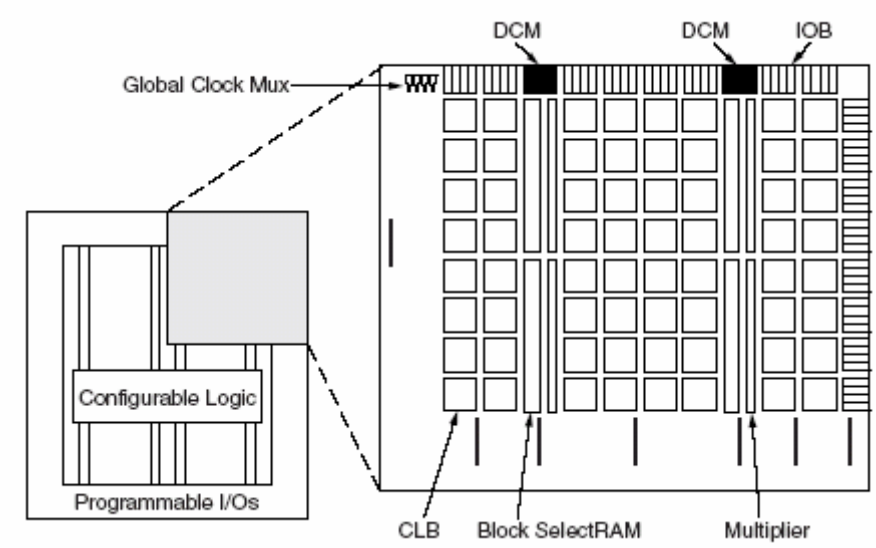


Figura 9: Estructura interna de Virtex II Pro

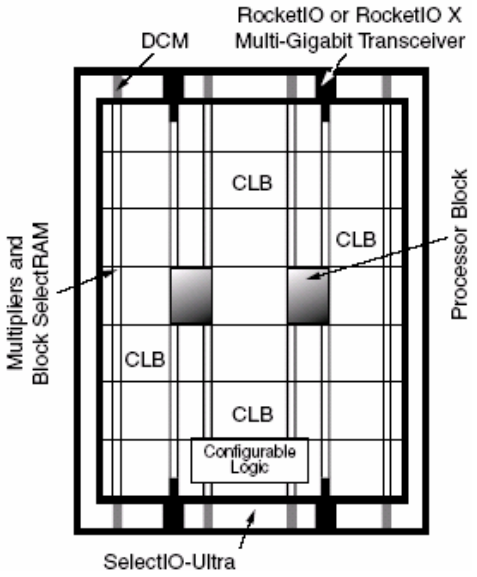


Figura 10: Pocesadors empotrados en Virtex II Pro

Los componentes que se aprecian en las figuras 9 y 10 son los siguientes:

- Bloques lógicos configurables (CLBs).
- Bloques SelectRAM. Provee un amplio número de elementos de almacenamiento RAM de 18 KB de doble puerto.
- Bloques multiplicadores de 18 x 18 bits.
- Celdas de entrada/salida (IOBs).
- Procesadores. La Vrtex II Pro cuenta con dos procesadores risc PowerPC 405. Es el procesador empotrado de mayor rendimiento de los que hay disponibles en una FPGA.
- DCM (Digital Clock Manager). Este módulo se encarga de la gestión del reloj y realiza tareas como multiplicar o dividir su frecuencia, eliminar el skew dentro del dispositivo o en los componentes externos, asegurar una señal de reloj limpia, desfasar el reloj una cantidad fija o incremental o devolver una señal espejo del reloj [7]. Las funciones realizadas y las señales generadas se pueden ver en la figura 11.

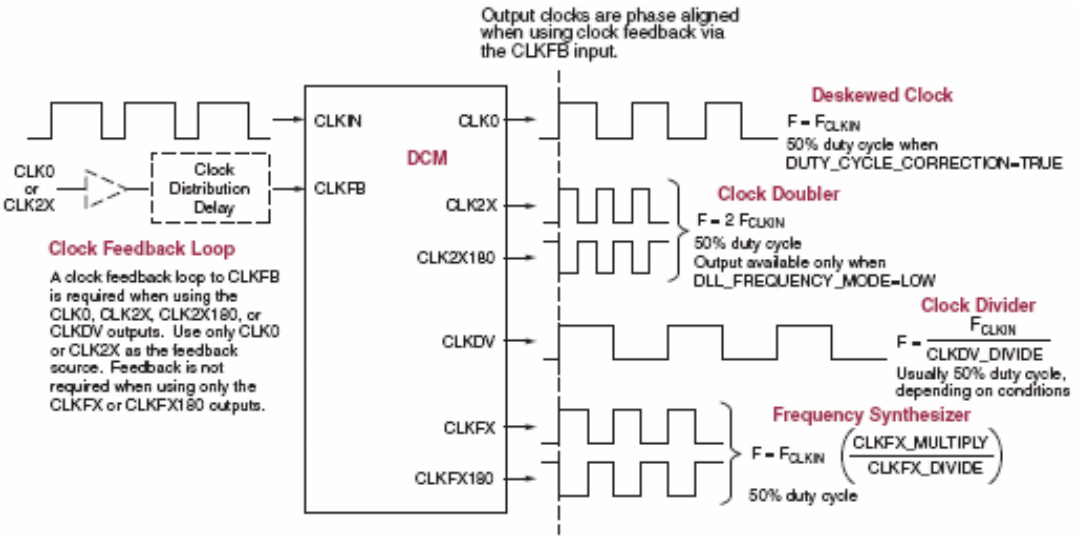


Figura 11: Relojes generados por el DCM [7]

Una característica interesante de esta FPGA es que permite reconfiguración parcial. Este tipo de reconfiguración es útil para aplicaciones que requieren cargar diferentes diseños en la misma área del dispositivo o la flexibilidad para cambiar parte de un diseño, en tiempo de ejecución, sin necesidad de resetear o reconfigurar el dispositivo al completo. De ahí que sea idóneo para la implementación de nuestro entorno de ejecución multitarea.

La reconfiguración parcial en la familia de FPGAs Virtex II Pro se realiza mediante una interfaz interna de reconfiguración conocida como ICAP. Anteriormente se realizaba mediante el modo esclavo SelectMap o el modo JTAG que necesitaban un controlador externo para enviar los datos de reconfiguración [8].

Datos técnicos de la placa: tiene 13969 slices, una matriz de CLBs de 80x46, RAM distribuída de 428 KB, 136 bloques multiplicadores, 136 BRAMs de 18 KB, 8 DCMs, 2 procesadores risc PowerPC y 8 multi-gigabit transmisores-receptores [6].

1.4 Analizador lógico: 16702B Logic Analysis System

Esta herramienta proporciona una interfaz, similar a la presentada por los simuladores software, para visualizar el valor de señales dentro de la FPGA. De este modo proporciona una forma sencilla de depuración.

Este modelo en particular cuenta con un monitor y una pantalla táctil de 800x600. Gracias al tamaño de la pantalla se pueden visualizar más señales de forma simultánea. También cuenta con un teclado y un ratón que facilita el acceso a los diferentes menús.

Figura 12: Foto del analizador lógico 16702B [9]

Posee un disco duro de 18GB, 125 MB de memoria RAM y un lector CD-ROM [9].

El entorno cuenta con un sistema de ventanas (figura 13) muy similar al de Windows que permite configurar el dispositivo y el manejo de sus distintas funciones de forma sencilla. Tiene acceso directo a las guías de ayuda del fabricante.

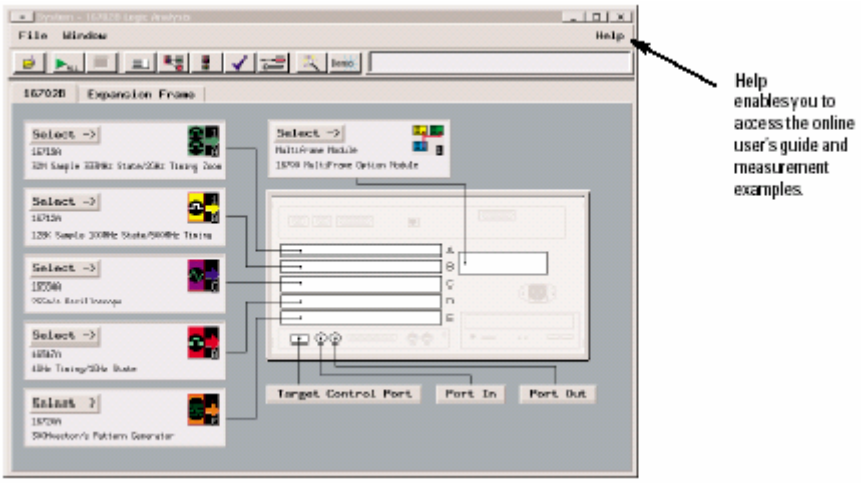


Figura 13: Sistema de ventanas del analizador 16702B [9]

Permite el análisis del estado y temporal de un sistema. También se comporta como un osciloscopio, generador de patrones y herramienta de post-procesado. En la figura 14 se aprecian los diferentes modos de trabajo que permite.

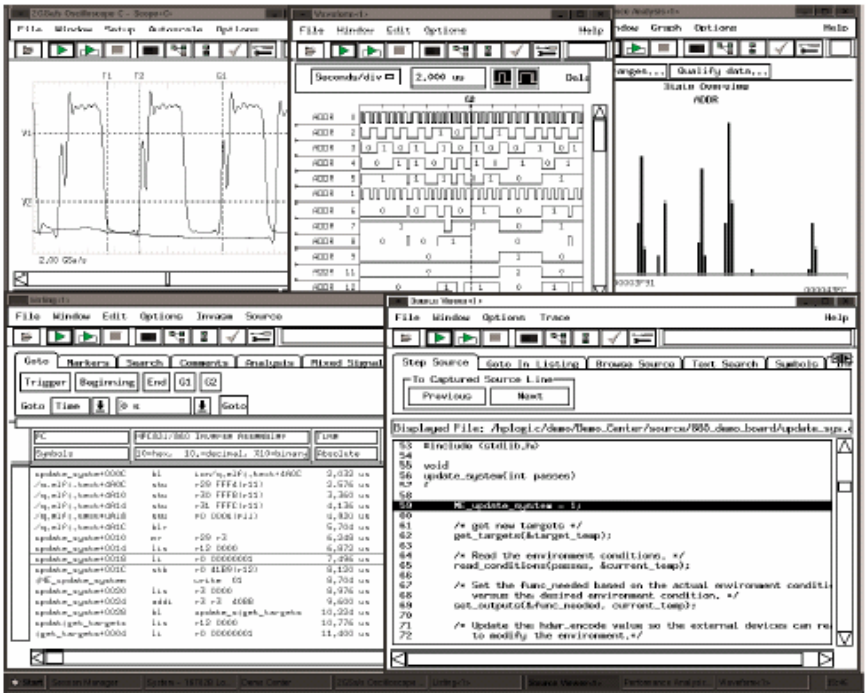


Figura 14: Sistema multiventana del analizador 16702B [9]

El principal uso que le hemos dado durante el desarrollo del proyecto ha sido el análisis de la forma de onda y estado de las señales presentes en la FPGA. En la figura 15 se ve la interfaz de análisis lógico.

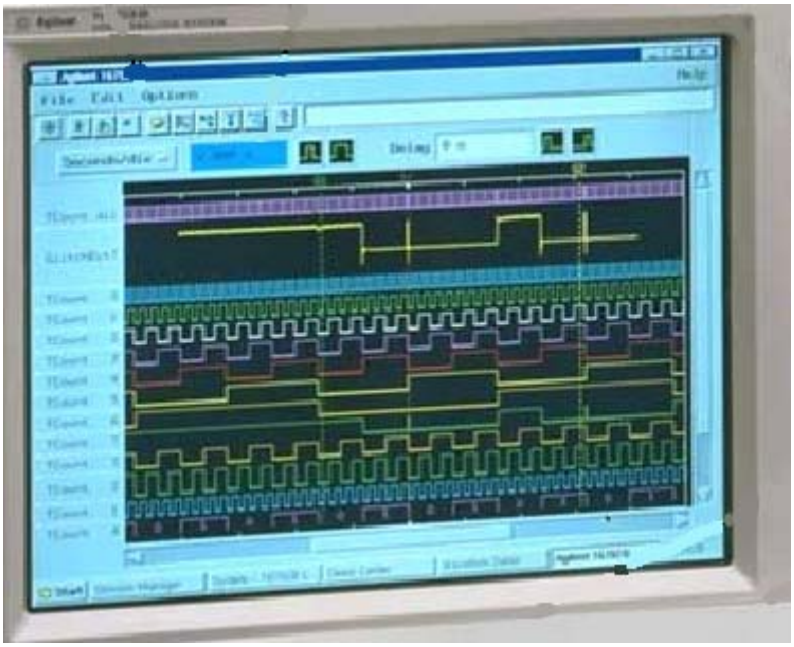


Figura 15: Análisis de onda del analizador 16702B [9]

1.5 Lenguajes de descripción hardware

El código utilizado en el diseño e implementación de los diferentes módulos del proyecto ha sido principalmente VHDL y, en menor medida, Verilog.

1.5.1 VHDL

VHDL es el acrónimo de VHSIC Hardware Description Language. Se trata de un lenguaje de descripción hardware que nació de la colaboración de la industria privada y las universidades. Fue estandarizado por IEEE a finales de los ochenta.

La independencia en la metodología de diseño, su capacidad descriptiva en múltiples dominios y niveles de abstracción, su versatilidad para la descripción de sistemas complejos, su posibilidad de reutilización y en definitiva la independencia de que goza con respecto de los fabricantes, han hecho que VHDL se convierta en el lenguaje de descripción hardware más importante.

Al estar creado específicamente para el diseño hardware permite implementar multitud de circuitos lógicos, tanto combinacionales como secuenciales, y de muy variada complejidad, desde un sencillo biestable a una CPU.

VHDL nos permite varias formas de describir un sistema:

- Funcional: se describe la forma en que comporta el circuito. La descripción se realiza de forma secuencial. Esta es la metodología más parecida a los lenguajes de programación software.
- Flujo de datos: define asignaciones concurrentes de señales.
- Estructural: se describe el circuito con instancias de componentes. Estos componentes se conectan entre sí formando un diseño jerárquico.
- Mixta: combinaciones de los modos anteriores.

1.5.2 Verilog

Es un lenguaje de descripción hardware usado para modelar sistemas electrónicos. Soporta el diseño, prueba e implementación de circuitos analógicos, digitales y de señal mixta a diferentes niveles de abstracción. Fue estandarizado por IEEE a mediados de la década de los noventa.

Sus creadores pretendían hacer un lenguaje próximo a C para que la sintaxis fuera familiar a los ingenieros y facilitar la aceptación del lenguaje. Tiene muchas de las estructuras de los lenguajes software como if o while.

Un diseño en Verilog consiste en una jerarquía de módulos que son definidos como conjuntos de puertos de entrada, salida y bidireccionales. Internamente, un módulo contiene una lista de cables y registros. Tiene sentencias secuenciales y concurrentes que describen el comportamiento del sistema.

Al igual que VHDL, tiene un conjunto de sentencias sintetizables que permite la síntesis del código para la generación de un mapa de bits y el volcado a una FPGA.

1.6 Memoria DDR SDRAM

La memoria DDR (Double Data Rate) es un nuevo tipo de memoria que funciona al doble de velocidad respecto a una SDRAM normal. Una memoria DDR mejora un 15% el rendimiento respecto a una memoria normal SDRAM [10]. Las especificaciones estándar serían las siguientes:

Nombre estándar	Velocidad del reloj	Tiempo entre señales	Velocidad del reloj de E/S	Datos transferidos por segundo	Nombre del módulo	Máxima capacidad de transferencia
DDR-200	100 MHz	10 ns	100 MHz	200 Million	PC-1600	1.600 GB/s
DDR-266	133 MHz	7.5 ns	133 MHz	266 Million	PC-2100	2.133 GB/s
DDR-333	166 MHz	6 ns	166 MHz	333 Million	PC-2700	2.667 GB/s
DDR-400	200 MHz	5 ns	200 MHz	400 Million	PC-3200	3.200 GB/s
DDR-466	233 MHz	4.2 ns	233 MHz	466 Million	PC-3700	3.700 GB/s
DDR-500	250 MHz	4 ns	233 MHz	500 Million	PC-4000	4.000 GB/s
DDR-533	266 MHz	3.7 ns	266 MHz	533 Million	PC-4200	4.200 GB/s
DDR-600	300 MHz	3.3 ns	300 MHz	600 Million	PC-4800	4.800 GB/s

Tabla 1: Chips y módulos de memoria DDR [10]

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. 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, 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 las diferentes operaciones.

Una característica importante en estas memorias es la existencia de dos relojes diferenciales para su funcionamiento (CLK y CLK_Z). Con éstos relojes generamos una única señal de reloj cuyos flacos de subida coincidirán con el flaco de subida de CLK o con el flanco de bajada de CLK_Z. Por lo tanto sólo trabajamos con el flanco de subida del reloj generado internamente.

Tanto la lectura como la escritura se realizan en ráfagas, de tamaño programable. El tamaño de la ráfaga puede tomar valores dos, cuatro u ocho palabras de 64 bits. La operación se realiza a partir de la dirección dada y se realizan tantos accesos como indique el tamaño de ráfaga fijado.

A continuación se muestra el módulo de una memoria DDR SDRAM con las señales de entrada y salida correspondientes:

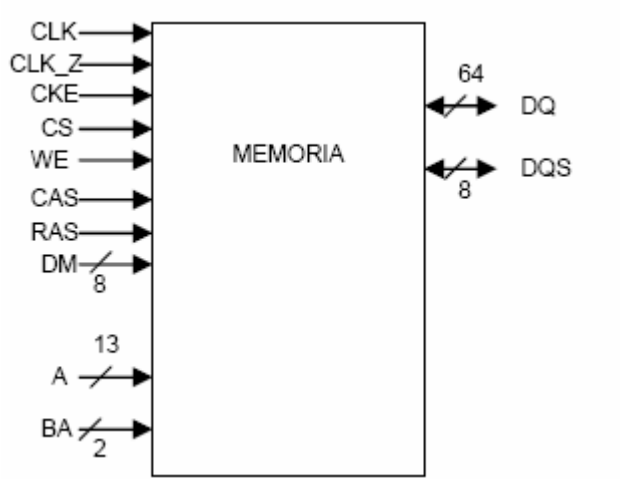


Figura 16: Esquema general de un módulo DDR SDRAM

En las siguientes tablas se explican con más detalles las señales.

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 al flanco con los datos para las lecturas y alineada en el centro para las escrituras.

Tabla 2: Entrada/Salida de un módulo DDR SDRAM

Significado de las señales de entrada y salida del módulo:

Entrada	Descripción
CLK/CLK_Z	- Entradas de reloj diferenciales con las que se genera el reloj interno. - Las señales de control y las direcciones son registradas en los flancos de subida de CLK y en los de bajada de CLK_Z.
CKE	- Capacitación del reloj. - Decapacita a baja.
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. - Bus de 8 bits.
A [12:0]	- Entradas de dirección. Es un bus de 13 bits. - Dependiendo de la operación puede ser: - 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 3: Entradas de un módulo DDR SDRAM

Las memorias DDR tienen dos registros, el *mode register* y el extended *mode register*.

- Registro de modo: 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 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 length determina el máximo número de columnas a las que se puede acceder durante un comando de lectura o escritura (2, 4 ó 8). 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 de LOAD MODE REGISTER y puede ser modo de operación normal o modo de operación reset DLL. La figura 17 muestra el significado de los bits que forman el registro y sus posibles valores.

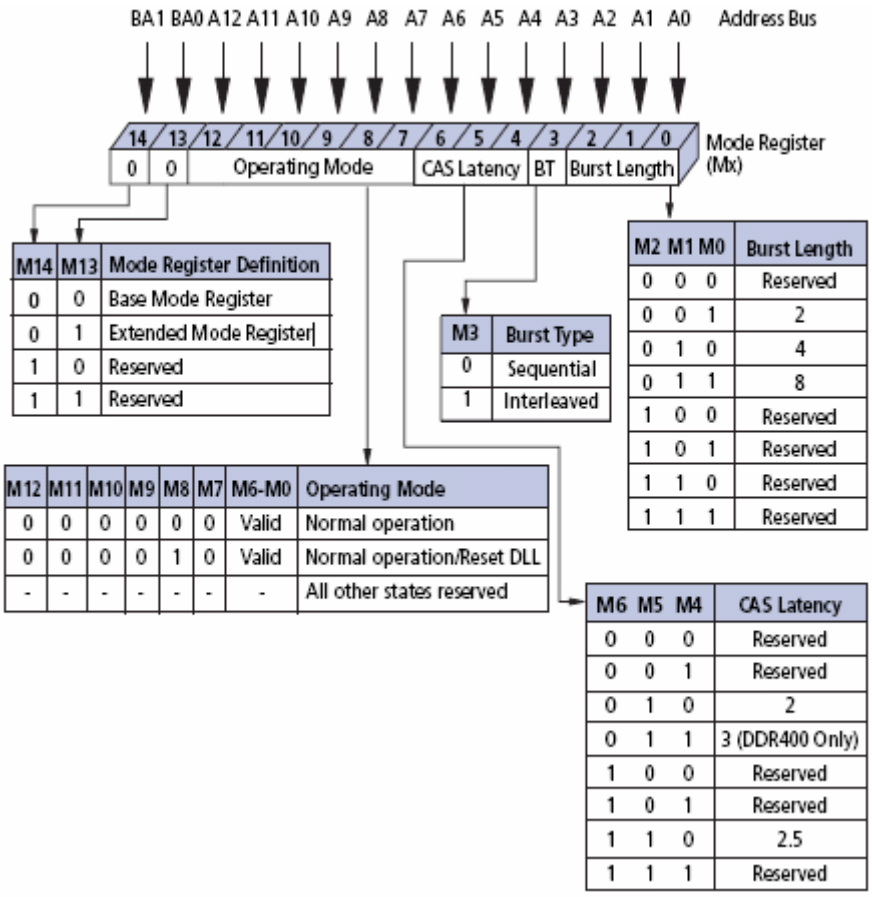


Figura 17: Registro de modo [11]

- Registro de modo extendido: Controla funciones adicionales a las del mode register. Entre ellas cabe destacar la de la capacitación o descapacitación del DLL. 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. La figura 18 muestra los valores posibles para cada campo.

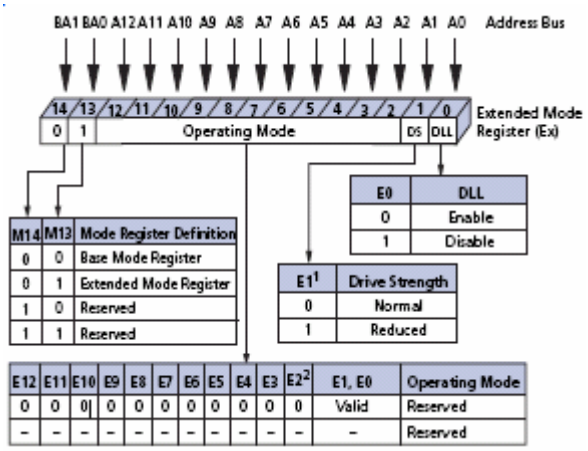


Figura 18: Registro de modo extendido [11]

Para poder realizar todas las acciones descritas tanto de configuración como de funcionamiento tenemos los siguientes comandos:

- Deselect: Evita que otros comandos entren en ejecución aunque no abortan los que ya se estén ejecutando.
- Nop: Pasa un ciclo de reloj sin que se realice ningún trabajo.
- Load_mode_register: Permite la carga de datos en el mode register o en el extended mode register.
- Active: Activa una fila de un banco particular en el que se vaya a realizar una secuencia de accesos.
- Read: Inicia una operación de lectura sobre una fila activa de un banco.
- Write: Inicia una operación de escritura sobre una fila activa de un banco.
- Precharge: Desactiva una fila abierta o activada de un banco particular o de todos los bancos.
- Auto-precharge: Permite realizar precargas sobre un mismo banco pero sin requerir ningún comando explicito.
- Burst terminate: Finaliza la lectura de una ráfaga.
- Auto refresh: Se lanza cada vez que se quiera realizar el refresco de la memoria.
- Self refresh: Este comando puede ser usado para mantener datos en la memoria incluso si se resetea.

Name (Function)	CS#	RAS#	CAS#	WE#	Address	Notes
DESELECT (NOP)	H	X	X	X	X	1
NO OPERATION (NOP)	L	H	H	H	X	1
ACTIVE (Select bank and activate row)	L	L	H	H	Bank/Row	2
READ (Select bank and column, and start READ burst)	L	H	L	H	Bank/Col	3
WRITE (Select bank and column, and start WRITE burst)	L	H	L	L	Bank/Col	3
BURST TERMINATE	L	H	H	L	X	4
PRECHARGE (Deactivate row in bank or banks)	L	L	H	L	Code	5
AUTO REFRESH or SELF REFRESH (Enter self refresh mode)	L	L	L	H	X	6, 7
LOAD MODE REGISTER	L	L	L	L	Op-Code	8

Tabla 4: Tabla de verdad de los comandos DDR [11]

1.6.1 Operaciones

1.6.1.1 Activación de banco / fila

Antes de realizar una lectura o una escritura sobre una fila del banco de registros, ésta deber ser "abierta". Esta operación se puede realizar con un comando *active*. Una vez hecho esto ya se podrá acceder a esa fila.

Para poder abrir otra fila dentro del mismo banco es necesario cerrar la fila abierta previamente.

1.6.1.2 Lectura

Para realizar una lectura es necesario ejecutar el comando *read*. Los datos de salida (DQ) sólo serán válidos una vez que haya transcurrido el tiempo de CAS Latency desde que se lanzo el comando *read*. A partir de ese momento llegará un nuevo dato hasta agotar la ráfaga. Se puede acabar la lectura de la ráfaga con el comando *burst terminate* aunque esta aún no hubiera concluido. En la figura 19 se muestra el protocolo de lectura.

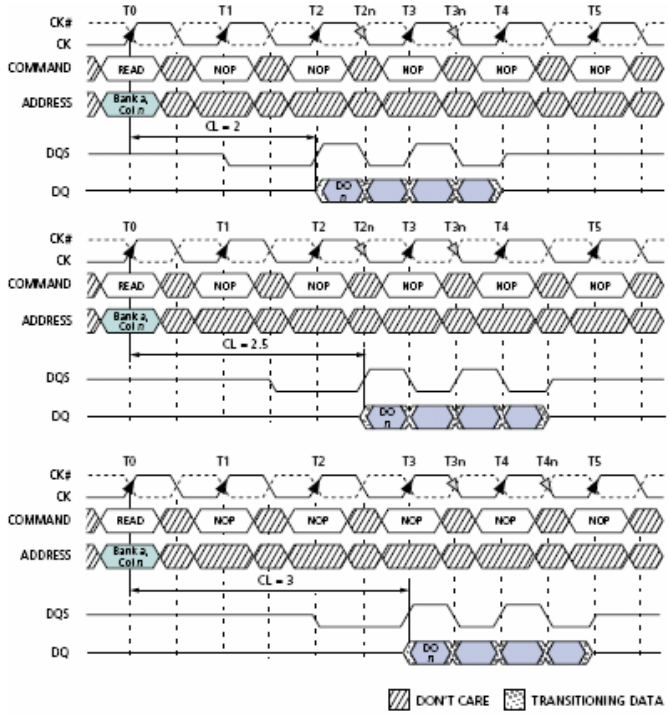


Figura 19: Ráfaga de lectura [11]

1.6.1.3 Escritura

Para realizar una escritura es necesario ejecutar el comando *write* Durante la escritura, 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 hasta concluir la ráfaga. Con el comando *burst terminate* se puede concluir la escritura de la ráfaga en cualquier momento como con la lectura. El protocolo de escritura se muestra en la figura 20.

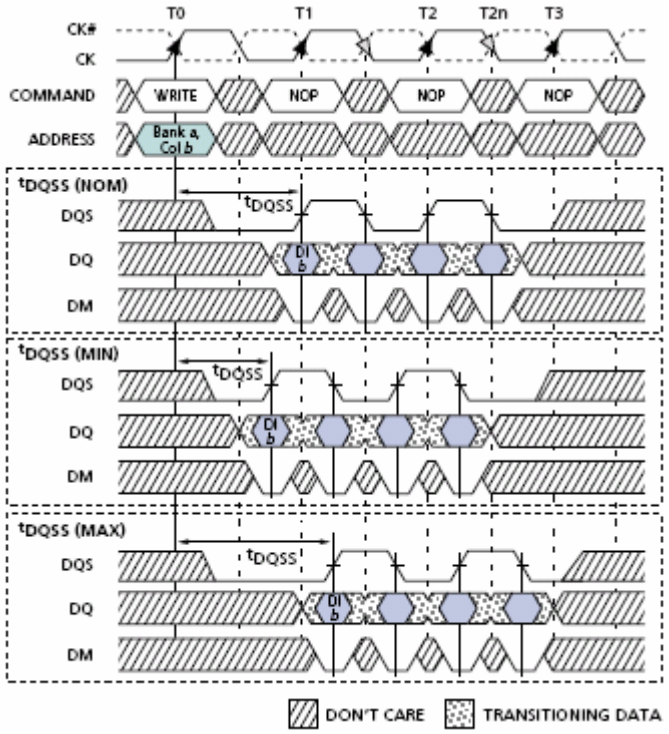


Figura 20: Ráfaga de escritura [11]

1.6.2 Inicialización

Para poder utilizar la memoria DDR hay que inicializarla previamente. En condiciones ideales esto tarda 200 µs. La secuencia de comandos, de modo resumido, que dicha inicialización realiza es la siguiente:

- DESELECT
- PRECHARGE
- LOAD MODE REGISTER para cargar el *extended mode registrer*
- LOAD MODE REGISTER para cargar el *mode registrer*
- PRECHARGE colocando todos los bancos del dispositivo en un estado IDLE
- Una vez en el estado IDLE, se realiza AUTO REFRESH cada vez que es necesario refrescar los datos.

Los pasos a realizar son veintiuno y se muestran de forma más detallada en la figura 21. Una vez realizadas todas las operaciones, la memoria está lista para ser utilizada.

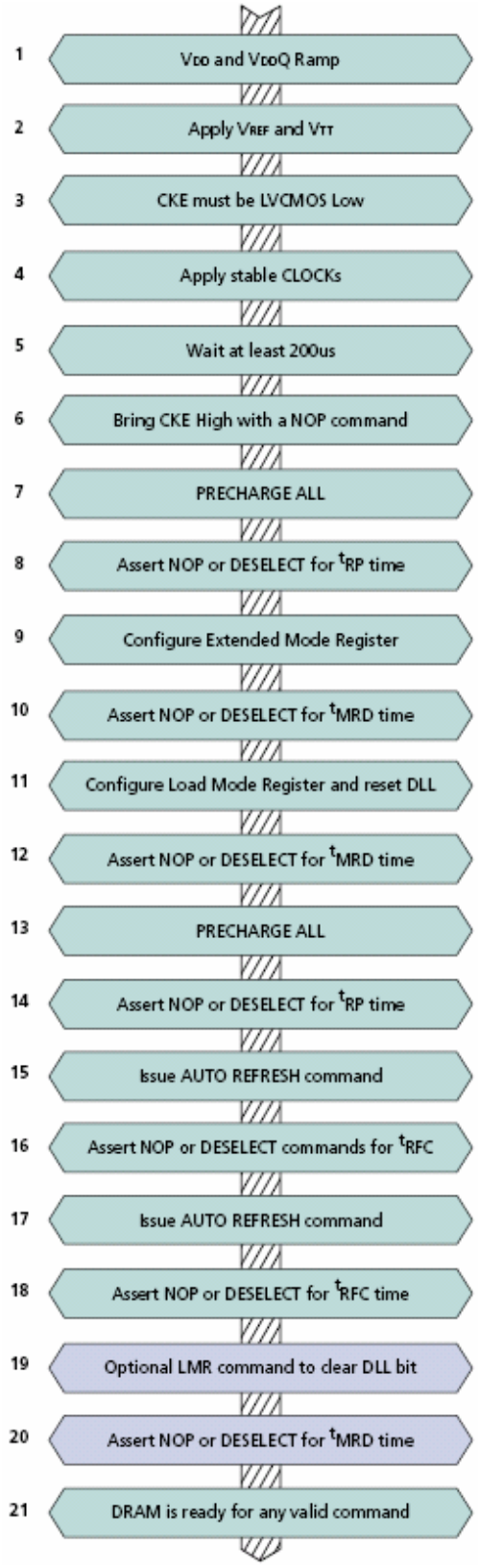


Figura 21: Diagrama inicialización de la DDR [11]

CAPÍTULO 2: HERRAMIENTAS SOFTWARE

Para el desarrollo de este proyecto se ha utilizado un paquete de herramientas desarrollado por Xilinx para diseñar, implementar, simular y verificar diseños hardware, así como realizar la configuración de la FPGA.

2.1 Xilinx ISE

El entorno de desarrollo ISE (Integrated Software Environment) es una herramienta de diseño de sistemas digitales. Las versiones utilizadas han sido la 8.1, 8.2 y 9.1i.

Con esta herramienta se puede crear, verificar, simular y sintetizar diseños basados en una FPGA o CPDL. Los pasos a seguir en el diseño e implementación se muestran en la siguiente figura:

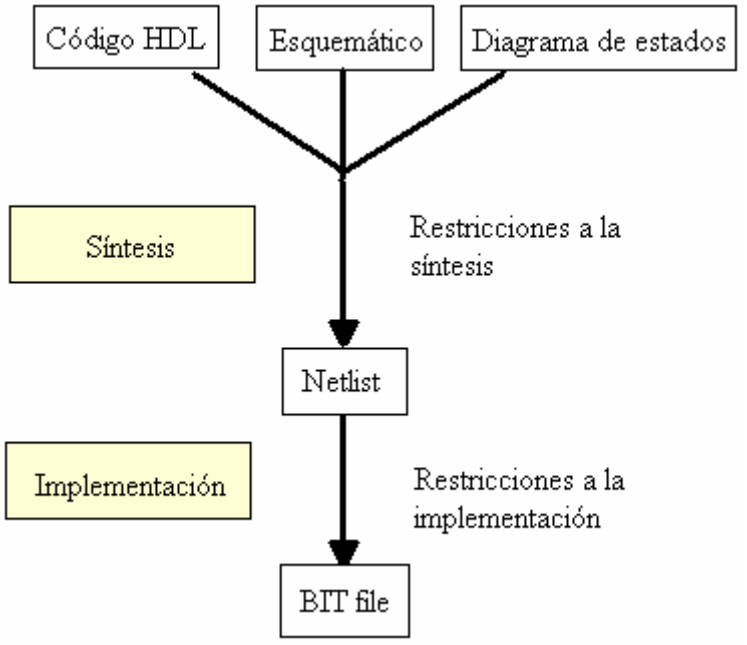


Figura 22: Flujo de diseño de circuitos digitales

Permite integrar otras herramientas para la simulación y verificación de los diseños, como son Modelsim y Chipscope.

El editor en el que se realiza el diseño es el Project Navigator. Los diseños se pueden generar en un lenguaje VHD, VHDL o Verilog, o mediante esquemáticos o dibujando diagramas de estado que transforma en código.

La interfaz del Project Navigator (figura 23) muestra de forma simultánea varia ventanas con toda la información necesaria para el desarrollo.

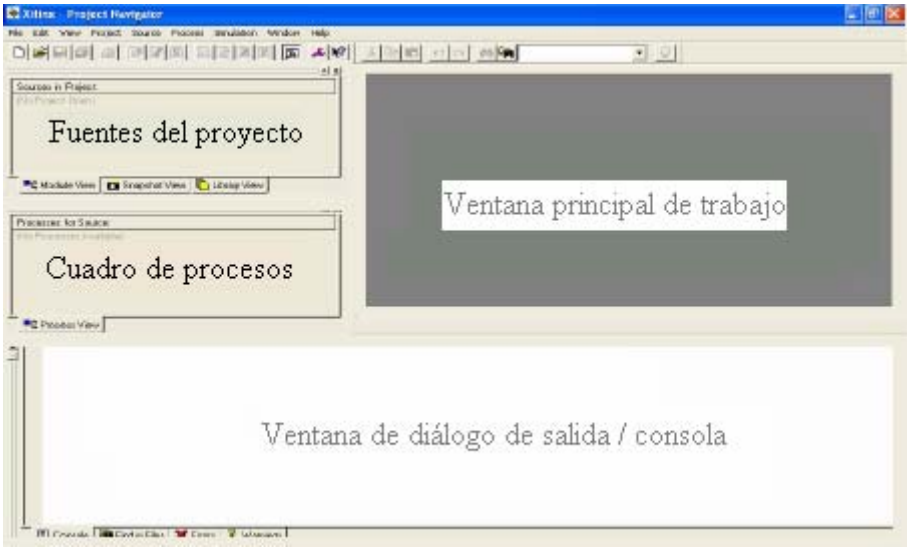


Figura 23: Interfaz Project Navigator

Las cuatro áreas en que se divide la ventana principal nos muestran los siguientes datos:

- Fuentes del proyecto: muestra, de forma jerárquica, todos los ficheros que forman parte del proyecto.
- Cuadro de procesos: muestra todos los procesos posibles para la fuente seleccionada en el cuadro anterior.
- Ventana principal de trabajo: en esta ventana se pueden editar los ficheros fuente, ver simulaciones…
- Ventana de diálogo de salida / consola: Esta ventana tiene varias funciones. Por un lado muestra todos los mensajes, errores y advertencias generados por los procesos. Por otro lado tiene un modo consola que permite realizar por línea de comandos cualquiera de las funciones que permite el interfaz.

2.1.1 Creación de un proyecto

Habrá que seleccionar la opción *New Project* del menú *File*. Aparece una ventana en la que se solicita el nombre del proyecto, la carpeta destino y el tipo de proyecto.

En el siguiente paso se deberá especificar el dispositivo con el que se va a trabajar y sus características.

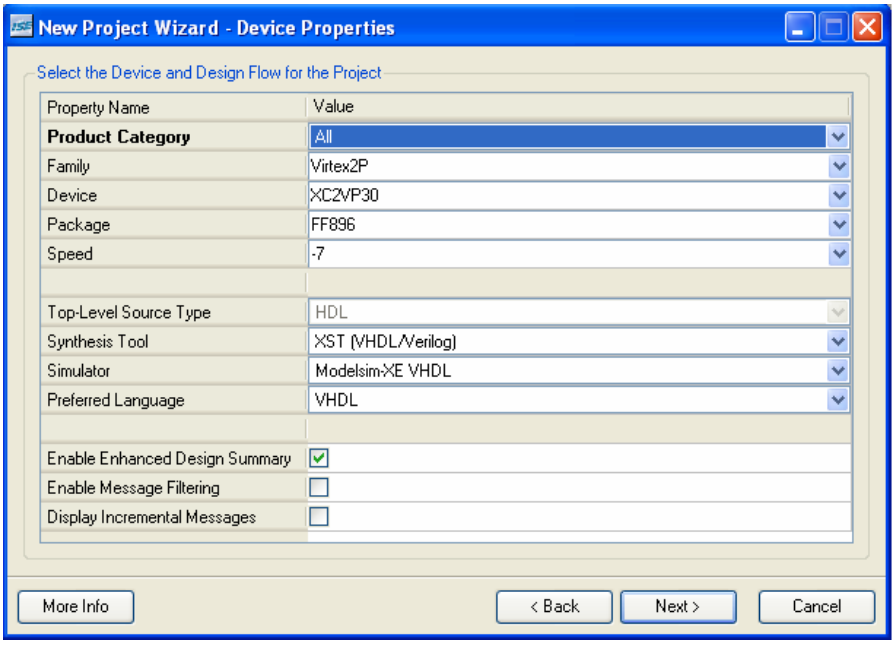


Figura 24: Ventana de creación de un proyecto

Los datos introducidos en la creación de este proyecto (figura 24), corresponden con el dispositivo sobre el que se vaya a volcar el diseño. Como ya se ha comentado en el capítulo anterior, el soporte utilizado ha sido XC2VP30 – FF896 de la familia Virtex II Pro. Cada FPGA puede funcionar a una velocidad, que depende de la tecnología utilizada. En este caso, la velocidad de funcionamiento es -7.

Durante la creación del proyecto se podrán crear nuevas fuentes o añadir ficheros ya existentes.

Antes de finalizar, nos muestra toda la información del proyecto creado por si se desea modificar algo.

2.1.2 Cuadro de Procesos y flujo de diseño

El cuadro de procesos nos muestra las operaciones que se pueden realizar sobre el fichero seleccionado en el cuadro de fuentes. Las opciones de síntesis e implementación solamente estarán disponibles si el fichero seleccionado es el más alto en la jerarquía.

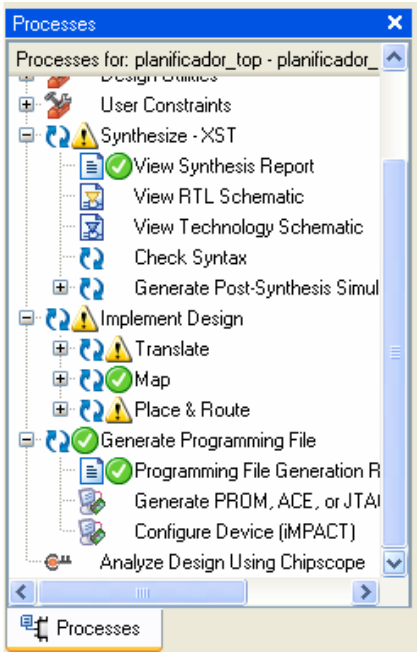


Figura 25: Cuadro de procesos

La función de síntesis transforma el código en un hardware concreto. Se definen restricciones temporales, asignación de pines y optimizaciones de cada uno de los componentes a partir de la descripción de comportamiento del sistema. Para realizar este proceso se deberá seleccionar la opción *Synthesize-XST*. Una vez finalizado ISE nos muestra un informe con toda la información de la síntesis.

El paso de implementación está dividido en tres fases: translate, map y place & route. Lo que se hace en este proceso es enlazar los módulos con las condiciones indicadas en el fichero de restricciones (.ucf) y reducirla a primitivas de Xilinx. A continuación se adapta al dispositivo en el cual se va a volcar el diseño y se colocan y se rutan los componentes físicos finales. Para realizar este proceso se seleccionará *Implement Design*. Cada una de las fases generará un informe.

En el fichero de restricciones se especificarán los pines a los que se deberá conectar cada señal de entrada y salida y su tecnología. Se pueden añadir restricciones temporales, como retardo máximo de una señal o frecuencia del reloj, y colocación de bloques en la FPGA.

El último paso será la generación del bitstream (fichero .bit) que será lo que se cargue en la FPGA. Este fichero contiene toda la información de localización de los dispositivos y los elementos de rutado. Para este proceso se seleccionará la opción *Generate Programming File*.

El flujo de diseño de un sistema implementado en un lenguaje de descripción hardware o mediante esquemático es el siguiente:

1) Implementación de cada uno de los módulos que forman el sistema.

2) Simulación funcional para comprobar que cada uno de los módulos se comporta como queremos.

3) Síntesis: debemos comprobar que todas las sentencias sean sintetizables ya VHDL es un lenguaje genérico para descripción, no sólo orientado a la síntesis.

4) Simulación post-síntesis: se comprueba que se ha sintetizado el circuito deseado y que funciona correctamente. Hay ocasiones en que la herramienta de síntesis ignora alguna sentencia no sintetizable dando lugar a un circuito que no es el deseado.

5) Colocación y rutado: El proceso de colocación consiste en situar los bloques digitales obtenidos en la síntesis de forma óptima, de forma que aquellos bloques que se encuentran muy interconectados entre si se sitúen próximamente. El proceso de rutado consiste en conectar adecuadamente los bloques entre si, intentando minimizar retardos de propagación para maximizar la frecuencia máxima de funcionamiento del dispositivo.

6) Extracción post-síntesis: una vez completado el paso anterior se extraen los retardos de los bloques y sus conexiones para poder realizar una simulación temporal.

7) Simulación temporal: esta simulación es necesaria porque es posible que el diseño no funcione a causa de los retardos internos del chip. Con esta simulación se pretende comprobar esto. Si se produce un error habrá que volver a los pasos anteriores.

8) Programación del dispositivo: se vuelca el mapa de bits en el dispositivo final y se comprueba el resultado.

Todos los pasos aquí descritos se realizarán a través de las opciones disponibles en el cuadro de procesos mostrados en la figura 25.

2.2 Modelsim

Esta herramienta se ha utilizado para realizar las simulaciones en comportamiento, post-translate, post-map y post-route del diseño para comprobar el correcto funcionamiento del mismo en las diferentes etapas del desarrollo.

Se puede lanzar a ejecución desde el ISE por lo que una vez abierto el programa aparece el diseño compilado y muestra directamente las formas de onda.

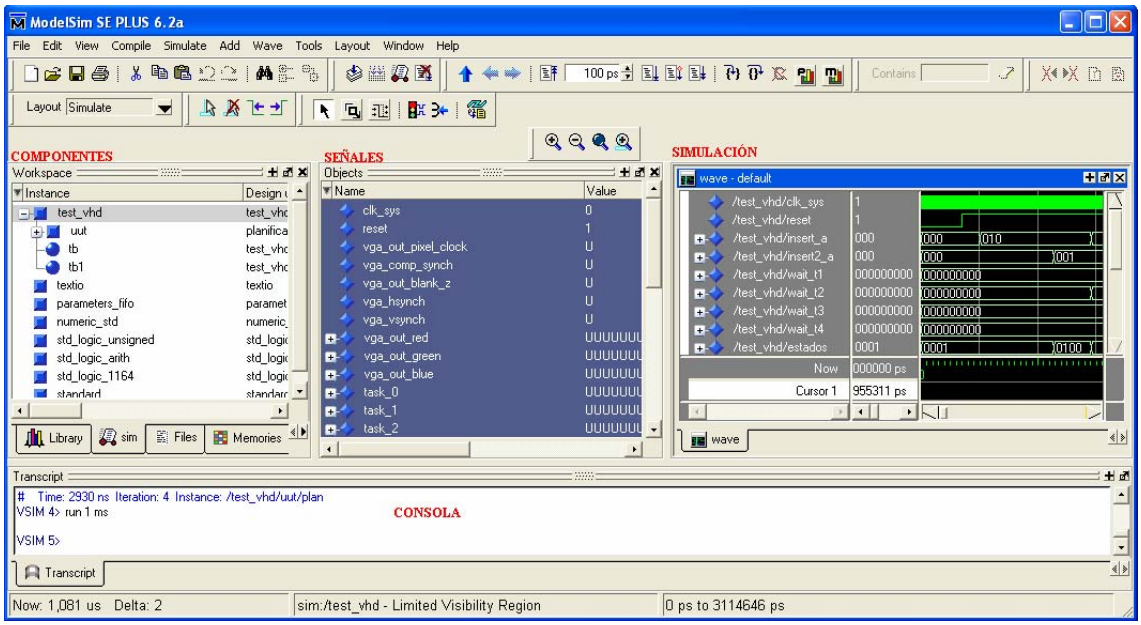


Figura 26: Ventana principal de Modelsim

La ventana principal, que se muestra en la figura 26, aparece dividida en cuatro áreas:

- Componentes: en este cuadro se muestran todos los componentes que forman parte del diseño así como las librerías utilizadas.
- Señales: se muestran todas las señales (tanto las definidas por usuario como las inferidas en el proceso de síntesis) que forman parte de la instancia seleccionada en la ventana de componentes. Aquí se pueden seleccionar las señales que se deseen añadir a la simulación.
- Simulación: se muestra el valor de las señales añadidas durante el tiempo que se ha corrido la simulación.
- Consola: mediante línea de comando nos permite ejecutar cualquier operación que Modelsim sea capaz de realizar, como añadir y eliminar señales de la simulación, ejecutar, compilar …

2.3 Chipscope Pro Tools

Es un conjunto de herramientas de depuración, proporcionadas por Xililnx, que nos permite verificar el valor que tienen las señales dentro de la FPGA. Se podría describir como la versión software de un analizador lógico.

Las dos aplicaciones utilizadas han sido Chipscope Pro Core Inserter y Chipscope Pro Analyzer.

El Chipscope Pro Core Inserter es una herramienta de post-síntesis usada para añadir al diseño un circuito, llamado core, que permitirá sacar al exterior las señales que queramos analizar [12].

El Chipscope Pro Analyzer es un interfaz a través del cual podemos ver las señales que forman parte del core generado. Permite elegir los disparadores y la forma en que se visualizan los datos [12].

Ambas aplicaciones pueden ser ejecutadas desde Xilin ISE.

2.3.1 Chipscope Pro Core Inserter

Esta herramienta se ejecutará después de realizar la síntesis del diseño ya que utiliza los ficheros generados en este proceso. En ella especificaremos las señales que se desean muestrear.

El primer paso será crear un fichero nuevo (*File* → *New*). En la ventana que aparece (figura 33) deberemos introducir el fichero .ngc (conocido como lista de cables y que se ha generado en la síntesis), el nombre del fichero de salida (extensión .ngo) y el directorio en que se creará. También nos pide la familia del dispositivo que se va a utilizar.

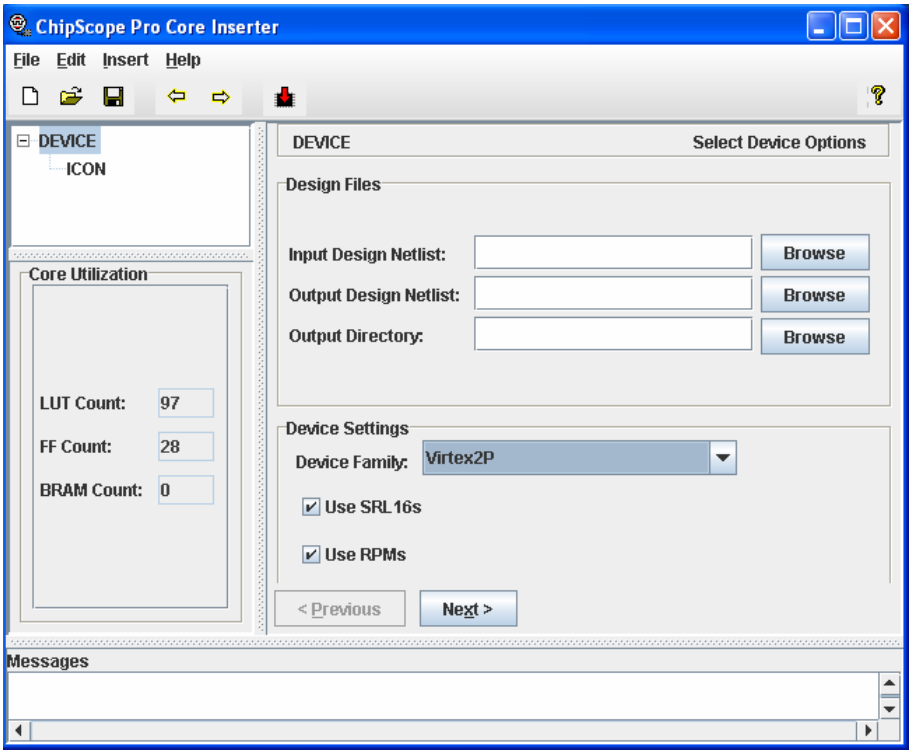


Figura 27: Chipscope Pro Core Inserter (Paso 1)

En el siguiente paso nos pedirá que definamos los disparadores y las señales a muestrear (figuras 28 y 29).

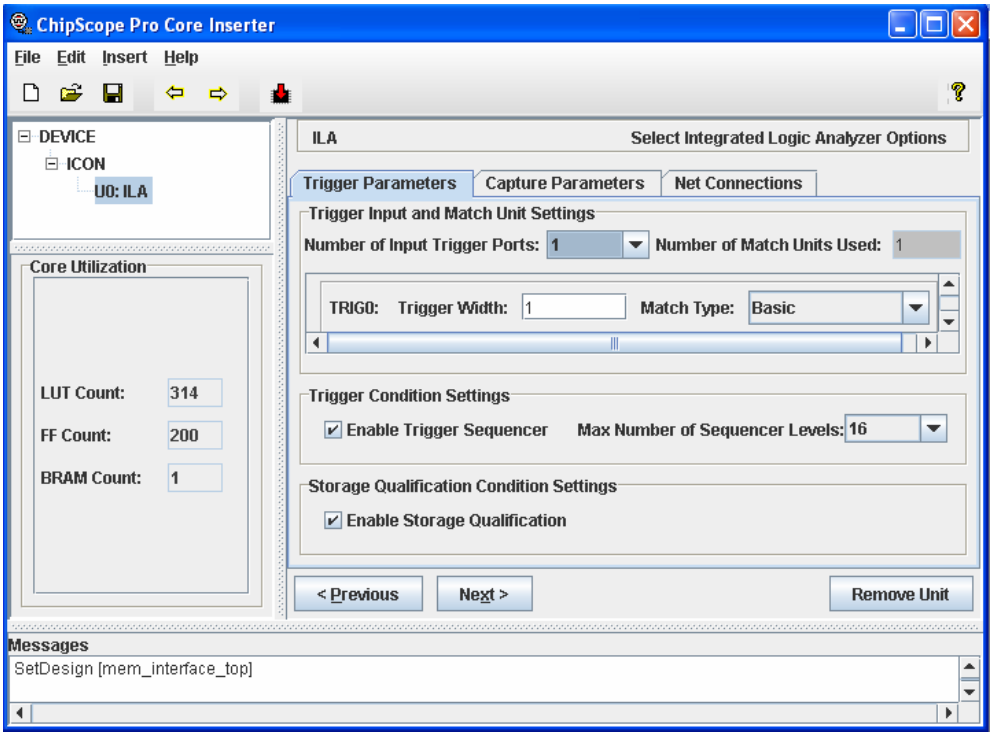


Figura 28: Chipscope Pro Core Inserter (Paso 2)

En la pestaña *Trigger Parameters* se introduce el número de disparadores y el número de bits que lo forman (ver figura 28).

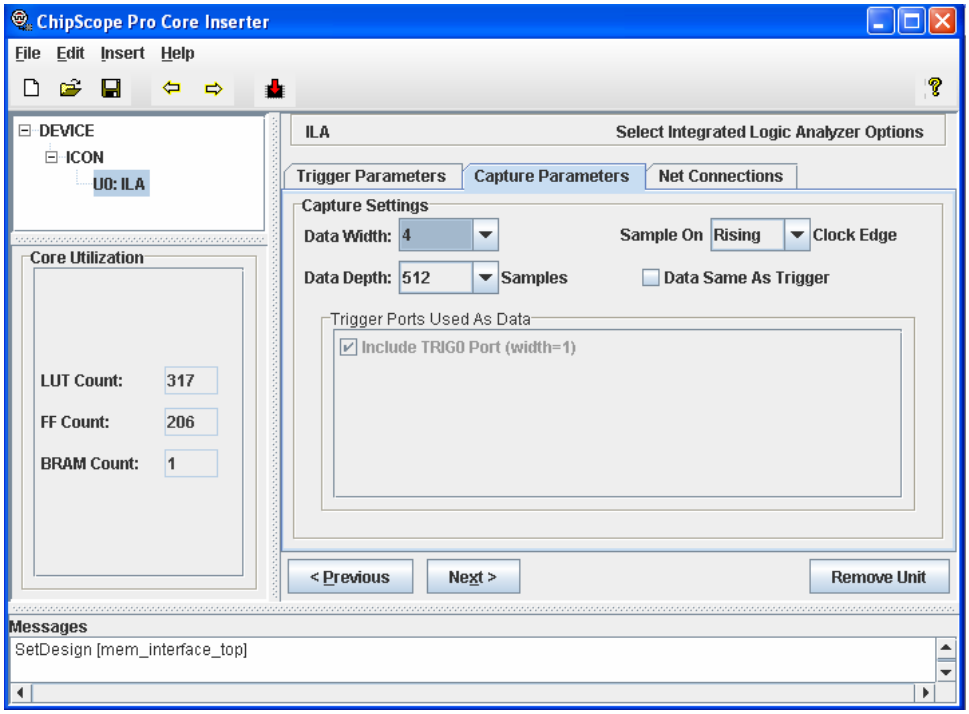


Figura 29: Chipscope Pro Core Inserter (Paso 3)

En *Capture Paramenters* (figura 29) se pide el número de ciclos que se mostrarán desde el primer disparo del trigger (*data depth*). Si las señales a muestrear no son las mismas que se ha especificado como disparadores habrá que indicar el número de bits a mostrar (*data width*). A estos bits serán a los que asignaremos las señales que queremos analizar.

En la pestaña *Net Connections* habrá que asignar al reloj, los disparadores y las salidas señales de nuestro diseño. Para realizar las asignaciones habrá que pinchar sobre el botón *modify connectios*. Estos pasos se muestran en la figura 30.

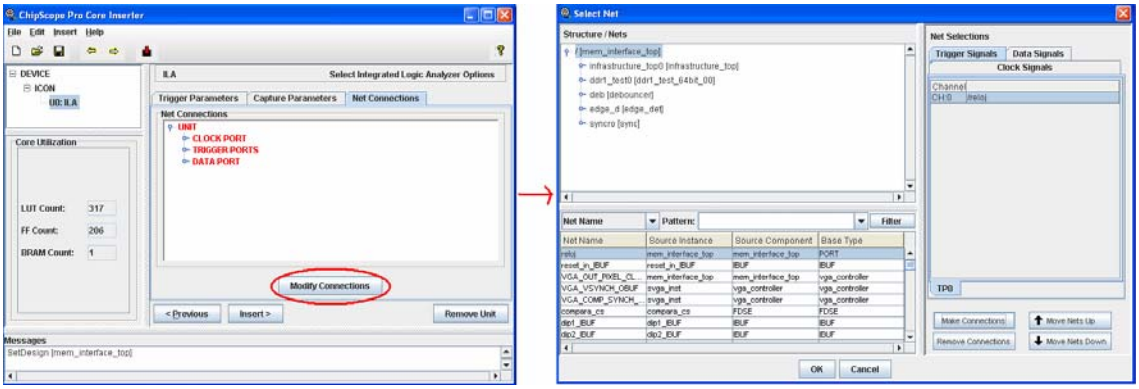


Figura 30: Chipscope Pro Core Inserter (Paso 4)

Tras asignar todos los cables, mediante el botón *make connections*, se confirma y se vuelve a la ventana principal. Una vez ahí se procederá a insertar el core (botón *insert*).

Una vez realizados estos pasos se procederá a la implementación del diseño tomando como entrada el fichero .ngo generado por esta aplicación.

2.3.2 Chipscope Pro Analyzer

Esta herramienta es un interfaz que nos permite ver las señales incluidas en el core generado por Chipscope Pro Core Inserter.

Para poder utilizar esta herramienta es necesario que la FPGA esté conectada al ordenador (del modo mostrado en la figura 31) mediante uno de los siguientes tipos de cables:

- Cable USB
- Cable paralelo tipo III o IV
- Multipro (modo JTAG)

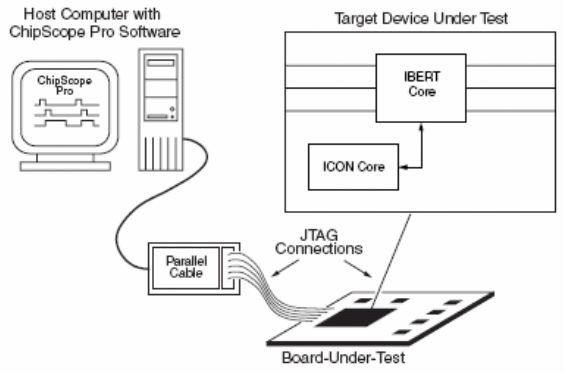


Figura 31: Modo de uso del Chipscope Pro Analyzer [12]

Para poder muestrear la señales tendremos que tener nuestro diseño cargado en la FPGA.

Lo primero que debemos hacer es seleccionar el tipo de conexión que vamos a utilizar. Para ello nos metemos en el menú *JTAG Chain* y seleccionamos el tipo de cable, tal como se muestra en la figura 32. Lo siguiente será hacer un escaneo para detectar todas las conexiones posibles. Lo haremos pinchando sobre el botón que aparece indicado en la siguiente imagen.

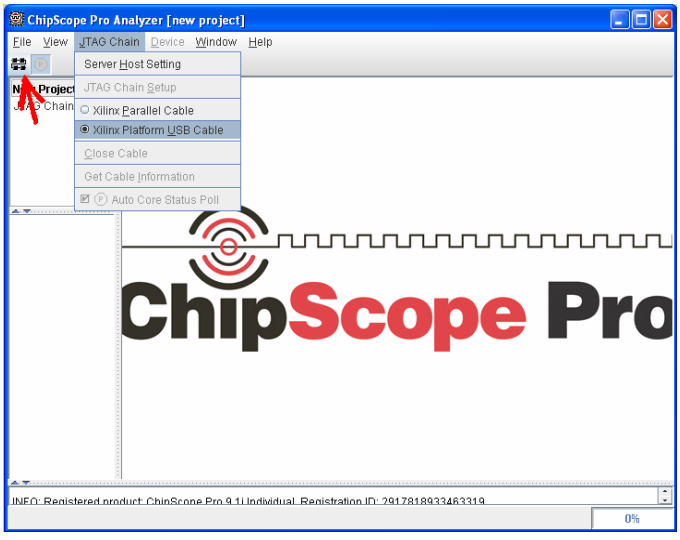


Figura 32: Interfaz de Chipscope Analyzer

Tras el escaneo se nos abre una ventana con todos los dispositivos detectados. Deberemos seleccionar aquel a través del cual queremos establecer la conexión (figura 33).

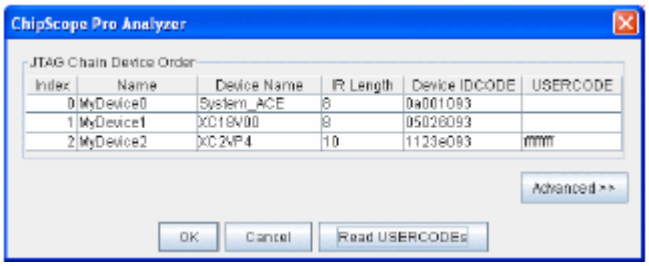


Figura 33: Selección de conexión con la FPGA

Lo siguiente es especificar la condición de disparo de los disparadores que habíamos definido en Chipscope Pro Core Inserter.

En el momento en que se cumpla la condición de alguno de los disparadores seleccionados aparecerán en pantalla todas las formas de onda especificadas. Los bits se podrán unir formando buses, renombrar o cambiar el color para simplificar la depuración. La edición de las señales y su visualización se muestran en la siguiente imagen.

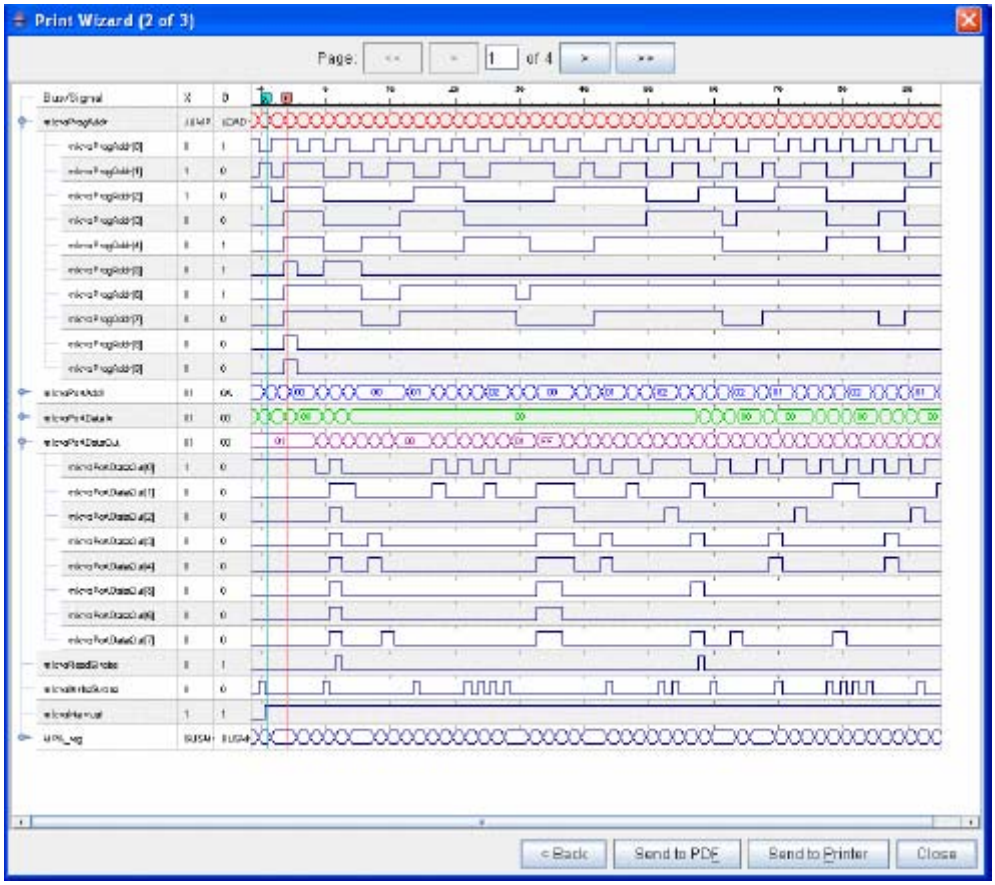


Figura 34: Visualización de señales en Chipscope Analyzer

2.4 MIG 007

El MIG 007 (Memory Interface Generator) es una aplicación proporcionada por Xilinx que nos permite implementar controladores de memoria. Los tipos de memoria soportados son DDR SDRAM (Spartan 3, Virtex II y Virtex II Pro) y DDR2 SDRAM diferencial y no diferencial (Virtex II y Virtex II Pro) [13].

Esta herramienta nos genera un controlador, en VHDL o Verilog, para la memoria seleccionada con la opción de incorporar un banco de pruebas que nos permite comprobar el correcto funcionamiento del mismo. Se crea una arquitectura jerárquica que encapsula todo el mecanismo de sincronización y gestión de señales de control.

La aplicación toma como entrada los datos proporcionados por el usuario a través de una interfaz gráfica y genera los archivos RTL (VHDL o Verilog), SDC, UCF y ficheros de documentos.

La interfaz de usuario permite ajustar los siguientes parámetros:

- Tipo de memoria para el que se va a generar el controlador (DDR1, DDR2 diferencial o DDR2 no diferencial).
- Seleccionar entre una memoria components o DIMMS.
- x4 / x8 para indicar el número de bits de datos por strobe.
- Familia de la FPGA que puede ser Spartan 3, Virtex II o Virtex II Pro.
- Número de dispositivo.
- Velocidad a la que funciona el dispositivo. Para la Spartan 3 están diponibles las velocidades -4 y -5, para Virtex II son -4, -5 y -6, y para Virtex II Pro -5, -6 y -7.
- Frecuencia a la que funcionará el controlador generado. Las velocidades máxima y mínima permitidas están fijadas por la velocidad del dispositivo fijada en el punto anterior.
- 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). Para un dispositivo y unos bancos fijados, nos puede limitar el ancho de los datos.
- Ancho de datos.
- Ancho de direcciones de fila. Puede ser 12, 13 o 14.
- Ancho de direcciones de columna. Puede tener uno de los siguientes valores: 9, 10, 11 y 12.
- Ancho de direcciones de banco de registros. Puede ser 2 o 3.

- Reserva y uso de los pines V_{RP} y V_{RN}. Si su casilla está marcada estos pines se incluirán en el fichero de restricciones.
- Ruta dentro de la jerarquía. Nos permite cambiar la jerarquía del controlador a generar para facilitar su integración en el entorno a utilizar.
- Directorio destino
- UCF de pines reservados
- Opciones de implementación como usar DCM o añadir un banco de pruebas

A partir de estos datos y la asignación de pines realizada en el *Pin editor* generará todos los ficheros comentados.

Pasos a seguir para la generación el controlador:

En primer lugar se introducirán todos los requisitos que debe cumplir el controlador (figura 35).

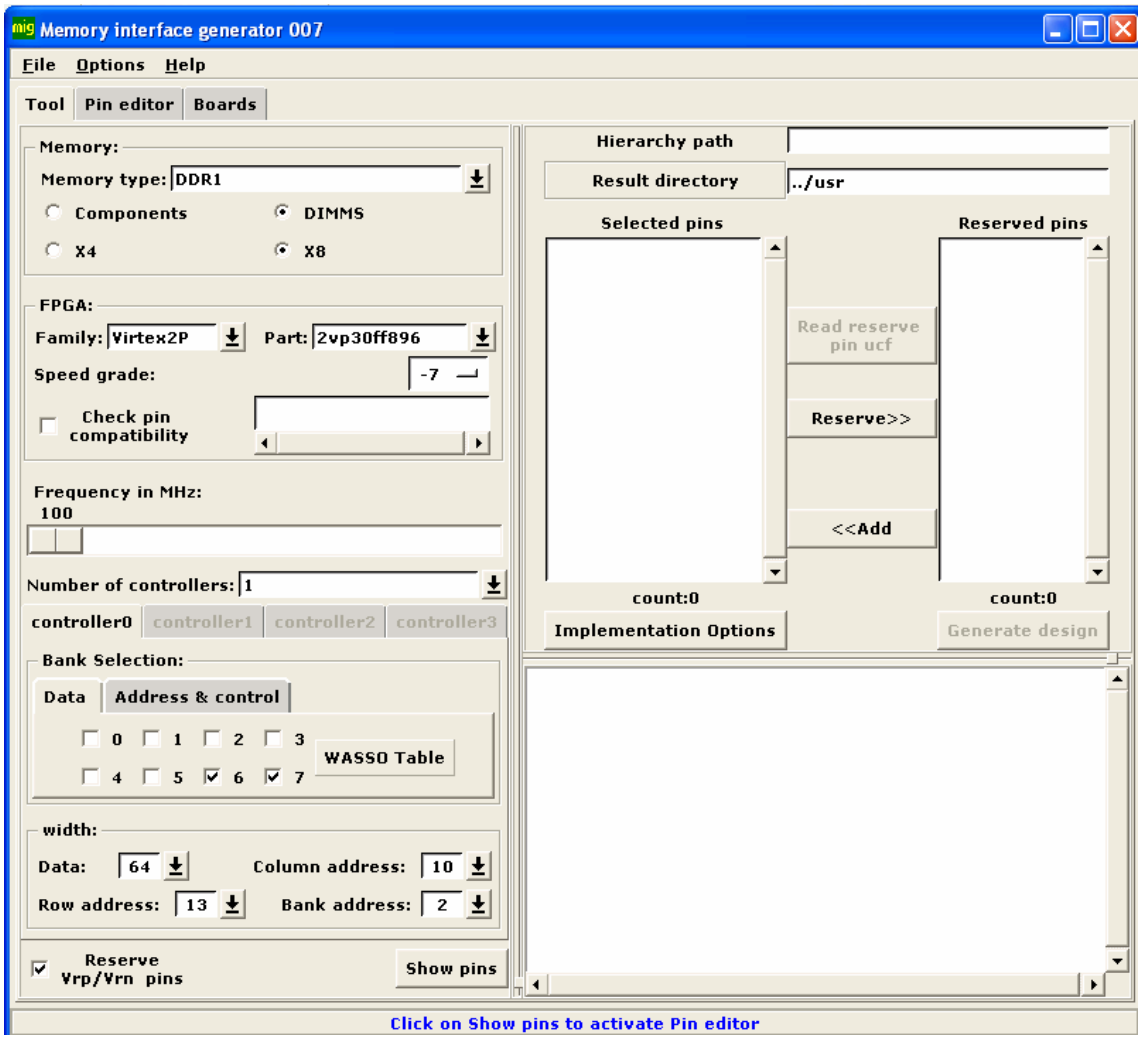


Figura 35: Generación del controlador con MIG: paso 1

El siguiente paso será indicar si se desea añadir un banco de pruebas al controlador, para probar su funcionamiento, y marcar si se utiliza un DCM para la gestión del reloj. Para ello habrá que pinchar sobre el botón *Implementation Option* tal y como se muestra en la figura 36.

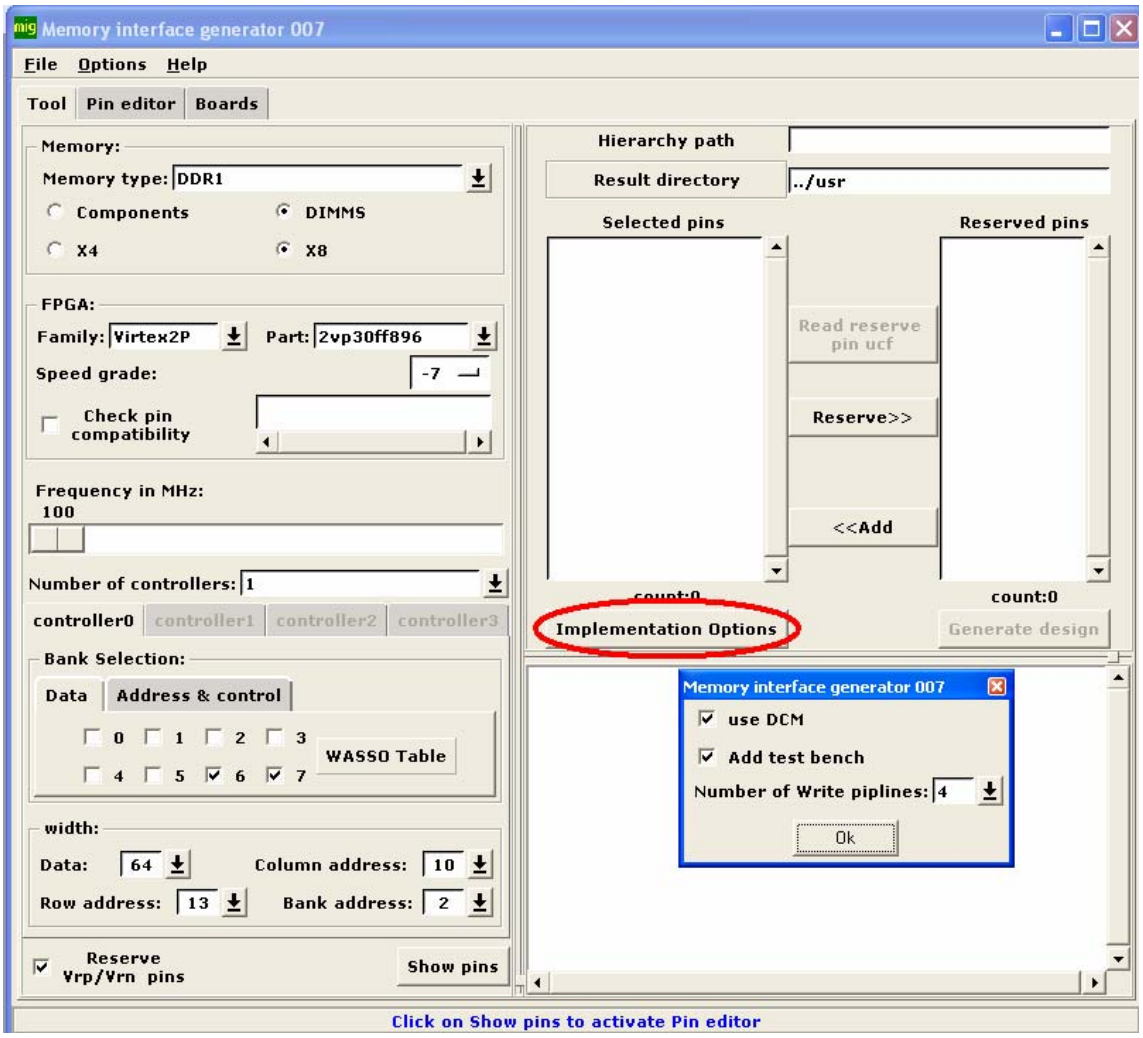


Figura 36: Generación del controlador con MIG: paso 2

A continuación, se deberá indicar, en el menú *Option*, la herramienta de síntesis, le lenguaje VHD en que se implementará y el tipo de controlador (figura 37).

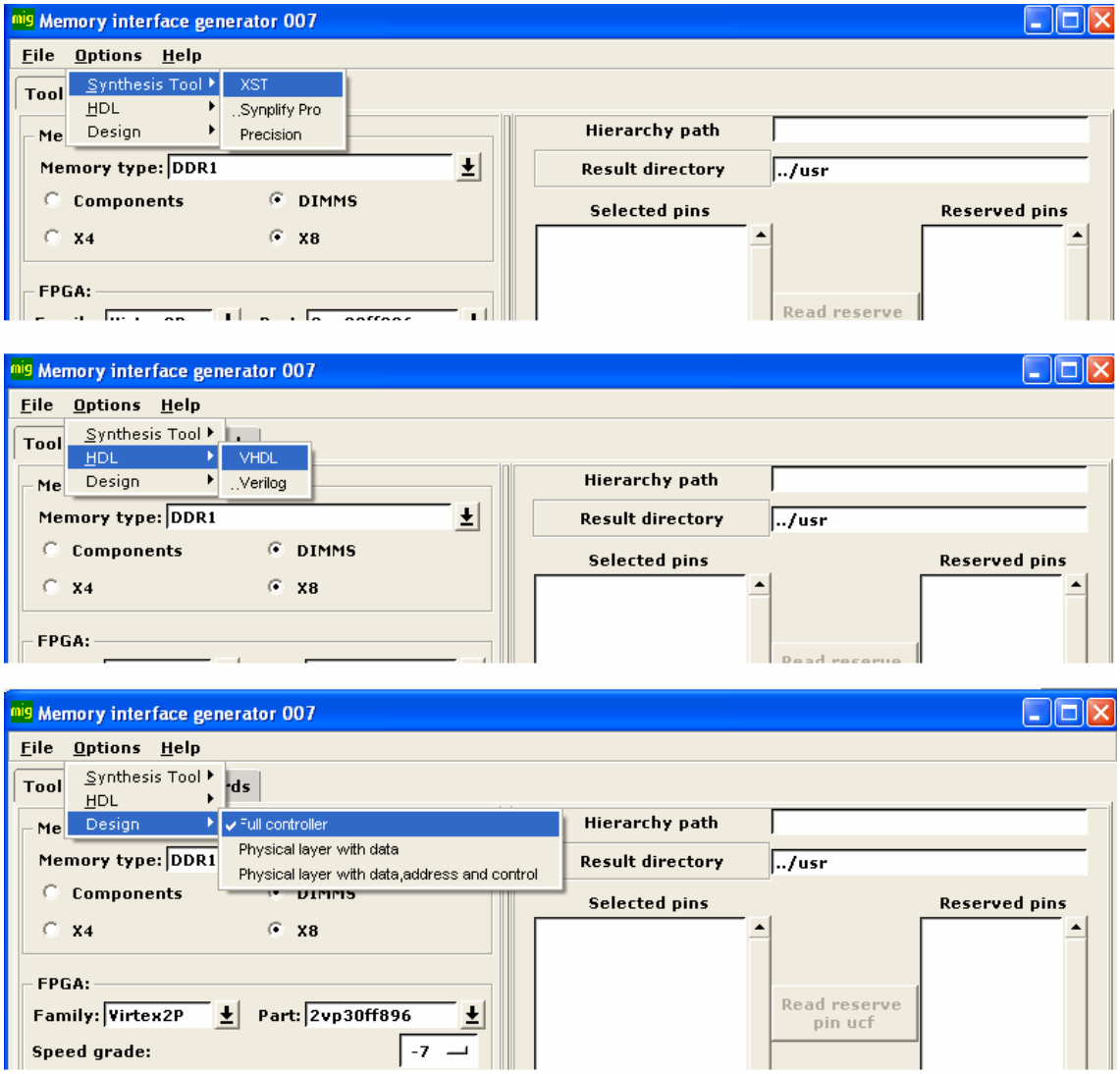


Figura 37: Generación del controlador con MIG: paso 3

Tras pinchar sobre el botón *Show pins* (se indica en la figura 38) se abre la pestaña Pin editor para asignar los pines de entada y salida a las señales del controlador. Con los datos introducidos se generará el fichero de restricciones.

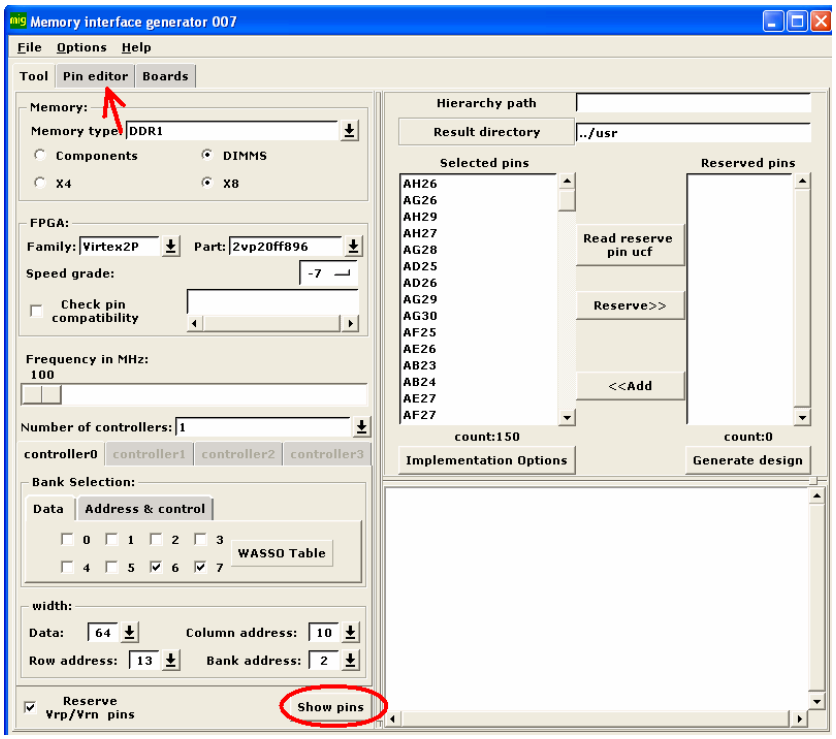


Figura 38: Generación del controlador con MIG: paso 4

En esta pantalla hay varias opciones, las asignaciones se pueden introducir a mano o cargarlas desde un fichero. Las diferentes opciones se muestran en la figura 39.

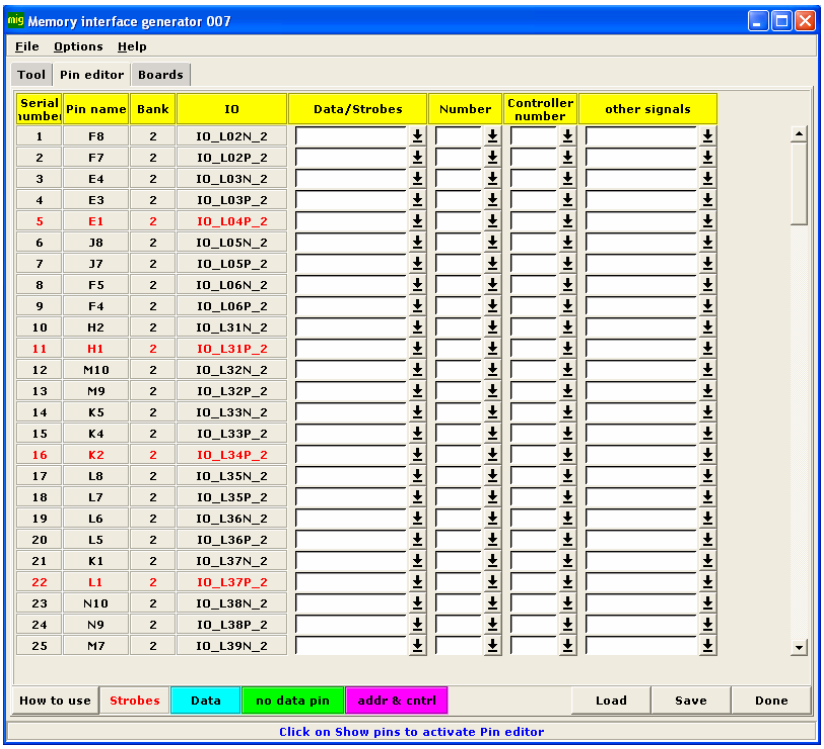


Figura 39: Generación del controlador con MIG: paso 5

2.5 iMPACT

Esta herramienta permite a los diseñadores configurar y programar de una forma sencilla las placas de prototipado, a través de una interfaz gráfica.

Al abrirlo hace un rastreo y una vez detectados los dispositivos establece la comunicación.

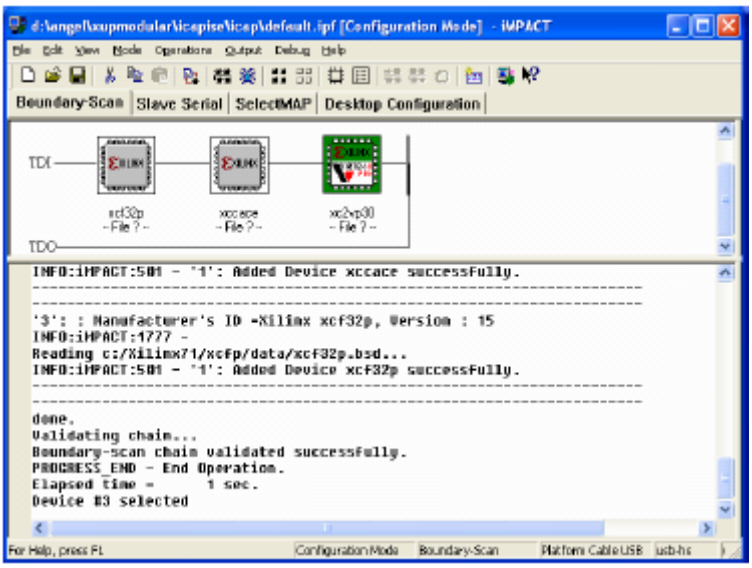


Figura 40: Ventana principal de iMPACT

Para cada unos de los dispositivos detectados nos dará la opción de asociarles un mapa de bits o en caso de que no se quiera utilizar ese dispositivo se seleccionará la opción *by pass*. Si como en la figura 40 solo se desea configurar el último de los dispositivos detectados habrá que seleccionar *by pass* dos veces y en el tercero seleccionar el fichero .bit que se le asignará.

Cada vez que se modifique el fichero asociado al dispositivo habrá que reprogramar seleccionando la opción *program*.

Esta herramienta se ha utilizado en este proyecto para cargar el bitstream generado en la FPGA.

CAPÍTULO 3: CONTROLADOR MEMORIA DDR SDRAM

Para poder utilizar una memoria DDR SDRAM es necesaria la implementación de un controlador que permita gestionar el dispositivo.

Dada la complejidad que presenta el diseño de este tipo de controlador se ha partido del código generado por la aplicación Memory Interface Generator (MIG) de Xilinx, que permite obtener controladores para este tipo de memorias de una manera, a priori, sencilla. Dicha aplicación ha sido explicada en el apartado de herramientas software.

3.1 Generación del controlador con la herramienta Memory Interface Generador

Los parámetros introducidos en la herramienta han sido los siguientes:

- Memory: memory type DDR1 y marcadas las opciones DIMMS y x8.
- FPGA: Family Virtex2P, Part 2vp30ff896
- Frequency: 100 MHz.
- Number of controllers: 1.
- Bank selection: Data → se marcan las posiciones 6 y 7.

  Address & Control → se marcan las posiciones 6 y 7.
- Width: Data → 64

  Column Address → 10

  Row Address → 13

  Bank Address → 2
- Implementation options: se marca DCM y Add test bench. Number of pipelines → 4
- En el menú Options: Synthesis Tool → XST

  HDL → VHDL

  Design → Full controller
- En la pestaña Pin Editor se ha cargado el fichero .pin que nos han proporcionado. Este fichero contiene todas las asignaciones de pines necesarias.

Con estos datos, nos genera un controlador para una memoria DDR1 con banco de pruebas que se ejecutará sobre una FPGA Virtex2P. La velocidad de funcionamiento de 100 MHz.

El diseño generado por el MIG tiene la siguiente jerarquía de componentes:

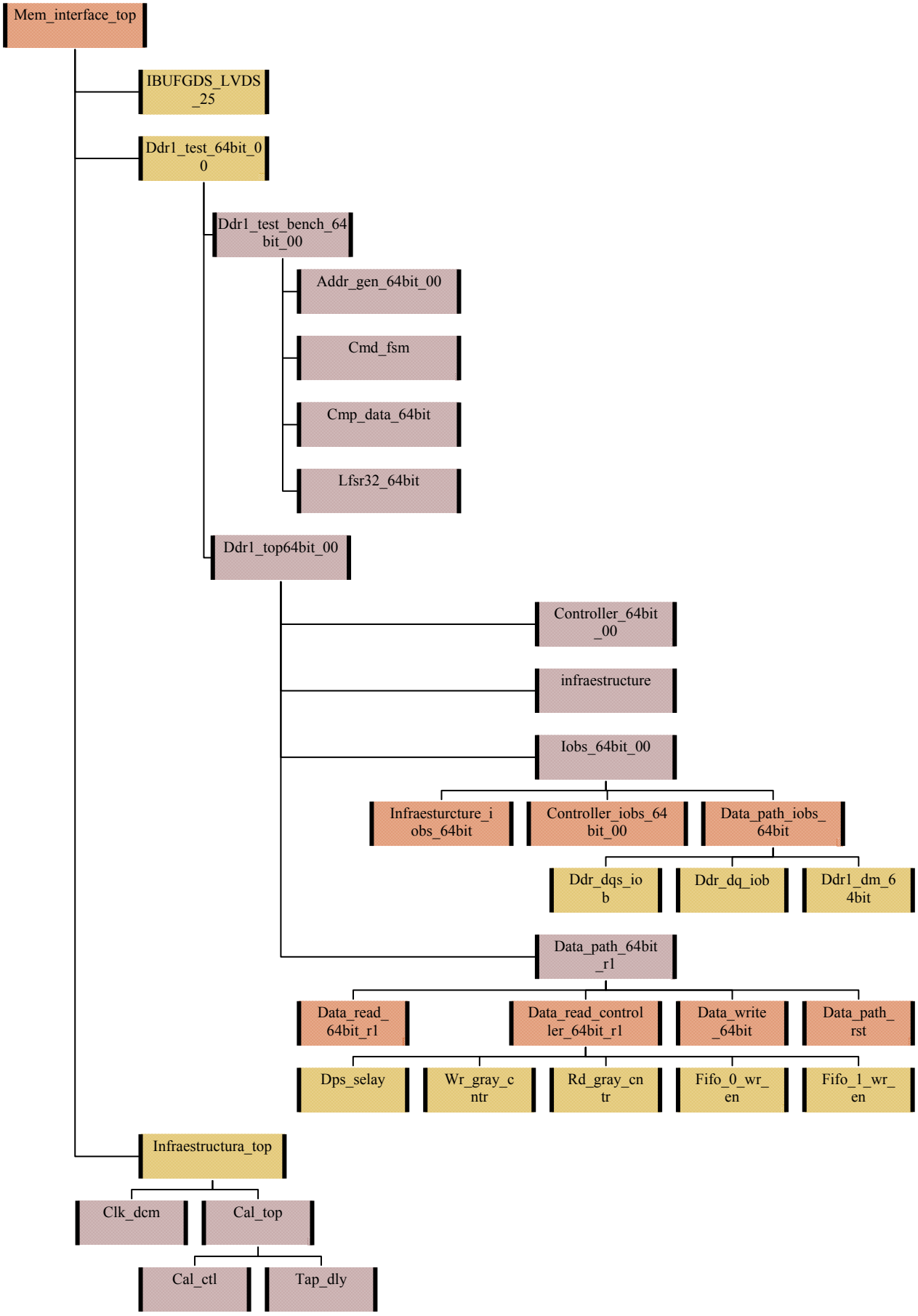


Figura 41: Jerarquía del controlador DDR SDRAM con banco de pruebas

Veamos una breve descripción del funcionamiento de los componentes más relevantes:

- Mem_interface_top: es el componente más alto en la jerarquía del controlador y la interfaz con el usuario. Cualquier señal que se quiera mostrar al exterior o meter al controlador deberá añadirse aquí. Tiene los siguientes módulos:
    - *Infraestructura_top*: recibe como entradas la señal de reset y el reloj de la FPGA y genera, mediante el uso de un DCM, las señales de reloj y reset que utilizará el controlador. También genera la señal *wait_200us* que se activa una vez transcurridos 200 µs desde el arranque del sistema.
    - *Ddr1_test_64bit_00*: dentro de este módulo se une el banco de pruebas con el controlador de la memoria.
- Ddr1_test_bench_64bit_00: aquí se encuentra encapsulado todo lo referente al banco de pruebas. Se compone de los siguientes módulos:
    - *Addr_gen_64bit_00*: es el encargado de generar las direcciones de memoria sobre las que se va a escribir o leer.
    - *Cmd_fsm*: es una máquina de estados que genera el código de la operación que debe realizar el controlador. Se podría decir que es el controlador de "grano grueso" ya que describe las funciones principales como lectura, escritura, refresco, inicialización… pero no supervisa las señales de control de la memoria; el encargado de esta tarea es controller_64bit_00.
    - *Cmp_data_64bit*: es un comparador de datos de tamaño 64 bit. Recibe una señal indicando que hay un dato nuevo disponible, el dato que ha leído y el dato que debería haber leído y genera una señal indicando si se ha producido un error de lectura.
    - *Lfsr32_64bit*: es el encargado de generar los datos. Hay dos módulos de este componente; uno se encarga de generar los datos de escritura y el otro los datos para el cmp_data_64bit.
- Ddr1_top_64bit_00: dentro de este componente se encuentra el auténtico controlador. Recibe todos los datos necesarios para el control de la memoria, tales como el código de operación a realizar, el registro de configuración, dirección de entrada, banco de registros, datos de entrada y máscara de los datos. También genera señales de control para otros módulos y maneja la ruta de datos de escritura y lectura. Comportamiento de algunos de sus componentes:
    - *Controller_64bit_00*: es el encargado de generar las señales de control de la memoria, es decir, en función del código de operación que recibe (generado en cmd_fsm) da los valores oportunos a cs, cas, we y ras teniendo en cuenta las características y requisitos de la memoria. También genera la petición de refresco. Su máquina de estados es la siguiente:

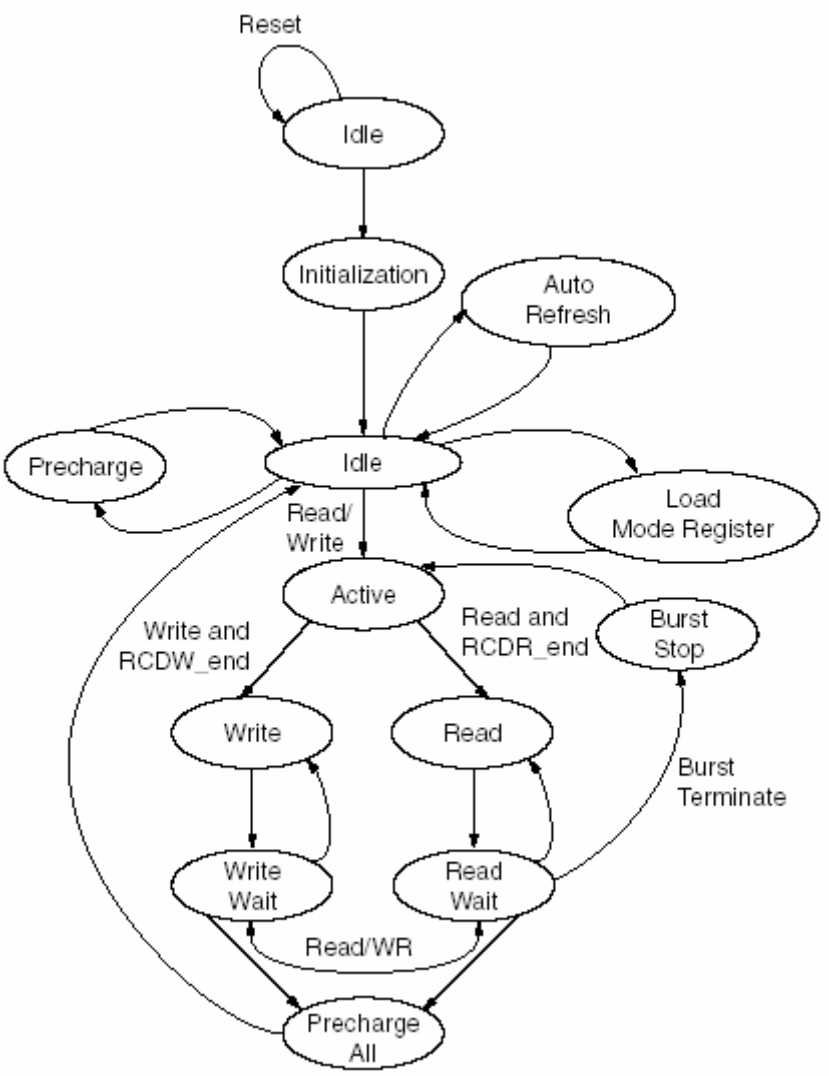


Figura 42: Máquina de estados de controller_64bit_00 [14]

- *Data_path_64bit_rl*: este módulo comprende la ruta de datos, tanto de entrada como de salida, del interfaz de la memoria. Su función consiste en almacenar lo datos leídos, transferir los datos de escritura al bus y capacitar los buffers. Sus principales componentes so:
  - Data_read_controller_64bit_rl: genera todas las señales usadas en la ruta de datos de la lectura.
  - Data_read_64bit_rl: contiene la ruta de datos de lectura.
  - Data_write_64bit: contiene la ruta de datos de escritura.

3.2 Comprobación del funcionamiento del banco de pruebas y primera corrección

El funcionamiento del banco de pruebas consta fundamentalmente de tres etapas:

- La primera comprende toda la inicialización de la memoria, durante la cual se fija el tamaño de ráfaga a 4.
- La segunda consiste en un barrido completo de la memoria inicializando todas las posiciones con el valor generado por el componente *lfsr32_64bit*.
- La tercera realiza sucesivos barridos de la memoria para leerla.

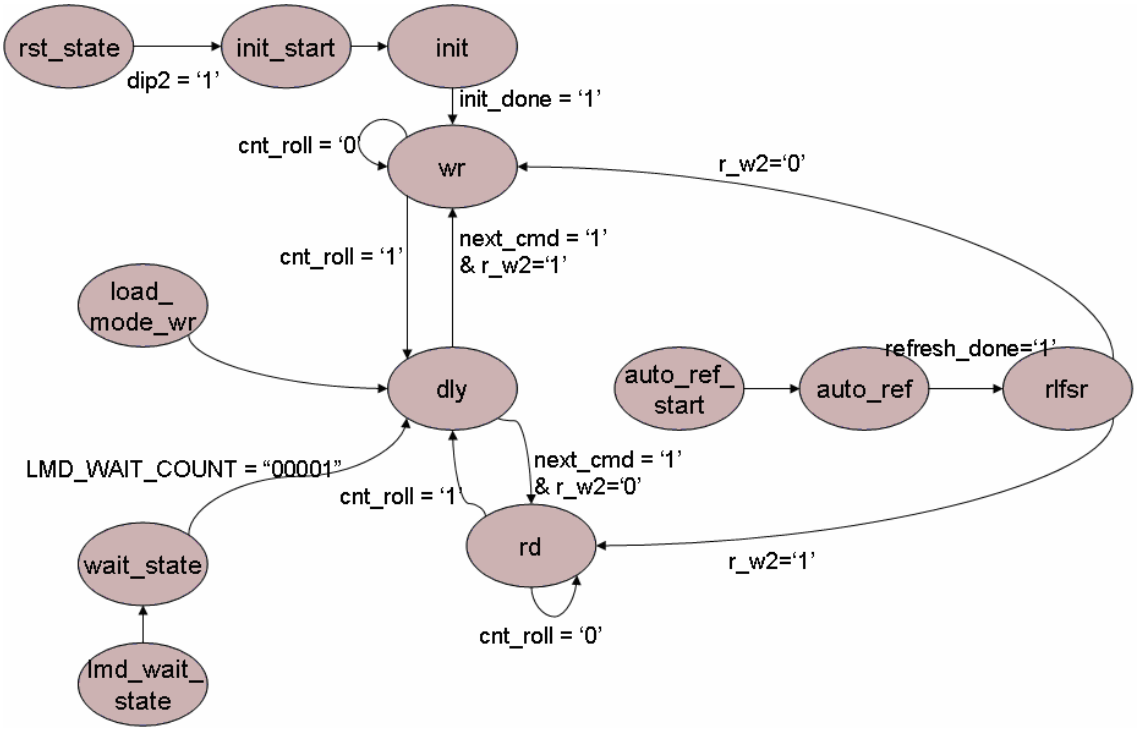


Figura 43: Máquina de estados del módulo cmd_fsm (banco de pruebas)

Como se puede ver en el diagrama de estados del componente *cmd_fsm* (figura 43) existen estados que no son alcanzables con lo cual las únicas operaciones que realiza son las descritas arriba. Este diagrama de estados fue obtenido mediante ingeniería inversa, estudiando el código generado por el MIG. La existencia de estados inalcanzables nos hizo sospechar que la herramienta no estaba suficientemente depurada y que aún se encuentra en una fase temprana de desarrollo.

Antes de continuar con la descripción del funcionamiento, es importante indicar que se incorporó un controlador vga en el módulo *mem_interface_top* para poder ver el resultado de las lecturas realizadas a través de un monitor.

Los códigos de operación generados por el autómata que se muestra en la tabla 5 en función del estado siguiente son:

Operación	Código
Inicialización	010
Escritura	100
Lectura	110
Auto refresh	010
Self refresh	011
Cargar configuración	101

Tabla 5: Códigos de operación

En una primera ejecución se comprobó que el resultado de las lecturas no era el esperado. Sin embargo el dato obtenido no se desviaba demasiado del correcto, sólo bailaban algunas cifras aunque el comportamiento empeoraba según transcurría el tiempo de ejecución. Esto y un análisis del autómata nos llevaron a pensar en un posible fallo en el refresco. Para asegurarnos sacamos por pantalla únicamente el resultado de las cuatro primeras lecturas (por defecto el banco de pruebas realiza sucesivos un barrido de la memoria completa). Los datos mostrados en esta prueba eran correctos aunque si se dejaba ejecutando un tiempo comenzaban a bailar algunos bits.

Tras analizar el autómata se comprobó que no tenía en cuenta la petición de refresco que recibía el módulo, de hecho los estados encargados de realizar esta tarea eran inalcanzables. La primera corrección que se hizo fue modificar la máquina de estados añadiendo una nueva transición, con prioridad sobre el resto, desde *dly* hasta *auto_ref_start* La figura 44 muestra la máquina de estados modificada.

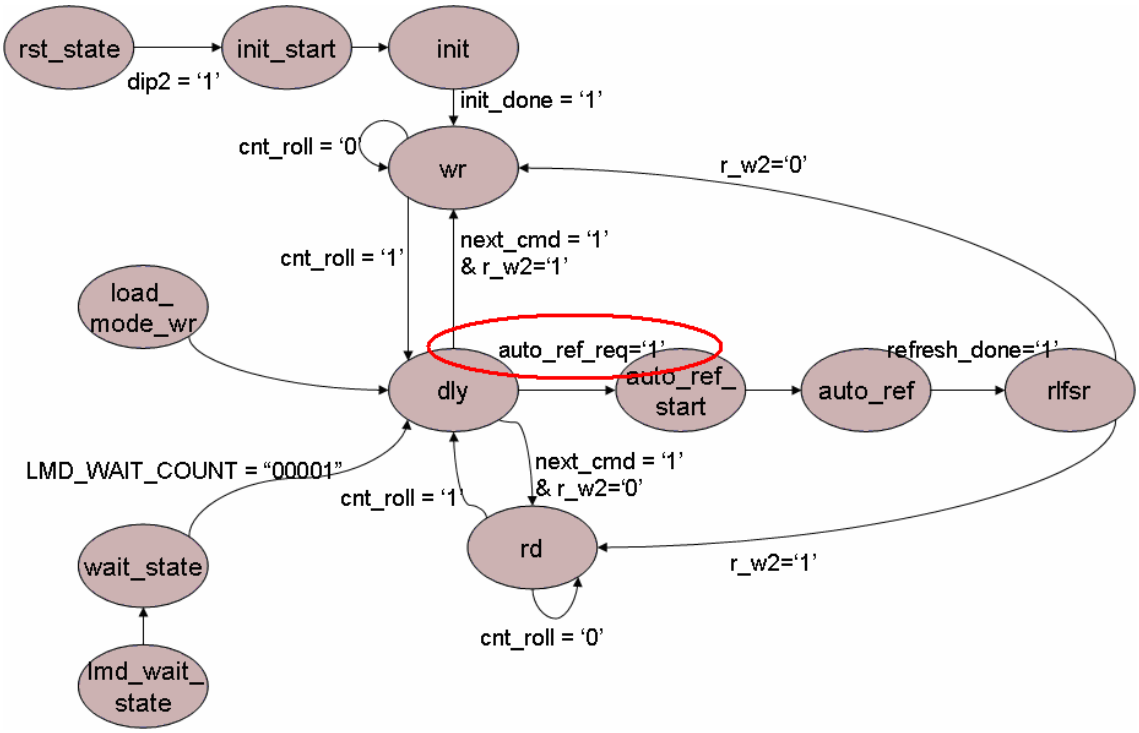


Figura 44: Máquina de estados del módulo cmd_fsm (banco de pruebas) con refresco

La idea de esta modificación es que en cuanto recibiera una petición de refresco se sirviera, antes de iniciar otra operación.

Tras realizar la pertinente prueba se comprobó que seguía funcionando del mismo modo, no se había producido ningún cambio.

3.3 Segunda corrección

Lo siguiente que hicimos, convencidos de que la instrucción de refresco estaba dada, fue analizar como se comportaba el controlador (*controller_64bit_00*) cuando recibía esta operación.

En la documentación de la memoria [11] nos da la definición de los dos tipos de refresco que puede realizar:

- Auto-refresh: comando que se utiliza durante el funcionamiento normal de la memoria. No es persistente por lo que debe ser lanzado cada 7.8µs.
- Self-refresh: comando utilizado para mantener los datos en memoria en caso de ser reseteado o apagado (la fuente de alimentación permanece encendida).

El controlador tiene asignado a auto refresh el código de operación "010" y a self refresh el "011".

Según esto, lo que nosotros necesitamos es una operación auto-refresh por lo que tendremos que enviar al controlador el código "010" para darle la orden de refresco. *Cmd_fsm* enviaba el código de operación de self refresh por lo que lo hemos modificado.

El controlador tiene implementada la operación auto refresh y genera una petición de refresco cada 7.7µs. Aunque el periodo de refresco debe ser como máximo de 7.8µs, lo hemos reducido para que *cmd_fsm* (encargado de dar la orden de operación) tenga un margen de tiempo para finalizar la operación que esté realizando. Es decir, la petición de refresco no se sirve en el momento de ser recibida sino que debe esperar a que termine la operación en curso, de ahí que se envíe antes del tiempo límite.

En la ejecución realizada tras esta segunda modificación no se apreciaron mejoras significativas.

3.4 Tercera corrección

Una vez realizadas las anteriores modificaciones en la entidad cmd_fsm volvimos a revisar nuevamente el código y era coherente. Es decir, partiendo de la suposición de que el controlador (*controller_64bit_00*) estaba bien implementado, todo lo demás hacía lo necesario para que se realizara el refresco. Llegado aquí nos quedamos en punto muerto, no se nos ocurría que podía fallar. Modificamos el código de *lfsr32_64bit* y *cmp_data_64bit* para intentar acotar el fallo pero estas pruebas no tuvieron resultados. En una de las pruebas con *cmp_data_64bit* fijamos el dato a mostrar por el monitor (este componente es el que envía los datos tras comprobar que son válidos) con un valor constante y comprobamos extrañados que el dato visualizado también cambiaba con el tiempo.

Después de esto decidimos utilizar un analizador lógico para poder ver el valor de las señales durante la ejecución en la FPGA.

Una vez aprendido el manejo del analizador procedimos al seguimiento de las señales de control. En las pruebas hechas las señales tenían el valor esperado. El siguiente paso fue muestrear la salida; es decir, los datos leídos de memoria. Aquí hicimos varias ejecuciones porque el tamaño del dato es 128 bits mientras que analizador sólo nos permite sacar 80 bits. Para simplificar la tarea decidimos escribir toda la memoria con el mismo dato, de este modo el valor visualizado sería independiente de la posición de memoria leída y de los bits mostrados.

En todas las ejecuciones dejamos la fpga funcionando unos minutos antes de ver los valores de salida y el resultado fue satisfactorio. El problema era el controlador vga, no la memoria.

A partir de aquí empezamos a trabajar en dos líneas de forma paralela. La primera consistía en reducir la frecuencia de funcionamiento del controlador vga a 50 MHz. (el sistema trabaja a 100 MHz.) y la implementación de un sincronizador para unirlo a la memoria. Más tarde se comprobó que éste no era el problema y se desechó.

La segunda línea de trabajo consistía en la implementación de un nuevo controlador vga. Dado que no teníamos el código del controlador, sólo teníamos el fichero de síntesis .ngo, optamos por hacer un controlador desde cero, totalmente independiente del que habíamos usado hasta entonces.

El nuevo controlador implementado por nosotros podía funcionar a una frecuencia máxima de unos 800 MHz por lo que era compatible con nuestro controlador de DDR SDRAM.

La memoria funcionando con refresco y el controlador vga nuevo está implementado en el proyecto intmemtest2bVGA.

3.5 Diseño del controlador sin banco de pruebas

Una vez realizadas las pruebas oportunas con éxito, el siguiente paso es eliminar el banco de pruebas para que el controlador de memoria pueda funcionar integrado en un sistema operativo. Hay dos posibles opciones para llevar esto a cabo:

- Crear el controlador con el MIG deshabilitando la opción de generar un banco de pruebas.
- Retocar el código existente, con el refresco implementado, para eliminar todo lo que no es necesario.

En el caso de generar un nuevo controlador sin el banco de pruebas nos encontramos con la ventaja de que nos da un código más claro y saca al componente más exterior (*mem_interface_top*), como entradas o salidas, las señales que antes generaba el *ddr1_test_bench_64bit_00*. Sin embargo este método tiene varias desventajas:

- Por un lado, tenemos que modificar el fichero de restricciones porque no tiene en cuenta los dos relojes que necesita la memoria.
- Es necesario escribir mucho código, similar al control de "grano grueso" implementado con el banco de pruebas, para generar las instrucciones que se le deben pasar al controlador.
- Habría que volver a implementar todo lo referente al refresco, punto que ya estaba solucionado en la implementación existente.

Optamos por depurar el controlador existente debido a estas desventajas y a que creímos que sería un proceso más sencillo y rápido.

El trabajo realizado ha consistido en eliminar todos aquellos componentes innecesarios, respetando al máximo posible la estructura existente para evitar problemas en el fichero de restricciones, y limpiar todas aquellas señales que ya no son útiles. Tarea que ha resultado ser más complicada de lo que parecía en un principio, debido a que el banco de pruebas está muy integrado, no es una entidad independiente.

Todas las modificaciones están realizadas sobre la versión intmemtest2c.

Se ha eliminado el módulo *ddr1_test_bench_64bit_00* y se ha sacado a un nivel superior el componente *cmd_fsm*, necesario para dar las instrucciones al controlador en función de las señales recibidas del usuario.

La nueva jerarquía de componentes se muestra en la siguiente figura:

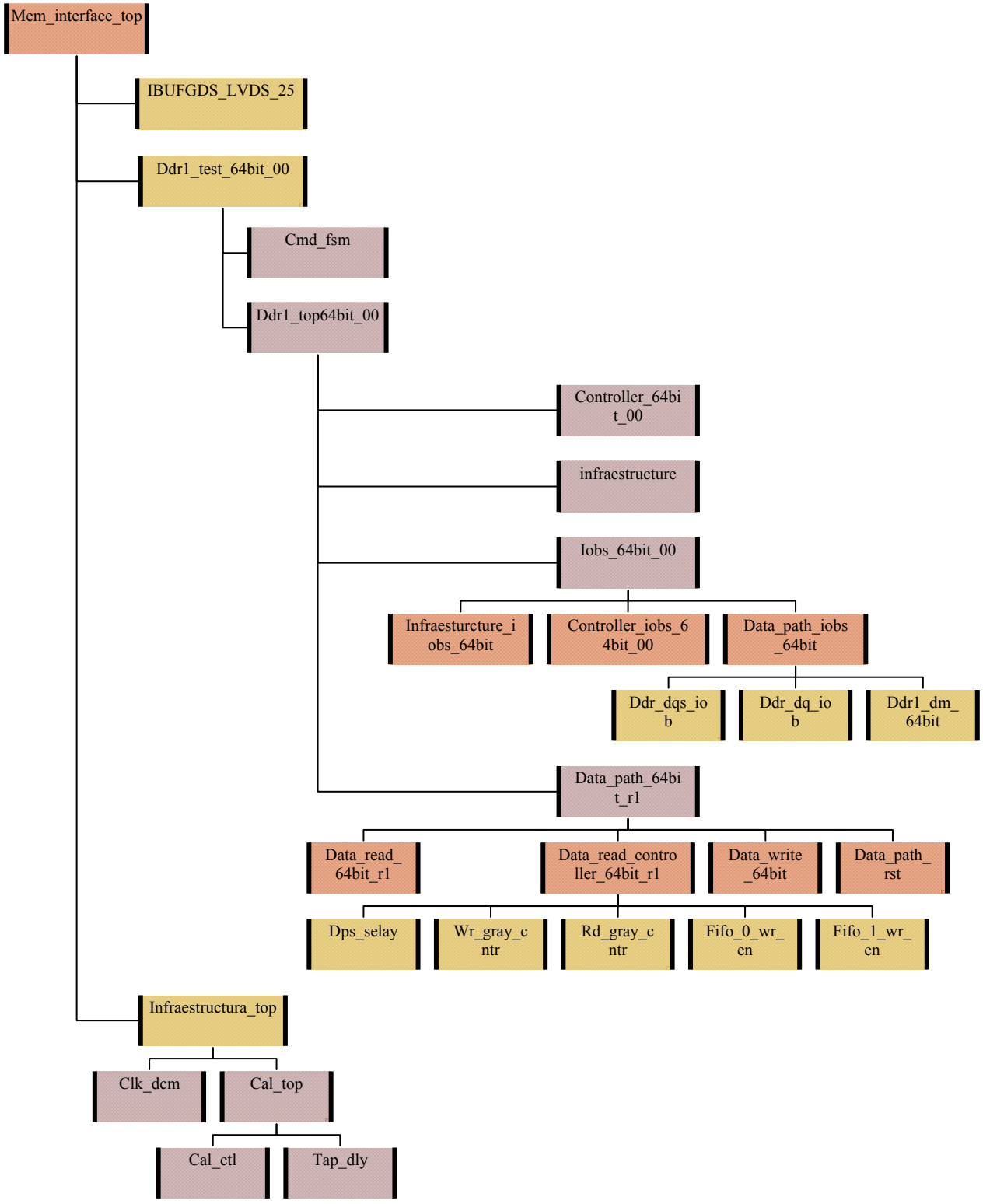


Figura 45: Jerarquía del controlador DDR SDRAM sin banco de pruebas

Las señales que antes generaba y validaba el banco de pruebas y que ahora las da el usuario aparecen como nuevas entradas y salidas en el módulo más externo de la jerarquía (*mem_interface_top*):

- user_output_data: out std_logic_vector(127 downto 0). Dato leído por la memoria. Aunque el tamaño de memoria son 64 bits, trabajamos con un tamaño de 128 porque se realizan dos lecturas o escrituras por ciclo. Es más sencillo considerarlo como una única operación con datos de tamaño doble.

- user_input_data: in std_logic_vector(127 downto 0)**.** Éste será el dato para la escritura de la memoria. La razón de por qué se trabaja con datos del doble de longitud se explica en el punto anterior.

- user_address: in std_logic_vector(24 downto 0). Dirección a partir de la cual se va a realizar una operación de lectura o escritura. Se accederá a tantas posiciones a partir de ésta como indique el tamaño de ráfaga.

- r_w: in std_logic. Señal que indica que operación se va a realizar. Si el valor es '0' indicará una lectura y si es '1' se realizará una escritura.

- enable_r_w: in std_logic. Señal que habilita la memoria. Es activa a alta.

El controlador esta implementado para funcionar con un tamaño de ráfaga 4.

El bloque *mem_interface_top* queda con las siguientes señales de entrada y salida:

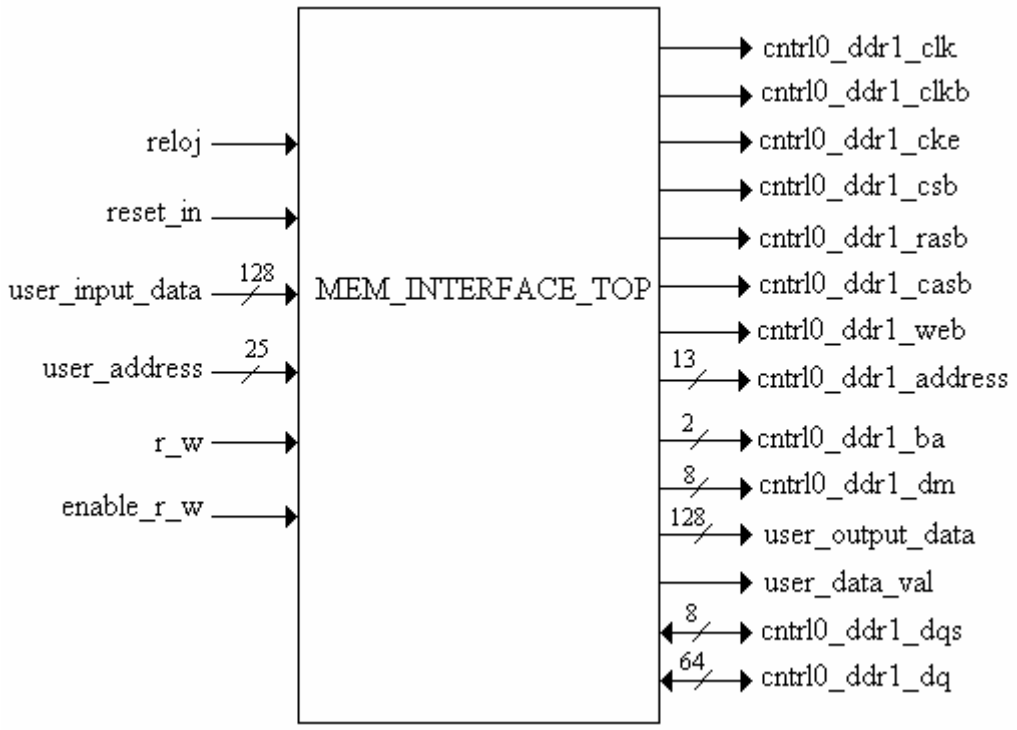


Figura 46: Esquema del controlador de memoria

En las siguientes tablas se explican la utilización de las señales mostradas en el esquema.

Señales de entrada y salida:

Entrada / Salida	Descripción
cntrl0_ddr1_dq	- Bus de datos
cntrl0_ddr1_dqs	- Strobe de los datos

Tabla 6: Entrada/Salida del controlador de memoria

Señales de entrada:

Entrada	Descripción
reset_in	- Señal de reset del sistema
reloj	- Entrada de reloj del sistema
user_input_data[2n-1 : 0]	- Bus de datos de entrada del controlador. - Sólo es válido para la operación de escritura. - El tamaño del bus es 2n, siendo n el ancho del bus de datos de la memoria. El propio controlador es el encargado de convertir el dato doble en dos simples. - De los dos datos, el formado por los bits más significativos se escriben en el flanco de subida y los menos significativos en el de bajada.
user_address [n : 0]	- Dirección de lectura o escritura de la memoria. - La dirección está formada por la fila, la columna y el banco de registros. Los bits más significativos corresponden a la fila y los menos significativos al banco de registros. La columna se encuentra entre estos dos datos.
r_w	- Señal que indica el tipo de operación a realizar. - Si r_w = 0 se hará un lectura. Si por el contrario está a alta realizará una escritura.
enable_r_w	- Señal de capacitación de lectura o escritura. Es activa a alta.

Tabla 7: Entradas del controlador de memoria

Señales de salida:

Salida	Descripción
cntrl0_ddr1_clk	- Señal de reloj
cntrl0_ddr1_clkb	- Señal de reloj invertido
cntrl0_ddr1_cke	- Señal de capacitación del reloj
cntrl0_ddr1_csb	- Señal que capacita el módulo encargado de la decodificación de los comandos.
cntrl0_ddr1_rasb	- Comando
cntrl0_ddr1_casb	- Comando
cntrl0_ddr1_web	- Comando
cntrl0_ddr1_address	- Dirección sobre la que se realizará la operación correspondiente
cntrl0_ddr1_ba	- Dirección del banco de registros.
cntrl0_ddr1_dm	- Máscara de datos para escrituras.
user_output_data	- Bus de datos de salida del controlador. - Sólo es válido para la operación de lectura.. - El tamaño del bus es 2n, siendo n el ancho del bus de datos de la memoria. Esto es debido a que se leen datos en el flanco de subida y de bajada con lo que vuelca al bus dos datos por ciclo.
user_data_val	- Señal que indica que hay un nuevo dato disponible en el bus *user_output_data*.

Tabla 8: Salidas del controlador de memoria

3.5.1 Eliminación de ddr1_test_bench_64bit_00 y modificaciones en cmd_fsm

Como ya he comentado anteriormente es en este bloque donde se han realizado las mayores modificaciones.

Tanto los módulos *addr_gen_64bit_00* y *lfsr32_64bit* se han eliminado porque estos datos (dirección accedida y dato generado aleatorio) nos los proporciona el usuario (sistema operativo). El componente *cmp_data_64bit* tampoco se utiliza. Sólo se conserva el componente *cmd_fsm* que es necesario para generar el código de operación a realizar por el controlador en función de las instrucciones que vienen del exterior.

Debido a que prácticamente todos los componentes y la mayoría de las funcionalidades que se realizan en este módulo no son necesarios se ha eliminado el módulo. Sólo se mantiene *cmd_fsm* que ha subido un nivel en la jerarquía (figura 45).

Dentro de *cmd_fsm* se realizan dos nuevas tareas que anteriormente eran función de *ddr1_test_bench_64bit_00*:

- Se genera el registro de configuración que se utiliza en la etapa de inicialización para determinar cual va a ser el comportamiento de la memoria. El registro de configuración consta de los siguientes campos:

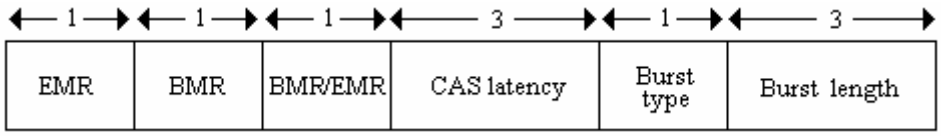


Figura 47: Esquema del registro de configuración

  - EMR: este campo se utiliza para el extended mode register. Su valor sirve para habilitar (‘1’) o deshabilitar (‘0’) el DLL.
  - BMR: se utiliza en el base mode register. Especifica si se utiliza el modo normal de operación (‘0’) o el modo normal con reset DLL.
  - BMR / EMR: bit de selección entre los tipos BMR (‘0’) o EMR (‘1’).
  - CAS latency: es la latencia de lectura. Puede tener tres valores que son 2 (“010”), 3 (“011”) y 4 (“100”), el resto están reservados.
  - Burst type: tipo de acceso al banco de registros. Puede ser secuencial (‘0’) o interpolado (‘1’). El controlador solamente funciona para el tipo secuencial.
  - Burst length: es el tamaño de ráfaga, es decir, el número de registros a los que se va a acceder desde la dirección dada. La longitud puede ser 2 (“001”), 4 (“010”) u 8 (“011”). Este controlador está preparado para realizar accesos con un tamaño de ráfaga 4.

La configuración que se hará de la memoria será para que funcione en modo normal de operación, con una latencia de lectura 3, tipo de ráfaga secuencial y de tamaño 4.

- Generar la señal de control *burst_done*, que necesita *controller_64bit_00* y que antes generaba *addr_gen_64bit_00*. Esta señal le indica al controlador que ya ha

finalizado la transferencia de datos tanto para la lectura como la escritura y éste finaliza la operación. Tras analizar el modo en que se generaba antes vemos que toma el valor '1' un ciclo después de finalizar la escritura/lectura de un bloque. Por esto, el modo en que se debe generar esta señal va muy ligado al tamaño de ráfaga que se da en la inicialización de la memoria (le hemos dado un tamaño 4. Se ha implementado con un contador módulo cuatro.

La máquina de estados, el núcleo de *cmd_fsm*, ha cambiado mucho su apariencia debido a que las señales de control que utilizaba procedían de los módulos eliminados. Respecto al autómata anterior se han eliminado todos aquellos estados inalcanzables que no se utilizaban, para simplificarlo.

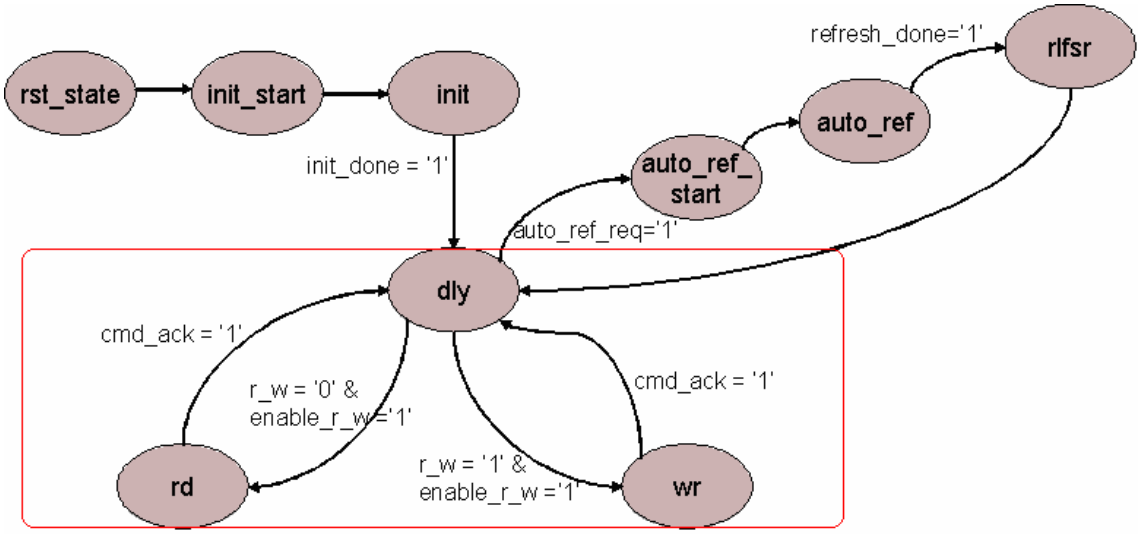


Figura 48: Máquina de estados del módulo cmd_fsm sin banco de pruebas

Los estados correspondientes a la inicialización y refresco no se han tocado, conservando las mismas transiciones.

El principal cambio se encuentra en la zona enmarcada en rojo (figura 48). La máquina de estados recibe tres nuevas señales que son *r_w*, *enable_r_w* y *cmd_ack*. Tanto *r_w* como *enable_r_w* proceden del usuario y hacen referencia a la operación a realizar y su capacitación. Por otro lado, *cmd_ack* procede del controlador interno y nos indica que éste ha reconocido el comando recibido y ha comenzado con su ejecución.

Analizando el diagrama de estados de la figura 48 vemos que se llega al estado dly tras realizar la inicialización de la memoria. Una vez ahí permanecerá a la espera de una petición de refresco (activación de *auto_ref_req*) o de una lectura o escritura (activación de *enable_r_w*).

La señal de refresco siempre tiene prioridad sobre el resto. Una vez que el controlador reciba la señal *refresh_done*, que indica que el refresco se ha concluido con éxito, se vuelve al estado de espera.

Desde los estador *rd* y *wr* se volverá a *dly* cuando reciban la confirmación de la operación del controlador.

Los códigos de operación se envían al controlador desde los estados *ini_startt* ("010"), *auto_ref_start* ("010"), *rd* ("110") y *wr* ("100").

CAPÍTULO 4: PLANIFICADOR DE TAREAS

Actualmente, los sistemas operativos son ampliados con funcionalidades que permiten una utilización más efectiva de la FPGA. Entre estas nuevas funciones se encuentra el algoritmo de planificación y ubicación de tareas en la FPGA [15].

El área de la FPGA se encuentra dividida en cuatro particiones de diferentes dimensiones. El tamaño de estas fracciones viene determinado por la altura del área de ejecución y el ancho de cada una de ellas.

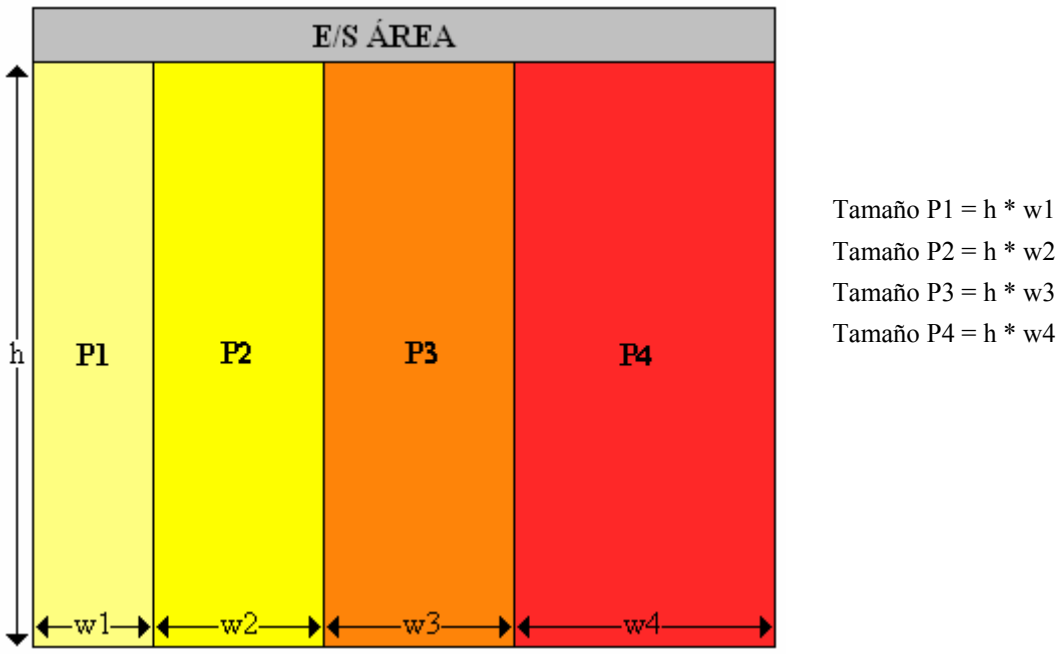


Figura 49: División del área de ejecución

El algoritmo de planificación busca el mejor aprovechamiento del área de la FPGA, basando sus decisiones en el tamaño de la tarea y, en segundo lugar, en las restricciones temporales de la misma.

Las principales características del sistema son:

- Cada partición tiene asociada una cola en la cual se alojan las tareas a la espera de ser ejecutadas.
- Solamente se puede ejecutar una tarea por partición en cada momento.
- Posibilidad de usar más de una partición para ejecutar una tarea cuando ésta no cabe en una sección simple.
- Las tareas entrantes al planificador son tuplas que contienen toda la información necesaria para su planificación y posterior ubicación.
- El mapa de bits que conforma la tarea se encuentra almacenada en una RAM. El acceso a los datos y su carga en la FPGA se realiza a través de un cargador de tareas.

4.1 Descripción

El planificador es el módulo encargado de la gestión de las tareas que hay en el sistema operativo. Tiene dos funciones que son decidir a cuál de las cuatro particiones del área de ejecución irán las tareas entrantes (planificar) y lanzarlas a ejecución en función del estado de cada partición de la FPGA (ubicar).

El planificador está diseñado para un entorno multitarea en FPGAs reconfigurables con divisiones 1D. Se considera que el área de ejecución está dividida en cuatro particiones, de diferente tamaño e independientes entre si, en las que se ejecutarán las tareas.

El área de la FPGA destinado a la ejecución será aproximadamente el 40 %, dejando la otra parte para la circuitería del sistema operativo. Los tamaños de las particiones será: w1 = 6, w2 = 10, w3 = 12 y w4 = 16.

El objetivo del algoritmo de planificación es el aprovechamiento máximo del área de la FPGA por lo que se utiliza una política de mejor ajuste, es decir, las tareas se insertarán en la menor de las particiones en la que quepa.

Otra característica importante es la posibilidad de ejecutar una tarea en más de una partición cuando ésta no cabe en una sección simple. En este caso, será necesario sincronizar las colas asociadas a las particiones para que lancen la tarea de forma simultánea. La sincronización de las colas se realiza mediante la inserción de gaps.

Un gap es lo mismo que una no operación. Veamos como funciona la sincronización de las colas:

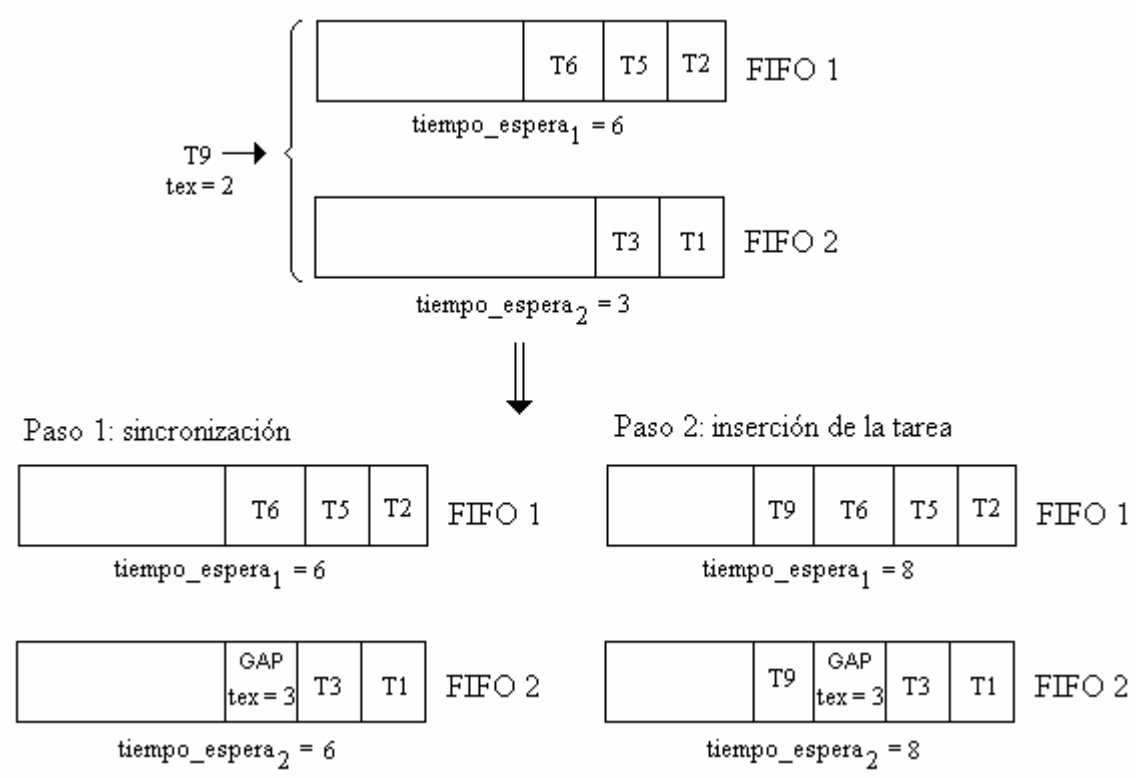


Figura 50: Combinación de particiones

En un primer paso se introduce un gap en la cola con menor tiempo de espera (en la figura 50 sería la fifo 2) para igualar dichos tiempos. A continuación se introduce la tarea en ambas colas.

Las tareas se planificarán en el instante de su llegada. Dicha planificación se realizará de una en una por lo que será necesario esperar a la activación de la señal task_ack o task_reject para enviar una nueva tarea al planificador.

Sin embargo, debido a que las tareas tardan un tiempo en ser leídas y a que la FPGA necesita un tiempo de reconfiguración para cada nueva instrucción recibida, la ejecución se realizará en slots de 32 ciclos. Es decir, se pretende reducir la resolución para poder despreciar los ciclos de reconfiguración y lectura, simplificando así la sincronización de las colas. Ejemplo: si una tarea tiene un tiempo de ejecución de 20 ciclos, el tiempo de espera se incrementará en un slot, teniendo un margen de 12 ciclos para la lectura y reconfiguración.

Tanto los tiempos de espera y ejecución como el tiempo máximo de ejecución serán considerados en slots, no en ciclos de reloj.

4.1.1 Esquema

A continuación se muestra el bloque del planificador como una caja negra con sus entradas y salidas:

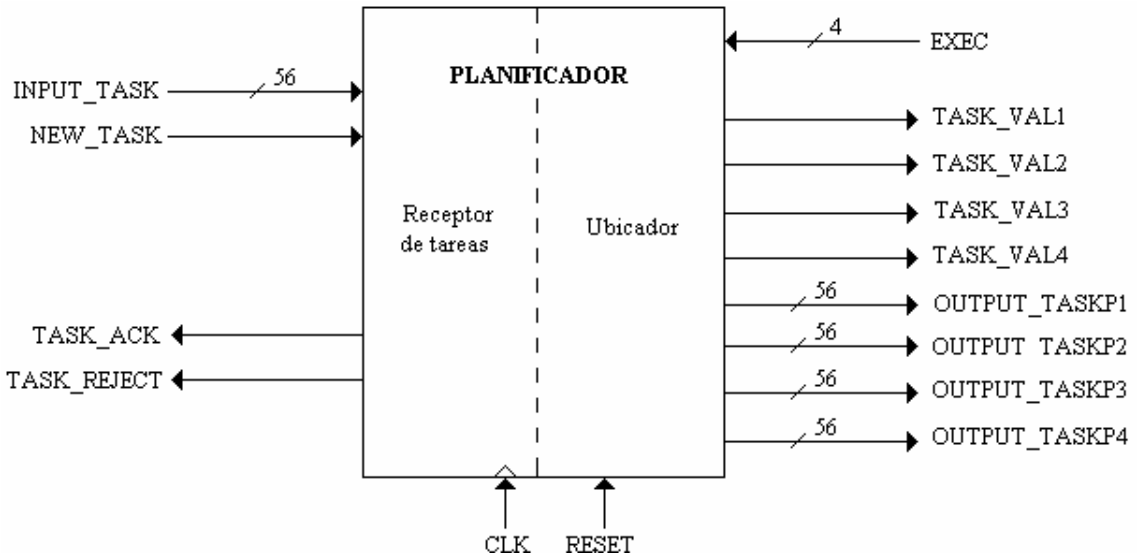


Figura 51: Esquema del módulo planificador

En la figura 51 se aprecian las señales utilizadas en cada una de las dos funciones del planificador. Por un lado tenemos el receptor de tareas que es el que se encarga de realizar la planificación y por otro lado se muestran las señales encargadas de la comunicación con el cargador de tareas.

En la tabla se muestra el significado de cada una de las señales mostradas en el esquema:

Entrada	Descripción
clk	- Señal de reloj del módulo.
reset	- Señal de reset asíncrono.
exec	- Seña que indica si la partición asociada a cada uno de sus bits está ejecutando una tarea o se encuentra en reposo. El bit menos significativo corresponde a la partición uno y el más significativo a la cuarta.
input_task	- Tarea a planificar.
new_task	- Señal que indica que hay un nuevo dato en la entrada del planificador.

Tabla 9: Entradas del planificador

Salida	Descripción
task_ack	- Señal que indica que la tarea ha sido planificada.
task_reject	- Señal que indica que la tarea ha sido rechazada.
task_val1 / task_val2 task_val3 / task_val4	- Estas señales se activan para indicar que hay una nueva tarea disponible en el bus output_task correspondiente.
Output_task1 Output_task2 Output_task3 Output_task4	- Cada uno se estos buses está asociado a una de las particiones del área de ejecución. En ellos se da la siguiente tarea a ejecutar.

Tabla 10: Salidas del planificador

La forma en que el planificador queda conectado con la FPGA se muestra en la siguiente figura:

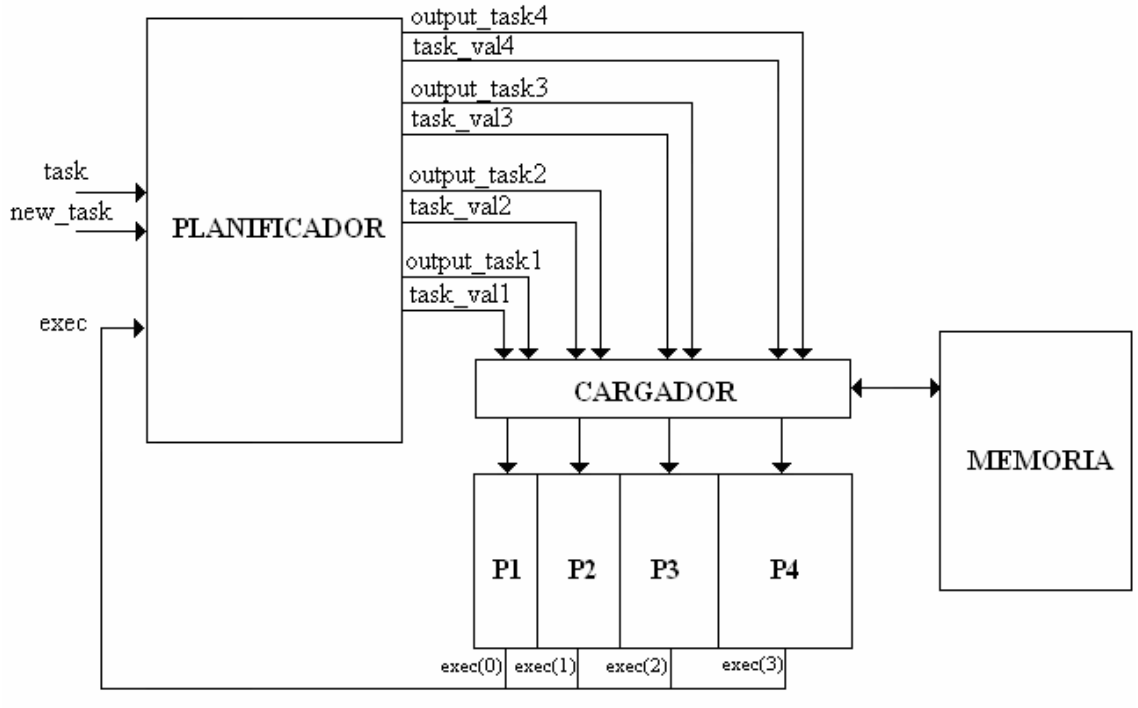


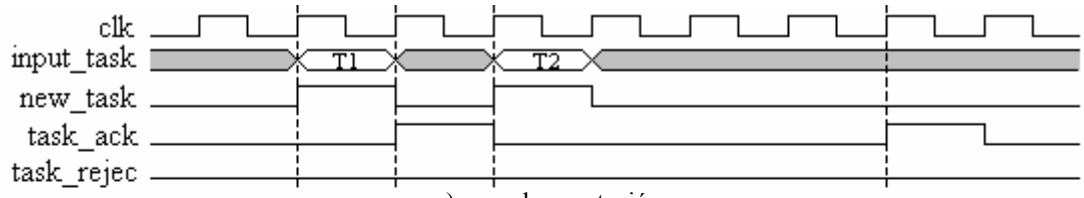
Figura 52: Conexiones entre el planificador y el entorno

Como se comentó anteriormente, la tarea entrante al planificador tiene toda la información necesaria para la planificación y los datos para la identificar la posición de la memoria en que se encuentra el mapa de bits.

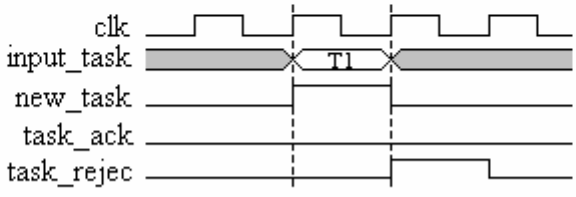
El cargador recibe la información necesaria para extraer las tareas de la memoria y realizar la carga de las mismas en las particiones correspondientes.

4.1.2 Protocolo de planificación

El protocolo de planificación es muy sencillo debido a que hay muy pocas señales implicadas.



a) caso de aceptación



b) caso de rechazo

Figura 53: Protocolo de planificación de tareas

La tarea a planificar deberá estar disponible en el momento en que se activa la señal *new_task* siendo indiferente el valor que tome cuando ésta esté a cero.

Tal como se puede apreciar en los diagramas a) y b) (figura 53) las señales *task_ack* y *task_rejec* son disjuntas, es decir, solamente una se activará en respuesta a *new_task*.

Si una tarea es aceptada (a), la señal de aceptación se activará al ciclo siguiente de detectar una nueva tarea, si se asigna a una partición simple, o cuatro ciclos más tarde, si se produce una fusión de particiones.

En caso de rechazo, task_rejec se activará siempre al ciclo siguiente de entrar la tarea.

Las tareas tienen que ser planificadas de una en una por lo que será necesario esperar a la activación de una de las dos señales de salida antes de volver a introducir una tarea nueva.

4.1.3 Protocolo de ubicación

Tal como se ha explicado en el apartado de descripción del planificador, el tiempo de ejecución de las tareas viene dado en slots de 32 ciclos, aunque el tamaño de slot es configurable, dado que, a este nivel, no tiene sentido hacerlo a nivel de ciclo. A efectos prácticos, esto significa que sólo se lanzarán tareas cuando comience un slot.

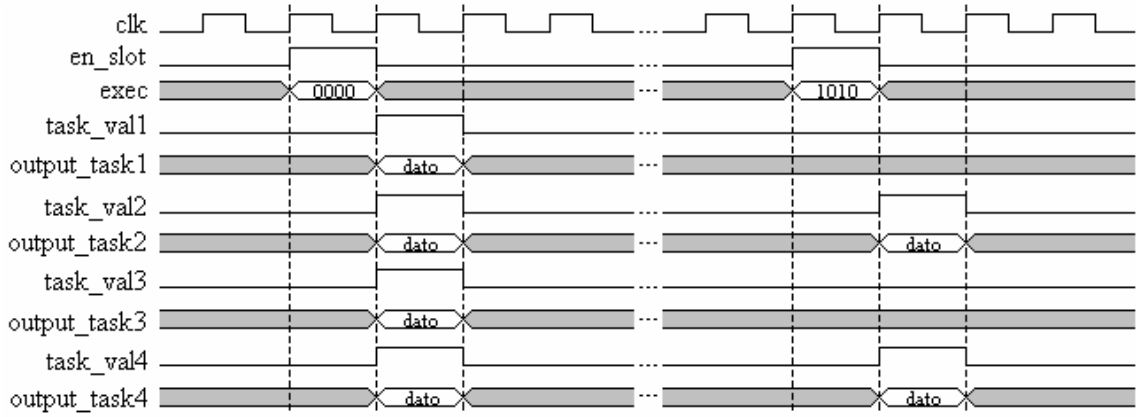


Figura 54: Protocolo de ubicación de tareas

La señal *exec* varía durante un slot pero el único valor que se tiene en cuenta es el que tiene cuando se activa la señal *en_slot* que indica el comienzo de slot.

EL dato válido aparece en los buses *output_task* únicamente cuando la señal *task_val* correspondiente aparece en alta.

El bit menos significativo de *exec* se corresponde con el estado de la partición uno y el más significativo con el de la cuatro. Solamente se vuelca un nuevo dato en las salidas correspondientes a las particiones en reposo. Por ejemplo, cuando exec vale 1010 aparece un nuevo dato en las señales correspondientes a las particiones dos y cuatro.

4.1.4 Formato de las tareas

El dato que se recibe a través de la señal input_task contiene toda la información necesaria para planificar la tarea, pero no incluye el mapa de bits..

La tarea tiene una longitud de 56 bits, los cuales se dividen en los siguientes campos:

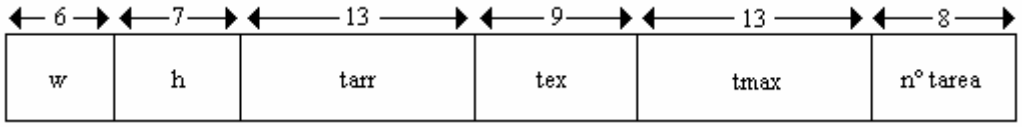


Figura 55: Campos en que se divide una tarea

- w: Este campo indica el ancho que ocupa la tarea.
- h: Este campo codifica la longitud que ocupa la tarea. En este caso, al encontrarnos frente a un problema de 1D, todas las particiones tienen el mismo valor, que será la longitud del área de ejecución, por lo que no se utilizará para la planificación. Se asume que todas la tareas tendrán un valor igual o menor al permitido.
- tarr: Indica el ciclo en que la tarea llega al planificador.
- tex: Tiempo de ejecución de la tarea.
- tmax: Ciclo de dead line o tiempo máximo para que la tarea finalice.
- nº tarea: Indica el número de la tarea. Van numerados por orden de llegada. Este campo se ha añadido para la depuración del módulo y para poder visualizar, de una forma clara, la tarea en ejecución a través de un monitor, mediante el controlador de vga. En caso de conectarse el módulo planificador con un cargador de tareas, también servirá como número de referencia para identificar el trabajo.

4.2 Arquitectura

Cada una de las particiones de la FPGA puede ejecutar en cada momento una única instrucción, por lo que cada una de ellas tendrá asociada una cola FIFO que le irá sirviendo las tareas secuencialmente.

Cada una de estas colas tendrá asociado un registro que almacena el tiempo de espera, es decir, los ciclos que tendrá que esperar para ser lanzada a ejecución. Este tiempo será la suma de los tiempos de ejecución de las tareas que están dentro de la cola más el tiempo que queda para que finalice la tarea que se está ejecutando en la partición asociada.

Los principales componentes del planificador se muestran en la figura 56:

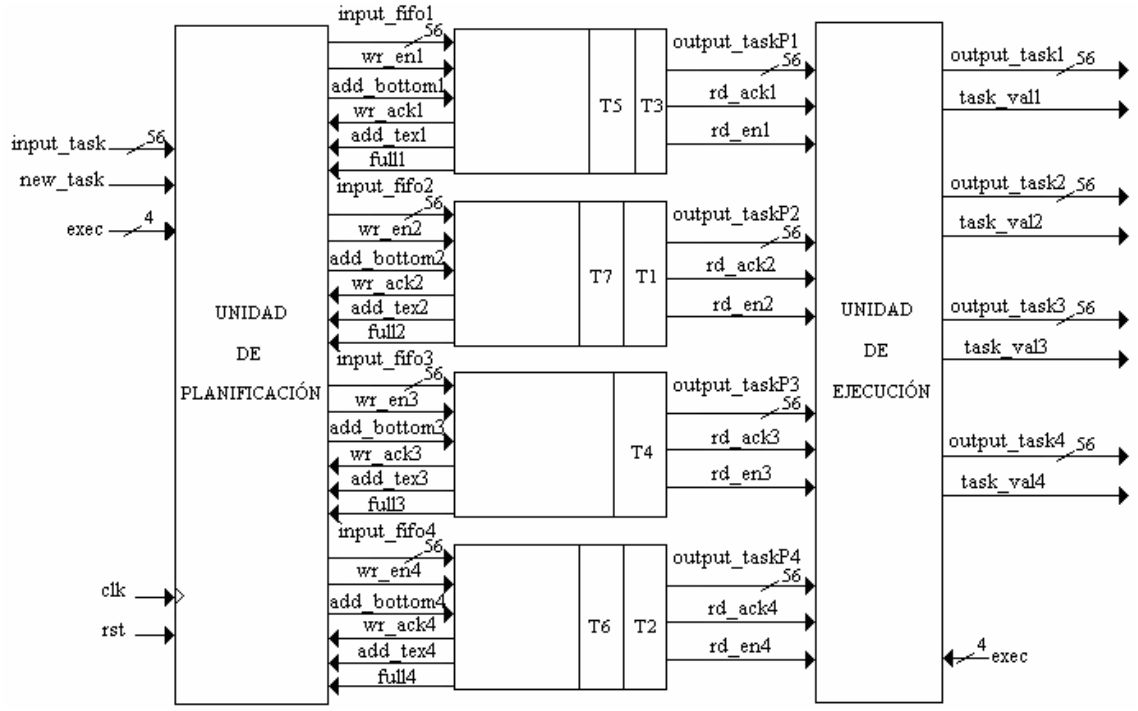


Figura 56: Visión general del planificador

Como se puede comprobar en la imagen superior, el planificador tiene tres componentes principales que son: unidad de planificación, unidad de ejecución y una cola fifo por partición. Se explicará con más detalle cada uno de los módulos en los siguientes apartados.

La unidad de ejecución y la de planificación trabajan en paralelo y son independientes entre si, siendo posible gestionar por separado las tareas entrantes y las salientes.

Los tamaños de las particiones del área de ejecución las he elegido de forma que haya una pequeña, dos medias y otra mayor para cubrir un amplio espectro de tamaños desaprovechando el menor área de FPGA posible, en función a mi banco de pruebas. Estos tamaños están declarados en el código como constantes por lo que se pueden modificar sencillamente sin afectar al diseño. En un futuro se puede ampliar la arquitectura añadiendo una unidad de ajuste dinámico que modifique los tamaños en función al patrón de tareas utilizado

4.2.1 Unidad de planificación

Como su nombre indica es el encargado de decidir en cual de las particiones se ejecutará cada tarea, basándose en las restricciones espaciales y temporales de la misma. En caso de que no sea posible cumplir ambas condiciones se rechazará.

Dentro de este módulo se encuentran los registros con los tiempos de espera asociados a las fifos; tiempos necesarios para comprobar si una tarea cumple las restricciones temporales impuestas por su tiempo de ejecución y su dead line.

Es un bloque secuencial compuesto por una unidad de control, cuyo núcleo es una máquina de estados, y una ruta de datos segmentada en tres partes.

4.2.1.1 Entadas y salidas

Entrada	Descripción
clk	- Señal de reloj del módulo.
reset	- Señal de reset asíncrono.
exec	- Seña que indica si la partición asociada a cada uno de sus bits está ejecutando una tarea o se encuentra en reposo. El bit menos significativo corresponde a la partición uno y el más significativo a la cuarta. - Es utilizada para el control del tiempo de espera.
input_task	- Tarea a planificar.
new_task	- Señal que indica que hay un nuevo dato en input_task
add_tex1 add_text2 add_tex3 add_tex4	- Estas señales, procedentes de las fifos, indican que se ha añadido una tarea nueva al final de la cola y que, por lo tanto, hay que añadir el tiempo de ejecución de la nueva tarea al tiempo de espera correspondiente.
full1 / full2 full3 / full4	- Señales procedentes también de las colas indican si están completas o no.
wr_ack1 wr_akc2 wr_ack3 wr_ack4	- Confirmación de escritura enviada por las colas como respuesta a una petición de escritura

Tabla 11: Entradas de la unidad de planificación

Salida	Descripción
input_fifo1 input_fifo2 input_fifo3 input_fifo4	- Dato que se envía a las colas para su escritura. Cada uno de ellos se pasa a su correspondiente fifo, según indica su número.
add_bottom1 add_bottom2 add_bottom3 add_bottom4	- Estas señales indican a las colas si el dato que se va a escribir debe ser añadido al final de las mismas o si, por el contrario, se puede intentar insertar en hueco existente. Cada uno de ellos es entrada de su correspondiente fifo, según número.
wr_en1 wr_en2 wr_en3 wr_en4	- Señales de enable de escritura que se envía a las colas. Cada una de ellas va a su correspondiente fifo.

Tabla 12: Salidas de la unidad de planificación

4.2.1.2 Unidad de control

La máquina de estados que controla el módulo es la siguiente:

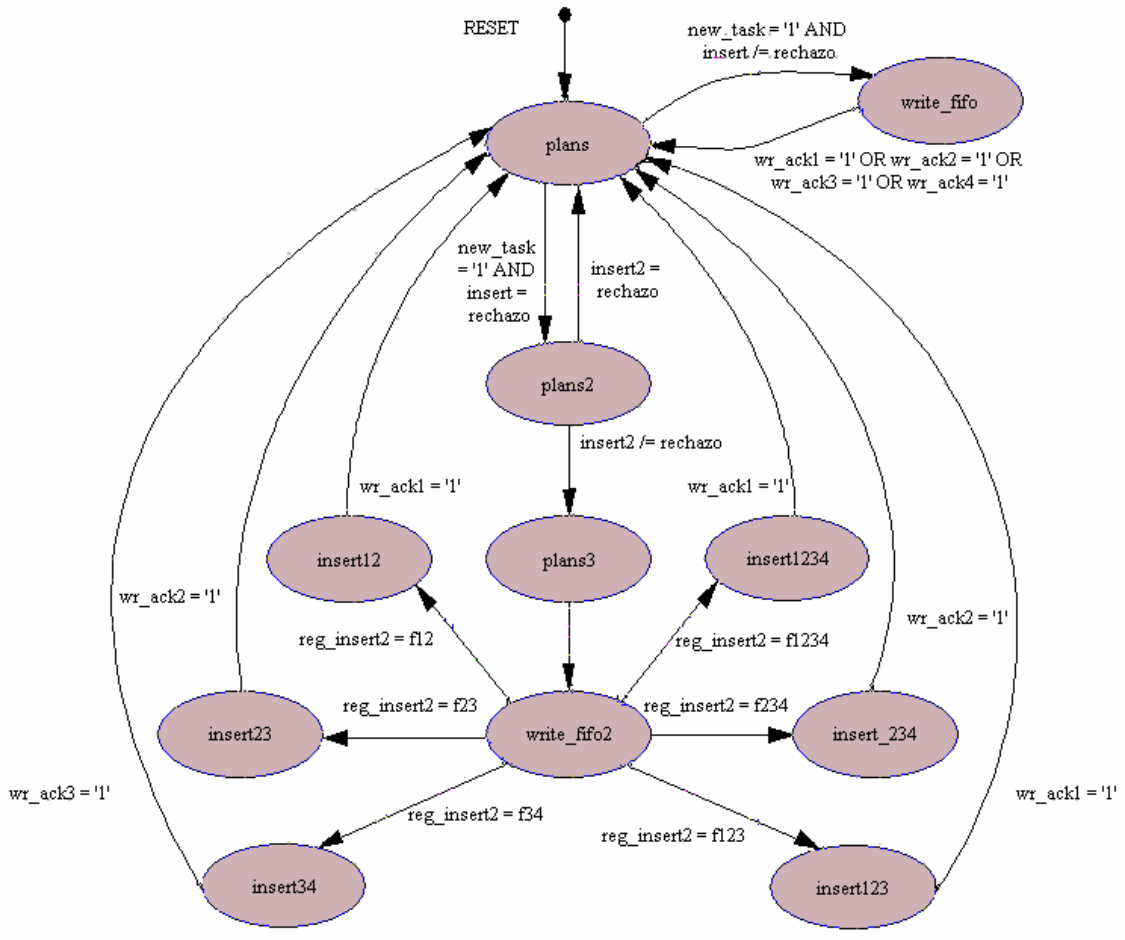


Figura 57: Maquina de estados de la unidad de planificación

Todas las funcionalidades del planificador se encuentran en esta máquina. Se ha diseñado con el objetivo de que tarde el menor número de ciclos posible en la toma de decisión. Estará en el estado inicial, *plans*, hasta que llegue una nueva instrucción; a partir de este momento, empezará la toma de decisiones.

En este módulo también es el responsable de hacer la conversión de ciclos a slots de tex y tmax.

Cada vez que entra una tarea nueva pueden pasar tres cosas:

- Que la tarea entre en una de las cuatro particiones. En este caso, durante el primer ciclo se comprueba cuál es la fracción más pequeña, no llena, que cumple las restricciones de espacio y tiempo. Durante el segundo ciclo, ya en el estado *write_fifo*, se realiza la escritura en la cola correspondiente y se espera la confirmación de la operación para volver al estado inicial.

Únicamente en este caso la señal *add_bottom* tendrá un valor '0', que significa que la tarea puede ser insertada en un gap. Esto se debe a que no tiene que estar sincronizada con ninguna otra cola y por lo tanto da igual el lugar en que se ejecute.

- Que la tarea no cumpla una de las restricciones, bien de espacio, bien de tiempo, de una partición y haya que combinar más de una fracción. En este caso, durante el primer ciclo se hacen dos comprobaciones en paralelo, por un lado la comprobación de que no cabe en ninguna partición, y por otro se calcula el tiempo de espera máximo de todas las combinaciones posibles. En el segundo ciclo, estado *plans2*, se comprueba cuál es la menor combinación, en cuanto a espacio, en la que se puede insertar la tarea. Durante el tercer ciclo, *plans3*, se calculan los tiempos de diferencia entre el mayor tiempo de espera de todas las colas implicadas y el resto. En el cuarto, *write_fifo2*, se escriben los gaps necesarios para la sincronización. Finalmente, en el quinto y último ciclo se escribe la tarea en todas las colas seleccionadas. Vuelve al estado inicial al recibir la señal de acknoledge de cualquiera de las colas ya que todas las escrituras tardan los mismos ciclos.

Sólo se pueden combinar aquellas particiones adyacentes debido al acceso al bus de datos. En este caso las combinaciones permitidas son P1+P2, P2+P3, P3+P4, P1+P2+P3, P2+P3+P4 y P1+P2+P3+P4.

- Que la tarea sea rechazada. En *plans*, se comprueba que no cabe en ninguna de las colas simples y en *plans2* se ve que no hay ninguna combinación posible en la que pueda encajar la tarea, activando la señal de rechazo y volviendo al estado inicial.

4.2.1.3 Ruta de datos

En el diseño de este módulo he tenido bastantes problemas por motivos de la frecuencia máxima de funcionamiento permitida. La causa era la gran cantidad de comparaciones que es necesario realizar y el consecuente retardo provocado por la lógica combinacional. Tras la síntesis de una primera implementación se obtuvo una frecuencia de funcionamiento de tan sólo 30 MHz. A continuación se dividió en dos partes pero se seguía sin alcanzar la frecuencia deseada. Ha sido necesario segmentar la ruta de datos en tres partes para conseguir una frecuencia superior a los 100 MHz.

El diseño final es una ruta de datos segmentada en tres partes. Para mayor claridad, se muestra, a continuación cada una de estas fases por separado.

Primera fase:

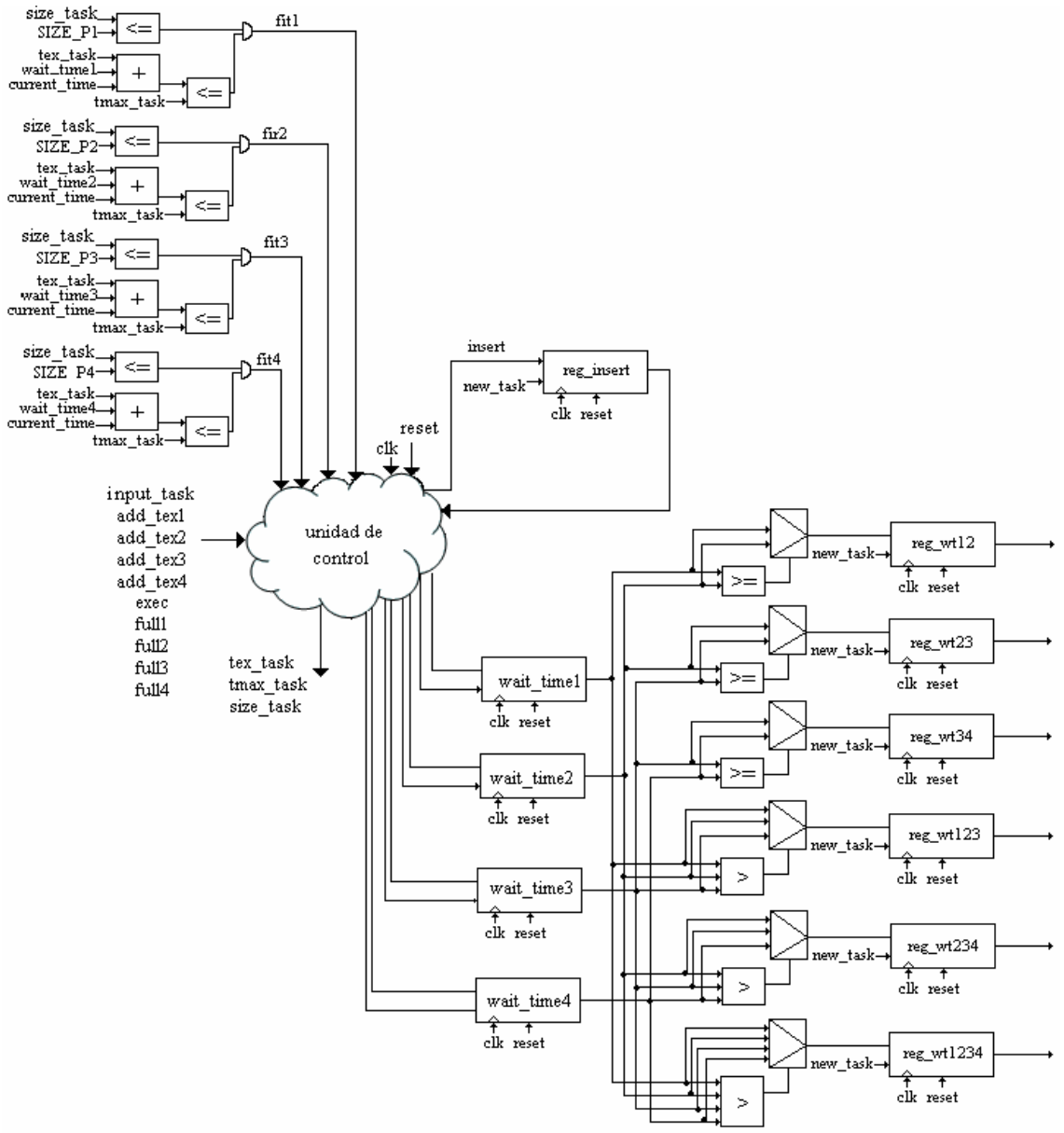


Figura 58: Primera etapa de la ruta de datos de la unidad de planificación

En este esquema consta de dos partes. En la primera se comprueba, de forma simultánea, si la tarea encaja en cada una de las particiones simples. Como puede darse el caso de que se active más de una de las líneas de selección (si la tarea cabe por tamaño en una de las particiones pequeñas es muy posible que también entre en las grandes), la unidad de control selecciona la partición más pequeña, cuya cola no esté llena, para la cual la señal *fit* está activa.

Para simplificar la lógica y reducir el tiempo de planificación se considera que hay diez colas a las que se puede mandar las tareas, las cuatro simples y otras seis virtuales que son todas las posibles fusiones de las reales (P1+P2, P2+P3, P3+P4, P1+P2+P3, P2+P3+P4 y P1+P2+P3+P4).

Para poder tratar las colas virtuales del mismo modo que las reales en necesario saber su tiempo de espera (necesario para ver si cumple las restricciones temporales de la tarea). El tiempo de espera será el mayor de los tiempos de las colas reales que forman la combinación, por ejemplo para la cola23 su tiempo de espera será el mayor de *wait_time2* y *wait_time3*.

La segunda parte de la figura 58 hace las comparaciones necesarias para el cálculo de los tiempos de espera de las colas virtuales, registros *wait_time*. Estos valores se calculan para la siguiente etapa, que los necesitará si la tarea no encaja en una partición simple.

Segunda fase:

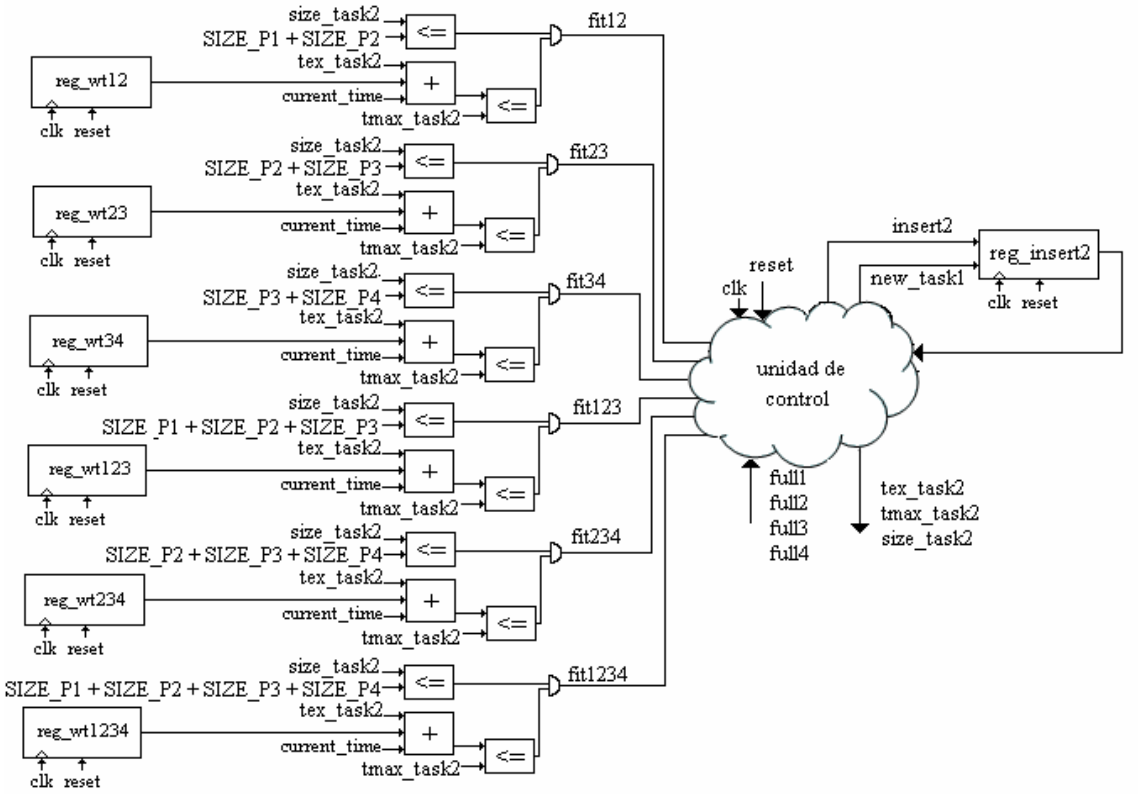


Figura 59: Segunda etapa de la ruta de datos de la unidad de planificación

En esta etapa se comprueban las restricciones para todas las colas virtuales en paralelo. La unidad de control selecciona la combinación de menor tamaño total cuyas colas no estén llenas y que cumpla con todas las exigencias.

El valor obtenido en esta etapa, *reg_insert2*, solamente será tenido en cuenta si el estado de la unidad de control es *plans2*.

Tercera fase:

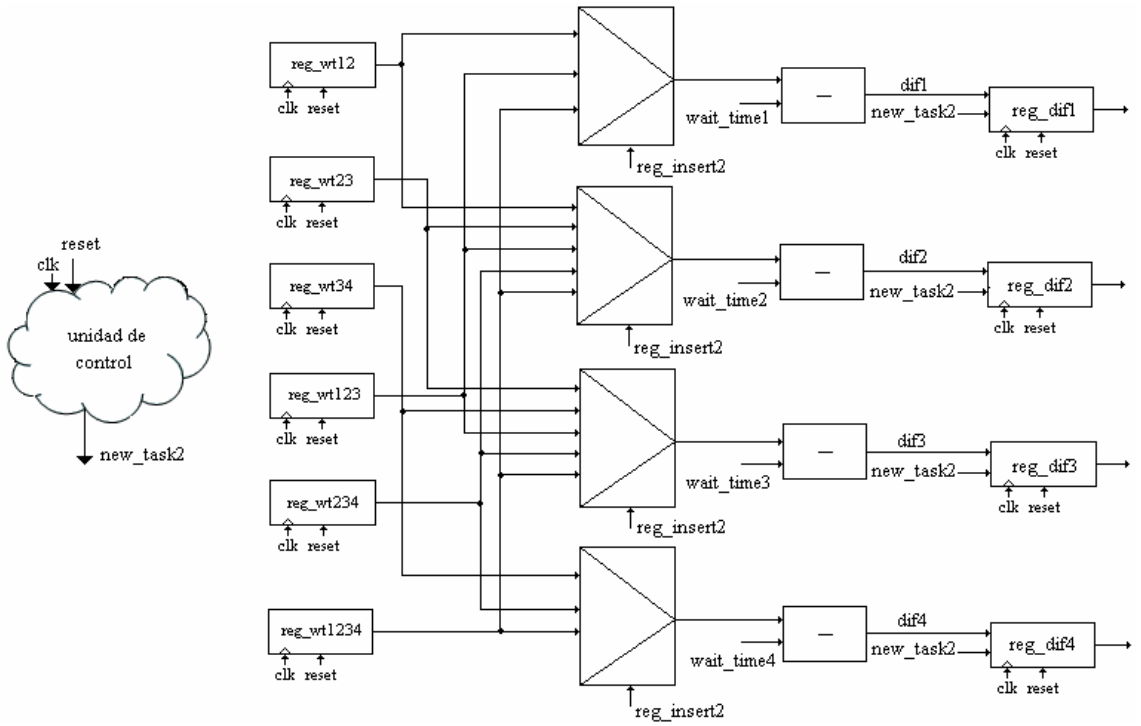


Figura 60: Tercera etapa de la ruta de datos de la unidad de planificación

En función de la combinación seleccionada se calcula la diferencia de tiempos que habrá que insertar en la colas/s con menor tiempo de espera, en forma de gap.

Finalmente, en función de estos últimos datos, los gaps y el valor de *reg_insert2*, se activarán unas señales de escritura u otras.

El valor que tomarán las salidas se muestran en las tablas 13 y 14.

Estado	Add_bottom1	Add_bottom2	Add_bottom3	Add_bottom4	Input_fifo1	Input_fifo2	Input_fifo3	Input_fifo4
Plans	-	-	-	-	-	-	-	-
Plans2	-	-	-	-	-	-	-	-
Plans3	-	-	-	-	-	-	-	-
Write_fifo	0	0	0	0	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)
Write_fifo2	1	1	1	1	“111111” & reg_task(49..30) & reg_dif1 & reg_task(20..8) & “11111111”	“111111” & reg_task(49..30) & reg_dif2 & reg_task(20..8) & “11111111”	“111111” & reg_task(49..30) & reg_dif3 & reg_task(20..8) & “11111111”	“111111” & reg_task(49..30) & reg_dif4 & reg_task(20..8) & “11111111”
Insert12	1	1	-	-	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	-	-
Insert23	-	1	1	-	-	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	-
Insert34	-	-	1	1	-	-	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)
Insert123	1	1	1	-	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	-
Insert234	-	1	1	1	-	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)
Insert1234	1	1	1	1	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)	reg_task(55..30) & ”0000” & tex_task2 & reg_task(20..0)

Tabla 13: Salidas Moore de la unidad de planificación

Estado	Wr_en1	Wr_wn2	Wr_en3	Wr_en4
Plans	0	0	0	0
Plans2	0	0	0	0
Plans3	0	0	0	0
Write_fifo	si reg_insert = f1 → 1	si reg_insert = f2 → 1	si reg_insert = f3 → 1	si reg_insert = f4 → 1
Write_fifo2	si insert_gap1 = 1 → 1	si insert_gap2 = 1 → 1	si insert_gap3 = 1 → 1	si insert_gap4 = 1 → 1
Insert12	1	1	0	0
Insert23	0	1	1	0
Insert34	0	0	1	1
Insert123	1	1	1	0
Insert234	0	1	1	1
Insert1234	1	1	1	1

Tabla 14: Salidas Mealy de la unidad de planificación

4.2.2 Unidad de ejecución

La unidad de ejecución comprueba el estado de cada partición y, cuando alguna pasa a estar en reposo, genera una petición de lectura a la cola fifo correspondiente. En cuanto esa lectura es confirmada envía la nueva tarea para que sea ejecutada.

Este módulo recibe únicamente las señales *exec* y en_*slot*. La primera ya ha sido explicada con anterioridad. En cuanto a la segunda, está generada por un contador que la activa cada 32 ciclos de reloj e indica que se inicia un nuevo slot, y por tanto se puede lanzar una nueva tarea.

El funcionamiento es el siguiente: si no hay ninguna tarea ejecutándose en alguna de las particiones y comienza un nuevo slot se realiza una lectura en todas aquellas colas cuyas particiones están en reposo.

Como se puede ver en la figura 51, hay una línea de petición de lectura y de datos por cada partición por lo que es posible servir tareas a más de una fracción simultáneamente.

4.2.3 Cola fifo

Junto a la unidad de planificación, es una de las partes más importantes de este módulo.

Aunque sigue la política de primero en entrar, primero en salir, no se comporta exactamente como una fifo normal, ya que cuenta con algunas características especiales para poder sacar más partido a algunas de las funciones del planificador.

4.2.3.1Esquema

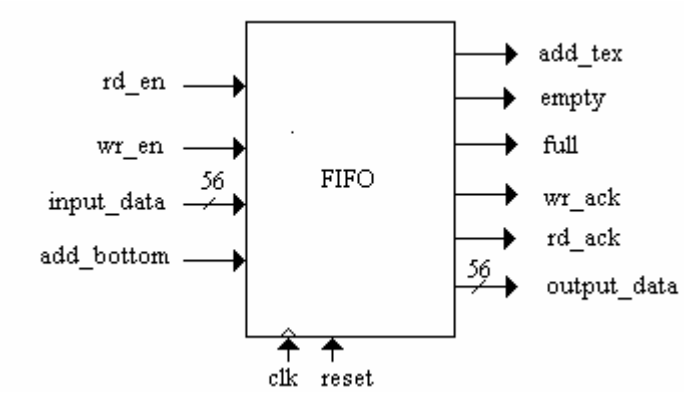


Figura 61: Esquema de una cola fifo

Las entradas y las salidas son las siguientes:

Entrada	Descripción
clk	- Señal de reloj del módulo
reset	- Señal de reset asíncrono
wr_en	- Señal de petición de escritura
rd_en	- Señal de petición de lectura
input_data	- Dato a introducir en la cola
add_bottom	- Señal que indica si el dato debe ser introducido al final de la cola o si se puede intentar meter en un gap

Tabla 15: Entradas de la fifo

Salidas	Descripción
add_tex	- Señal que indica si se ha añadido la tarea al final y por tanto hay que incrementar el tiempo de espera de la cola.
full	- Señal que indica si la cola está llena
empty	- Señal que indica si la cola está vacía
wr_ack	- Señal de finalización de escritura
rd_ack	- Señal que indica la finalización de la lectura y la validez del dato disponible en el bus *output_data*
output_data	- Dato de salida

Tabla 16: Salidas de la fifo

La arquitectura general de la fifo es la que se muestra en la figura presentada a continuación.

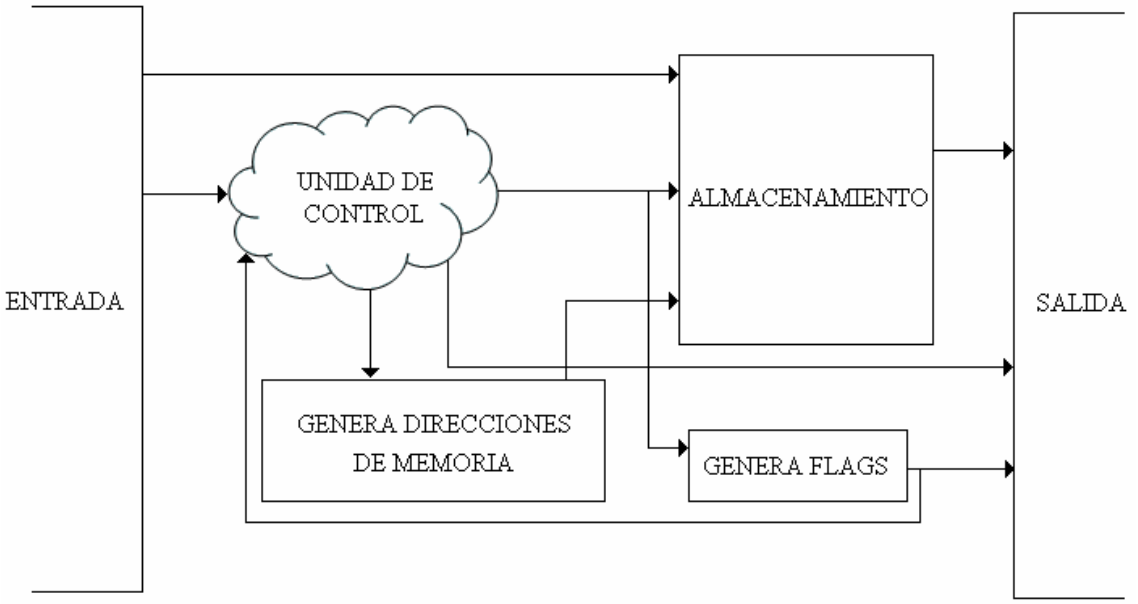


Figura 62: Arquitectura general de la cola fifo

La unidad de control contiene una máquina de estados, en función de la cual se generan las señales de control y las salidas de confirmación. El almacenamiento consiste en una BRAM de doble puerto, que permite leer y escribir en el mismo ciclo; tiene una capacidad de 256 datos, es decir, está preparada para alojar 256 tareas. El generador de flags consiste en un contador ascendente-descendente con el número de elementos que hay en todo momento el la cola y genera las señales de lleno y vacío. En cuanto al generador de direcciones indica cual es la siguiente dirección a leer y a escribir.

Debido a las funcionalidades extras impuestas por la especificación del módulo y del planificador, la fifo ha sido rediseñada en diversas fases del desarrollo del proyecto.

4.2.3.2 Primera fase de diseño

El primer paso fue diseñar una cola fifo simple (figura 63).

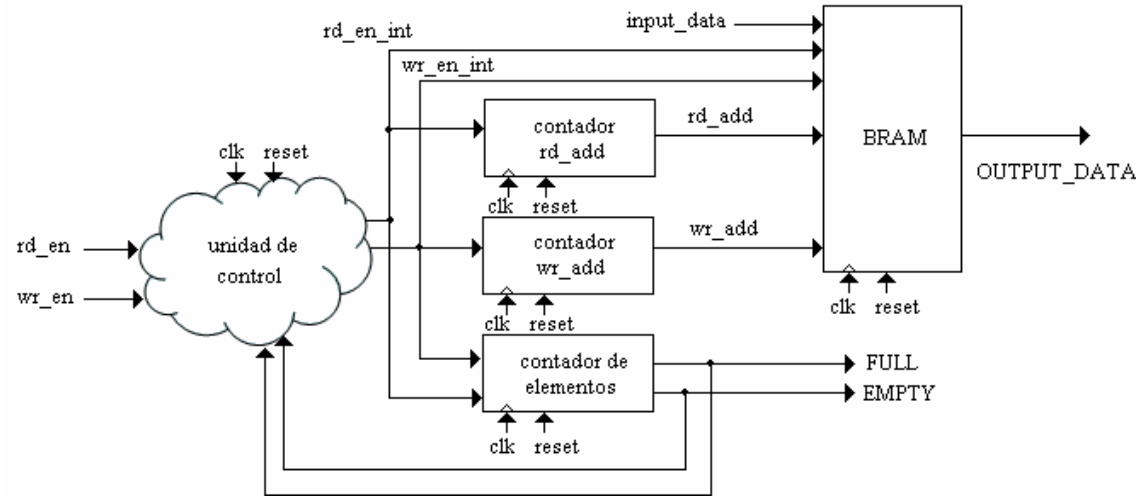


Figura 63: Arquitectura de la fifo en la primera fase de desarrollo

Tanto el contador *rd_add* como *wr_add* son ascendentes y se incrementan en caso de que llegue un lectura, si no está vacía, y una escritura, si no está llena. El contador de elementos se incrementa con las escrituras y se decrementa con las lecturas.

El ancho de palabra de la BRAM tiene un tamaño de 56 bits, el tamaño de la instrucción.

Las salidas son los flags de *full* y *empty*, y el dato leído.

4.2.3.3 Segunda fase de diseño

Partiendo de una fifo corriente se intento adaptar para la gestión de gaps impuesta por la combinación de particiones.

Para evitar que el rendimiento de las particiones decaiga debido a la inserción de gaps, o lo que es lo mismo, no operación, cada vez que se inserta una nueva instrucción se comprueba si su tiempo de ejecución es inferior al del último hueco existente. En caso afirmativo la tarea se inserta en el hueco, adelantando así su ejecución y aprovechando el tiempo que tiene que esperar la partición para su sincronización con las otras colas.

Es necesario añadir un registro que contenga la dirección en que se encuentra el último gap insertado. Para agilizar la comparación de los tiempos de ejecución, y evitar una lectura en la BRAM, también se almacena el tiempo de duración del hueco.

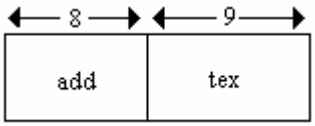


Figura 64: Contenido del registro que almacena el último gap

En el momento en que se inserte una tarea, ahora es necesario comprobar si se trata de una tarea o un gap. El tratamiento es el mismo en ambos con la salvedad de que el gap habrá que registrarlo.

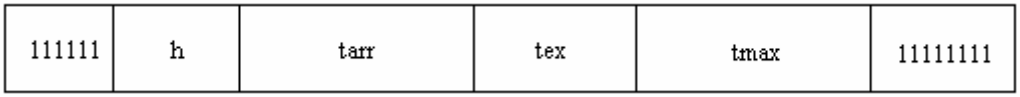


Figura 65: Valores que identifican un gap

Como se aprecia en la imagen anterior la identificación del hueco se puede hacer de dos formas. A nivel de lógica se utilizará el campo w. Su valor es único ya que cualquier tarea que entre al planificador con este valor será rechazada por tener un tamaño mayor que el total del área de ejecución (101100) y por lo tanto nunca llegará a la fifo. El otro campo especial, número de tarea, se utiliza solamente para identificar el gap durante la simulación ya que, por pantalla, sólo se muestra este campo.

La ruta de datos queda de la siguiente manera con las novedades introducidas:

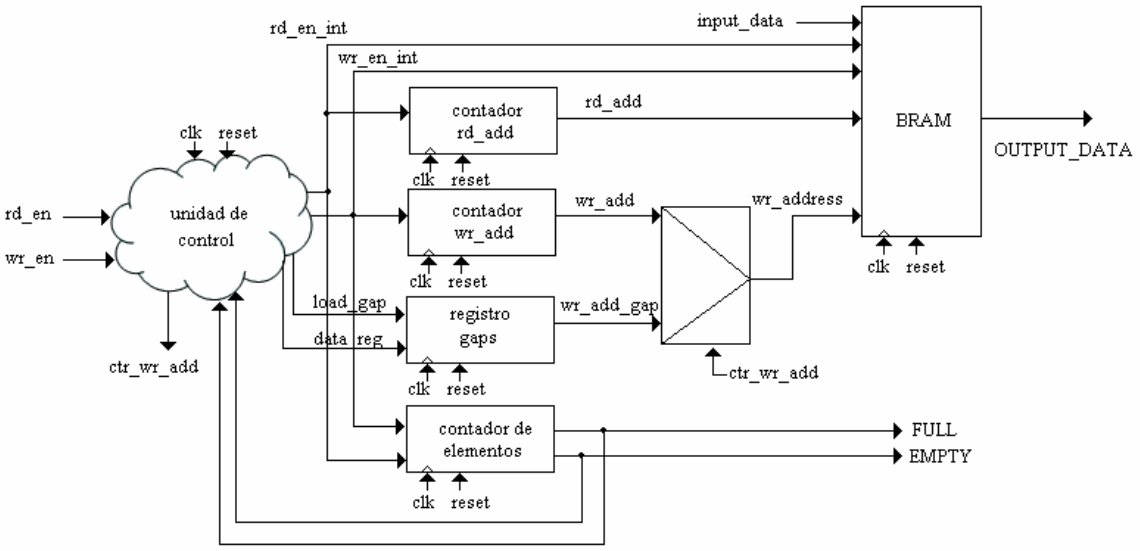


Figura 66: Arquitectura de la fifo en la segunda fase de desarrollo

No es un buen diseño para la gestión de gaps debido a la forma en que se generan la dirección de lectura. Al tratarse de un contador ascendente siempre se leerá siguiente posición de la memoria. Debido a esto la única posibilidad es sustituir el gap por una tarea. Si la tarea entrante tiene igual tex que el hueco el funcionamiento es correcto, sin embargo, en caso de ser menor aparece un problema.

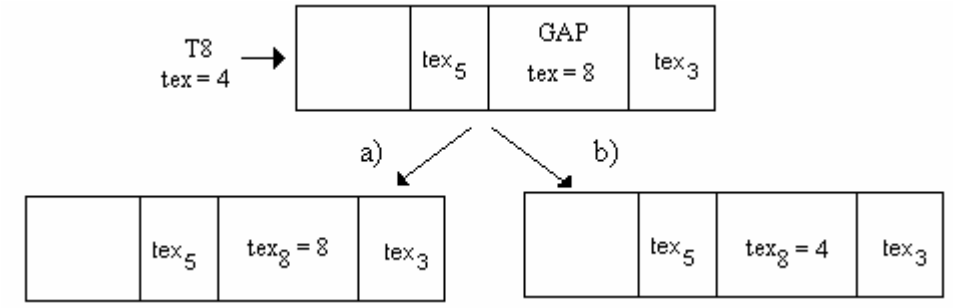


Figura 67: Opciones para tratar la inserción en gap

Habría dos formas de tratar el problema mostradas en la figura 67. La opción a) tiene el inconveniente de que si se modifica el tex de la tarea, dentro de la fifo, se consigue la sincronización pero, al no generarse un nuevo gap (en este caso aparecería otro con tex = 4), se pierde la ocasión de ocupar esos ciclos con una nueva tarea, con la consiguiente caída del rendimiento.

En el caso b) se pierde la sincronización de las colas, por lo que esta opción no es válida.

El comportamiento deseado se lograría con una cola dinámica que permite crear nodos en mitad de la lista (figura 68).

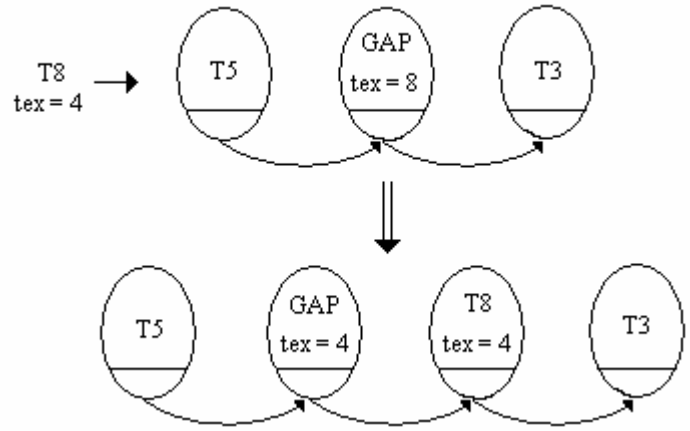


Figura 68: Fifo implementada como una lista dinámica

De este modo se crea un nuevo hueco y la posibilidad de introducir una tarea en él, aprovechando el tiempo que es necesario esperar para lanzar T5 sincronizada.

4.2.3.4 Tercera fase de diseño

Para hacer que la cola se comporte como un lista dinámicamente enlazada se concatena a cada tarea un puntero a la siguiente tarea a ejecutar. Para ello hay que aumentar el tamaño del dato contenido en la BRAM.

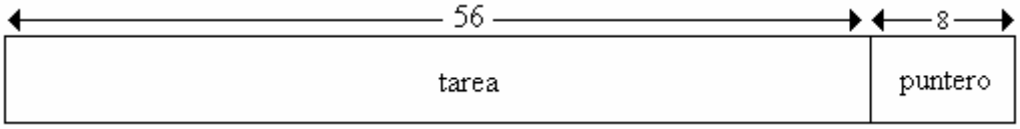


Figura 69: Dato almacenado en la BRAM

Como la siguiente dirección a leer se coge de la memoria habrá que cambiar el contador rd_add por un registro que almacene este valor. Debido a este cambio, ahora la lectura tarda dos ciclos en completarse porque necesita esperar a que el dato leído esté en la salida de la BRAM para saber a que dirección debe acceder a continuación.

En esta etapa se añaden las salidas de confirmación *rd_ack* y *wr_ack*.

Este modelo tampoco es muy bueno porque cada vez que se necesite actualizar el puntero nos vemos obligados a reescribir también la tarea, no se pueden modificar sólo unos bits. Esto nos obliga a leer la tarea para volver a escribirla junto al nuevo puntero.

4.2.3.5 Cuarta fase de diseño

Para poder actualizar el puntero sin necesidad de tocar la tarea se rediseña la parte de almacenamiento dividiéndola en dos BRAMs, una para las tareas y otra para punteros. Tanto para leer como para escribir se accede a la misma dirección en ambas memorias.

Este cambio se hace sobre la fifo simple para comprobar el correcto funcionamiento del sistema de punteros antes de complicarlo con la gestión de gaps.

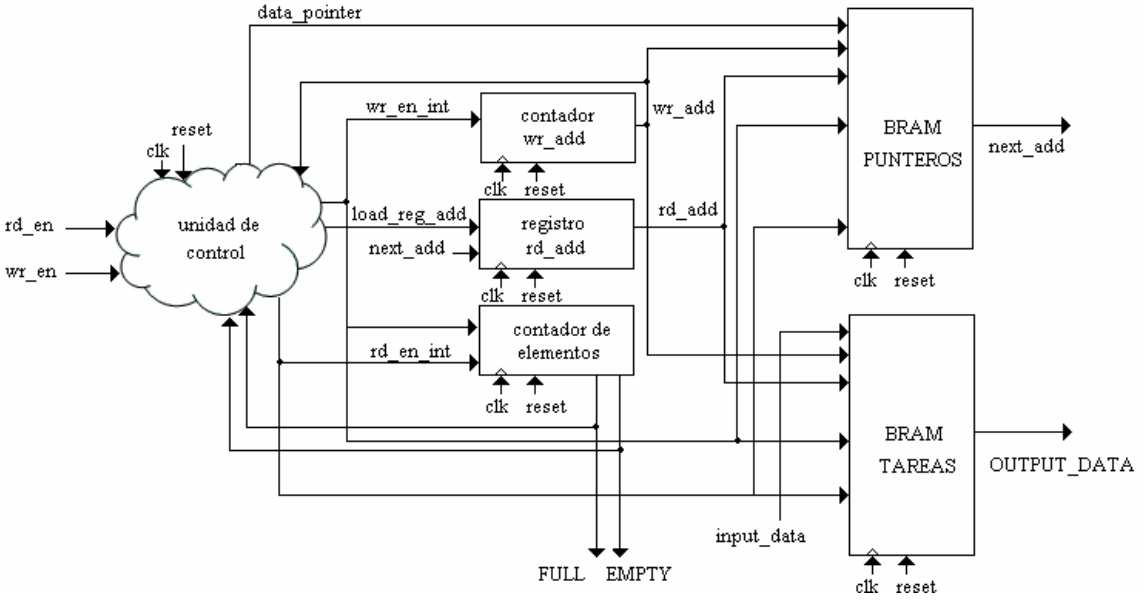


Figura 70: fifo simple implementada como una lista dinámicamente enlazada

4.2.3.6 Quinta fase de diseño

Una vez comprobado el correcto funcionamiento de este sistema se procedió a incorporar toda la lógica necesaria para la gestión de gaps. El diseño obtenido en esta fase es el final.

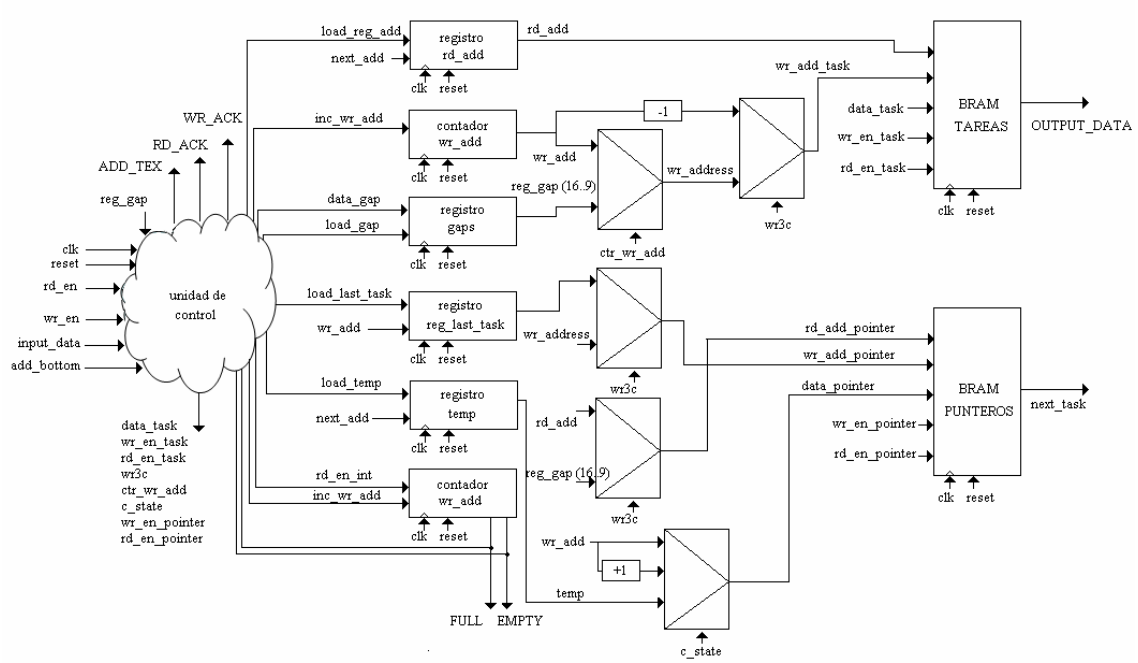


Figura 71: Ruta de datos final de la fifo

Cuando se recibe una instrucción de lectura se pueden dar dos casos. El primero es que se lea una tarea: esta operación no tiene ninguna particularidad, se realiza una lectura normal. El segundo es que se lea el la dirección de memoria apuntada por el registro del último gap: en este caso, además de realizar la lectura, será necesario resetear dicho registro ya que el hueco ha pasado a ejecución.

La operación de escritura es algo más compleja que la lectura y requiere distinguir más casos:

- Se escribe un gap: se realiza una escritura simple al final de la cola. La señal *add_tex* toma el valor '1'. La dirección de la memoria en que se ha escrito se guarda en el registro *reg_gap*.

- Se escribe una tarea: se comprueba en cual de las siguientes situaciones nos encontramos y en función a eso se realiza las operaciones oportunas:

    - No existe ningún gap, el tex de la tarea es mayor que el del gap o la señal *add_bottom* está activa. En este caso se realiza una escritura normal. La señal *add_tex* toma el valor '1'. La dirección de la memoria en que se ha escrito se guarda en el registro *reg_last_task*.

    - El tex de la tarea es igual al del gap existente y *add_bottom* vale '0'. En este caso se escribe en la BRAM de tareas, en la dirección apuntada por el registro de gaps, el dato entrante, sin tocar el puntero.

- El tex de la tarea en inferior al del gap. Es la escritura más compleja de todas y necesita de tres ciclos de reloj para realizarse:
  - Ciclo 1: en la BRAM de punteros se lee la dirección almacenada en el registro de punteros y se carga en un registro temporal. También se accede a la dirección guardada en *reg_last_task* y se escribe la posición que tendrá la próxima escritura.
  - Ciclo 2: se escribe en ambas memorias, posición *reg_gap*, la tarea entrante y la dirección en que se insertará el nuevo gap.
  - Ciclo 3: se escribe en ambas memorias el nuevo gap y la dirección almacenada en el registro temporal, escrito en el primer ciclo.

Esta ruta de datos tiene dos restricciones para evitar problemas provocados por la escritura en un gap:

- La lectura tiene prioridad sobre la escritura. La BRAM tiene la capacidad de escribir y leer en el mismo ciclo, el problema llega si se produce una lectura y una escritura de tres ciclos a la vez. Como se ha explicado más arriba, en el primer ciclo de esta operación de escritura es necesario realizar una lectura en la memoria de punteros. Como no se pueden realizar dos lecturas, lo que se ha resuelto es retrasar un ciclo el comienzo de la escritura y servir la lectura en primer lugar para evitar problemas en la sincronización de las colas.
- Si el puntero de lectura y el de último gap apuntan a la misma dirección de memoria el hueco no será tenido en cuenta a la hora de realizar una escritura. Esto se debe a que sería posible leer el gap mientras éste está siendo sustituido o actualizado y leer un valor erróneo.

Todas estas restricciones y características referidas quedan reflejadas en la máquina de estados del controlador:

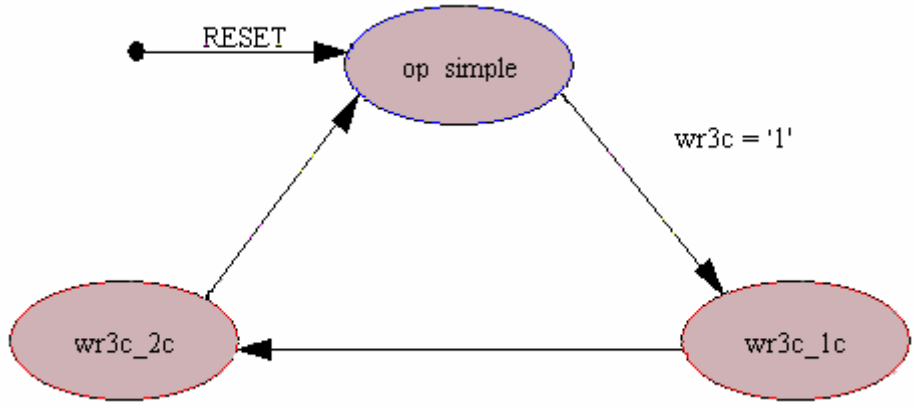


Figura 72: Máquina de estados del controlador de la fifo

4.2.3.7 Simulaciones

Para ver mejor como funciona, a continuación se muestran una serie de simulaciones.

Es esta primera simulación se realizan los siguientes pasos:

- Escritura de un gap.
- Escritura de una tarea con *add_bottom* activa.
- Escritura de una tarea con el mismo tiempo de ejecución que el gap.

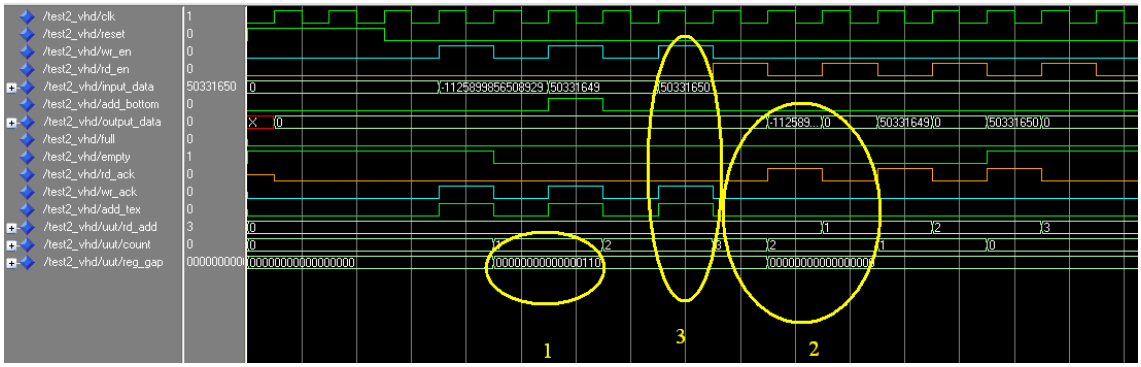


Figura 73: Simulación 1 de la fifo

En la simulación figura 73 se pueden ver varios detalles. En primer lugar, el punto marcado con un 1 muestra como al escribir un gap el *reg_gap* guarda la dirección en que se ha introducido, la cero. En el segundo punto se ve como al leerlo el registro se resetea.

Otro detalle, que se muestra en el punto 3, es que al insertar una tarea con el mismo tiempo de ejecución que el gap, este no lo sobrescribe por cumplirse que la dirección a leer y la del gap es la misma.

En la segunda simulación se realizan las siguientes operaciones:

- Escritura de una tarea
- Escritura de un gap
- Escritura de una tarea con *add_bottom* activa.
- Lectura / escritura de una tarea con menor tiempo de ejecución que el gap.
- Escritura de una tarea con menor tex que el gap resultante de la anterior escritura.
- Lectura durante la escritura anterior.
- Escritura de una tarea con mayor tex que el gap.
- Escritura con igual tiempo de ejecución que el gap

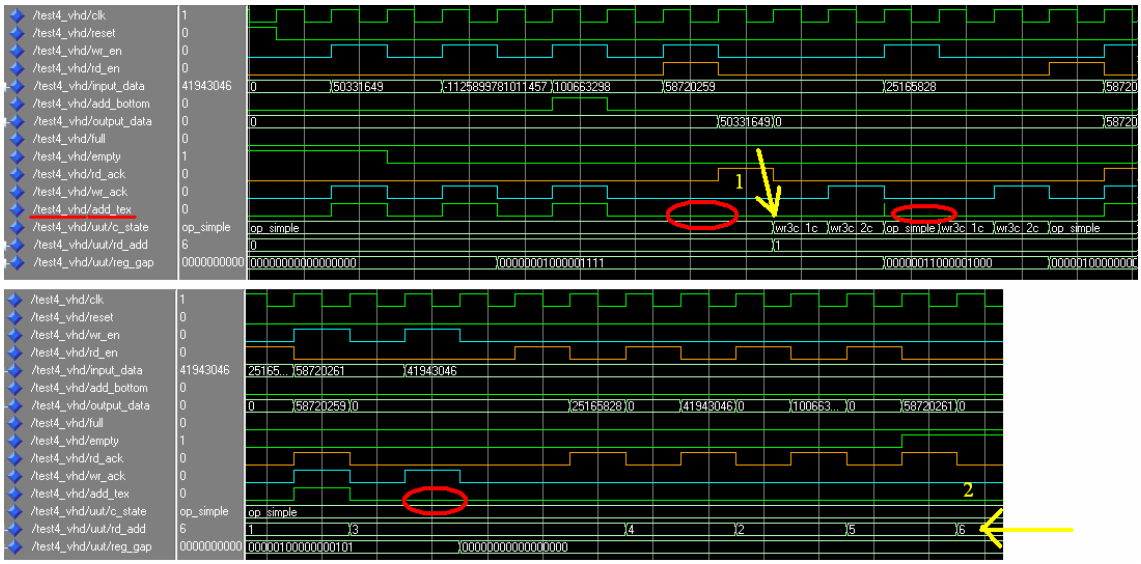


Figura 74: Simulación 2 de la fifo

Como se ha explicado anteriormente, si llega una lectura y una escritura de tres ciclos, la escritura se retrasa un ciclo debido a que la lectura tiene prioridad. Esto está indicado por la flecha 1 en la figura 74.

La flecha 2 llama la atención sobre el contenido de *rd_add* que indica la dirección de la que se lee. Como se aprecia en la imagen, las posiciones no son correlativas debido a la inserción de tareas en gaps.

Marcado en rojo se comprueba que en todas aquellas escrituras en gap la señal add_tex no se activa. Conviene recordar que esta señal indicaba al planificador que debe sumar al tiempo de espera de la cola el tex de la tarea. Como las tareas se han metido en un gap, el tiempo de espera no se incrementará ya que lo que hacen es aprovechar un tiempo perdido en sincronización.

Otro detalle que se puede apreciar en la imagen es como se actualiza el registro de gaps cada vez que se inserta una tarea en él. Las modificaciones corresponden tanto al nuevo tiempo de ejecución como a la dirección de memoria en que se almacena.

4.3 Simulación

Para comprobar el funcionamiento del planificador se ha creado una entidad superior llamada planificador_top. Los componentes se muestran a continuación:

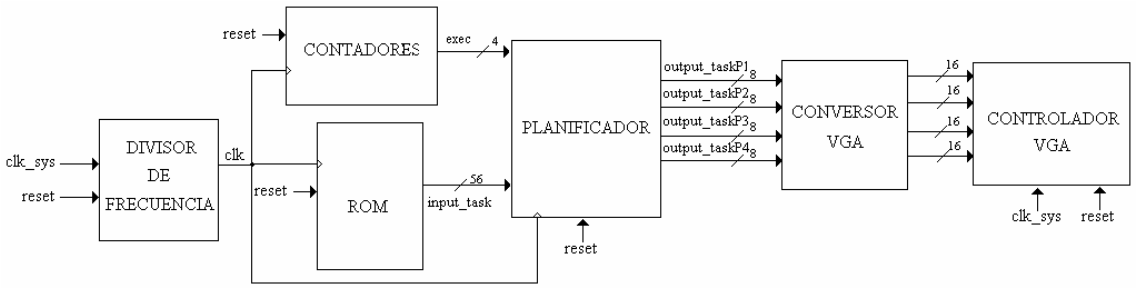


Figura 75: Diseño del banco de pruebas

- Divisor de frecuencia: el divisor se ha introducido para poder ver el resultado por un monitor.

- ROM: contiene 32 tareas que se irán lanzando en sus correspondientes ciclos de llegada. Los datos con los que se inicializa la rom se encuentran en el fichero *rom_raw_data.dat*.

- Contadores: este módulo consiste en cuatro contadores que simulan la ejecución de las tareas en FPGA. Cuando se lanza una tarea se carga el contador asociado a esa partición con el tiempo de ejecución correspondiente y se va decrementando en uno cada slot que pasa. Fija el valor *exec* que es el que genera las peticiones de lectura.

- Conversor VGA: recibe el identificador de las tareas en ejecución y los convierte en una cadena de 16 bits para ser mostrado por pantalla.

La depuración ha sido complicada. Tras comprobar el correcto funcionamiento a través de una simulación de comportamiento con Modelsim se procedió a generar el fichero .bit y a cargarlo en la FPGA. Lo que se veía través del vga no se correspondía con la planificación prevista.

Ha sido necesario realizar las simulaciones post-translate, post-map y post-route para depurar el sistema. Fue difícil el rastreo de las señales debido a que el ISE las renombra en el proceso de traducción.

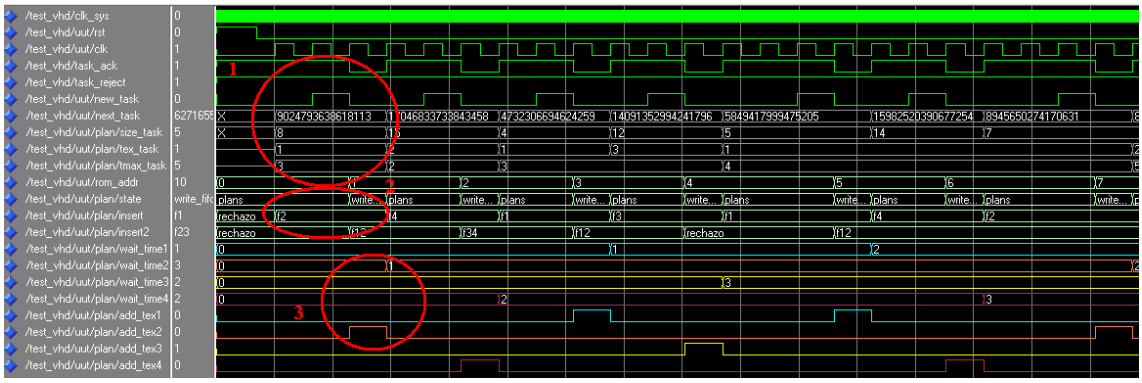
El lote de tareas que se le pasa al planificador, con los campos w – tarr – tex – tmax – nº tarea (por orden de llegada), son:

8 – 2 – 24 – 68 – 1	14 – 41 – 40 - 180 - 12	13 – 87 – 17 - 387 - 23
15 – 5 – 42 – 58 – 2	10 – 44 – 17 - 174 - 13	14 – 91 – 19 - 587 - 24
4 – 8 – 28 – 84 – 3	12 – 47 – 79 - 240 - 14	6 – 95 – 27 - 476 - 25
12 – 11 – 77 – 93 – 4	6 – 51 – 29 - 229 - 15	11 – 100 – 40 - 321 - 26
5 – 15 – 16 - 98 - 5	12 – 54 – 56 - 239 - 16	15 – 106 – 12 - 259 - 27
14 – 18 – 20 - 115 - 6	16 – 58 – 24 - 278 - 17	10 – 111 – 19 - 493 - 28
7 – 22 – 23 - 119 - 7	5 – 61 – 29 - 281 - 18	7 – 115 – 33 - 526 - 29
7 – 25 -32 - 130 - 8	11 – 66 – 16 - 304 - 19	12 – 120 – 21 - 528 - 30
53 – 29 – 16 - 143 - 9	7 – 70 – 28 - 283 - 20	42 – 126 – 14 - 633 - 31
20 – 32 -26 -141 - 10	9 – 74 – 41 - 315 - 21	18 – 134 – 40 - 584 - 32
5 – 38 – 24 - 136 - 11	24 – 80 – 32 - 400 - 22	

Viendo la muestra a ejecutar ya se observa que la tarea número 9 será rechazada por su tamaño, que es mayor que el área de ejecución.

Para realizar la simulación se ha modificado el divisor de frecuencia para poder ver los resultados en un breve espacio de simulación.

Vamos a analizar en primer lugar el proceso de planificación y las escrituras en las fifo.



Gris: datos de la tarea en el bus de entrada

Azul: señales asociadas a la partición 1

Naranja: señales asociadas a la partición 2

Amarillo: señales asociadas a la partición 3

Fucsia: señales asociadas a la partición 4

Figura 76: Planificación de las tareas 1 a la 7

En la imagen superior se comprueba que todas las tareas son aceptadas. Vamos a fijarnos con más detalle en la primera, el resto tienen el mismo comportamiento.

Punto 1. El círculo señala los valores de la tarea disponible en el bus de entrada cuando *new_task* se activa. También se aprecia que la señal *task_ack* se pone a cero, es activa a baja, lo que quiere decir que la tarea ha sido planificada con éxito. La unidad de los valores montrados por *tex* y *tmax* son slots, no ciclos de reloj.

Punto 2. Aquí se aprecia que *insert* ha tomado el valor f2, lo que quiere decir que la tarea será insertada en la cola asociada a la partición 2 durante el estado write_fifo. Basándonos en que el tamaño de la tarea es 8 y que la primera partición tiene tamaño 6 y la segunda 10, se puede afirmar que la planificación es correcta, teniendo cuenta también que los tiempos de espera son cero y no influyen.

Punto 3. Aquí se muestra como la tarea ha sido insertada al final de la cola y por lo tanto su tiempo de espera se incrementa con el valor *tex*.

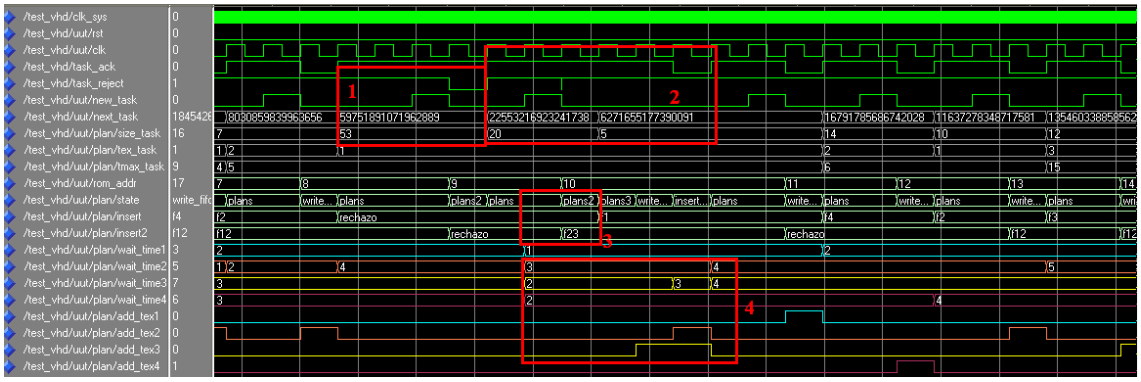


Figura 77: Planificación de las tareas 8 a la 14

En esta fase de la simulación aparecen dos casos nuevos. Se explican en detalle los casos de las tareas 9 y 10.

Punto 1. Si nos fijamos en el valor de *size_task* de la tarea 9 se observa que es 53. Como ya se adelantó, esta tarea es rechazada por tener un tamaño superior al del área total de ejecución, que es de 44.

Punto 2. Atendiendo también al tamaño en la tarea 10 se ve que no cabe en ninguna de las particiones, y que de poder ejecutarse será mediante una combinación de particiones.

Punto 3. La señal *insert* tiene el valor rechazo, lo que conlleva la transición al estado *plans2*. Aquí se ve que la señal *insert2* toma el valor f23, que significa que la tarea se ejecutará en las particiones 2 y 3.

Punto 4. Como se va a realizar una combinación de las particiones 2 y 3 habrá que sincronizar ambas colas. Aquí se aprecia que la cola 2 tiene un tiempo de espera de 3 slots mientras que el de la cola 3 es de sólo 2. Debido a esto se inserta un gap en ésta última y a continuación la tarea en ambas colas. Esto se aprecia en las modificaciones que sufren los tiempos de espera.

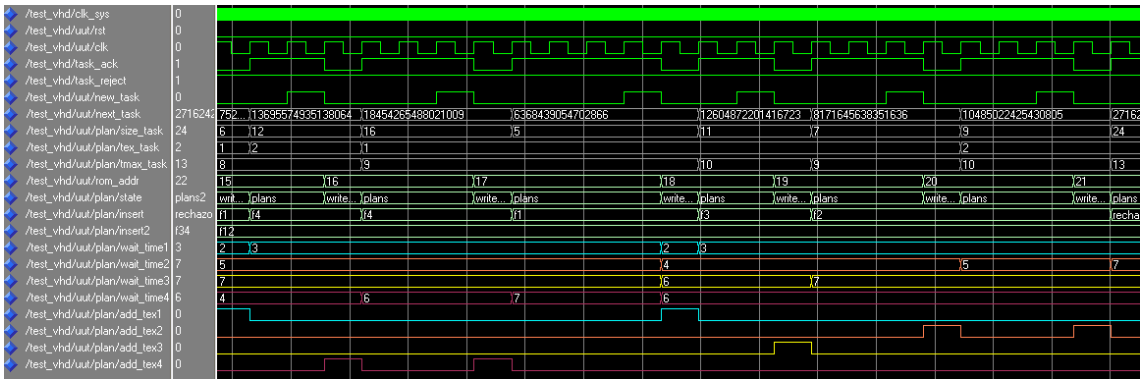


Figura 78: Planificación de las tareas 15 a la 21

En esta imagen se aprecia que todas las tareas desde la 15 a la número 21 son aceptadas.

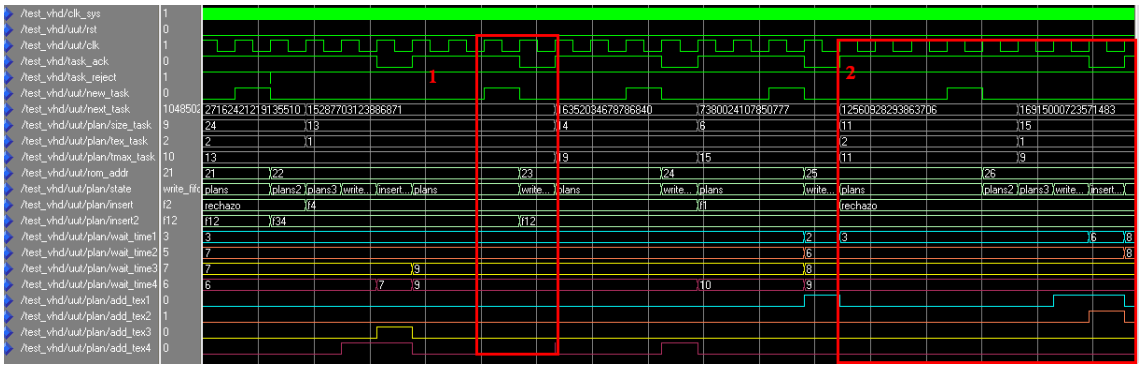


Figura 79: Planificación de las tareas 22 a la 26

De nuevo se aceptan todas las tareas desde la número 22 a la 26. Nos encontramos de nuevo con dos casos de combinación de particiones, en la primera tarea mostrada y la última. El caso de número 22 es como el explicado anteriormente.

Punto 1. Al insertar la tarea 23 en la partición 4 se observa que la señal *add_tex4* no se activa y, por lo tanto, no se incrementa el tiempo de espera. Esto significa que se ha insertado en un hueco existente.

Punto2. Planificación de la tarea número 26. Su tamaño es 11 y por lo tanto debería ir a la tercera partición o en su defecto a la cuarta. Esto no se produce por las restricciones temporales que no se cumplen. Para comprobarlo hay que ver que la suma de su tex, el tiempo de espera y el slot actual es menor o igual que el tmax. Debido a que no se cumple la condición la tarea en mandada a las particiones primera y segunda.

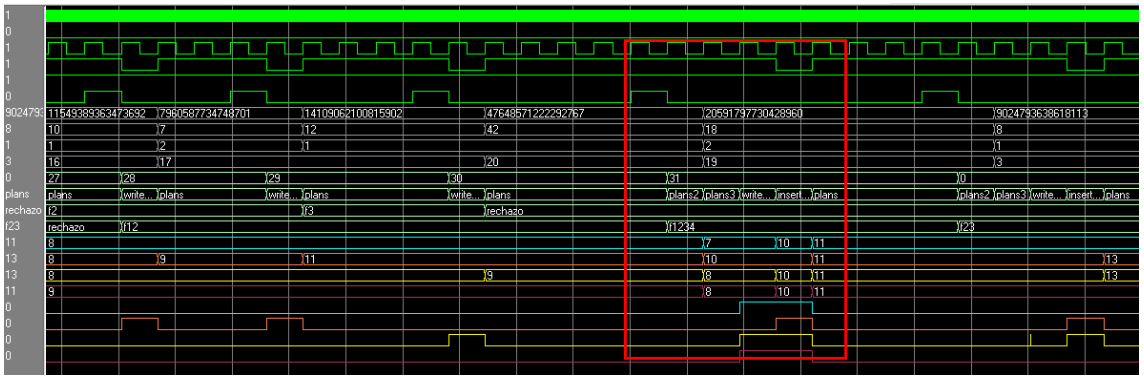


Figura 80: Planificación de las tareas 27 a la 32

Todas las tareas son aceptadas. En la zona enmarcada se aprecia la planificación de la tarea número 31 con un tamaño de 42. Para poder ejecutarla es necesaria la utilización del área de ejecución completa, lo que conlleva la combinación de todas las particiones.

Una vez vista la planificación realizada para cada una de las tareas vamos a ver su ejecución.

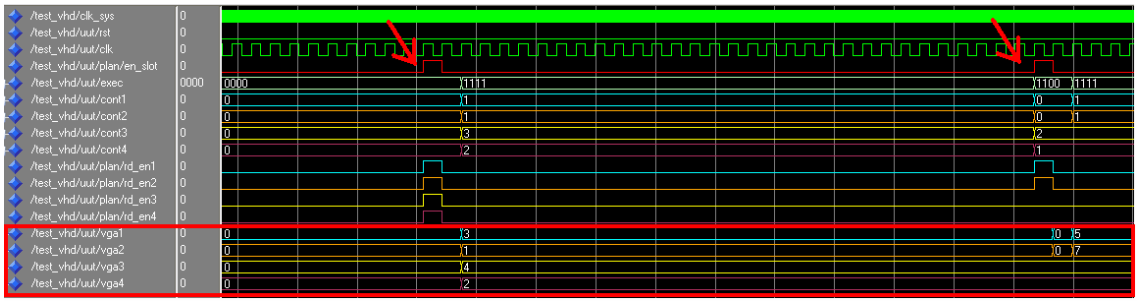


Figura 81: Ejecución de tareas en los dos primeros slots

Las instrucciones se lanzan al comienzo de un slot (indicado por las flechas). En ese momento se comprueba el valor de exec y se hace una lectura en todas aquellas colas para las que su bit correspondiente es 0. En el mismo instante de la lectura se cargan los contadores con el número de slots a ejecutar y se inicia la cuenta atrás.

En el recuadro de la parte inferior se ve la tarea alojada en cada partición (ejecutándose). El cero que aparece delante de las tareas 5 y 7 indican que la FPGA no está ejecutando, sino obteniendo la siguiente instrucción.

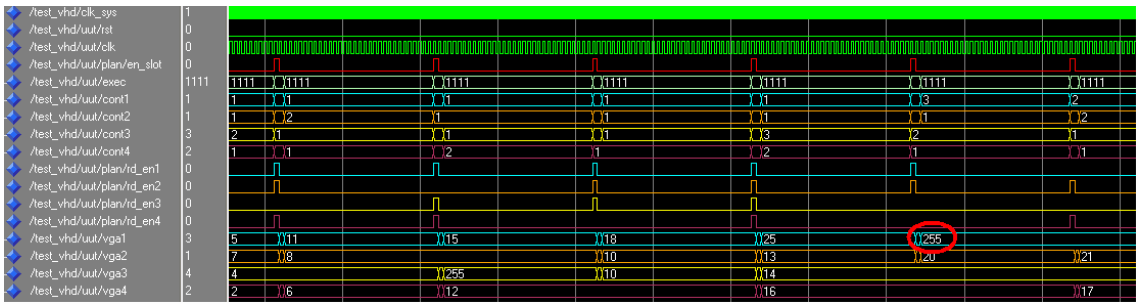


Figura 82: Ejecución de tareas desde el slot tercero al octavo

Aquí se aprecia que en la partición 1 se ha leído un gap. En cuanto la partición adyacente finalice sus ejecuciones previas se producirá lo combinación de ambas.

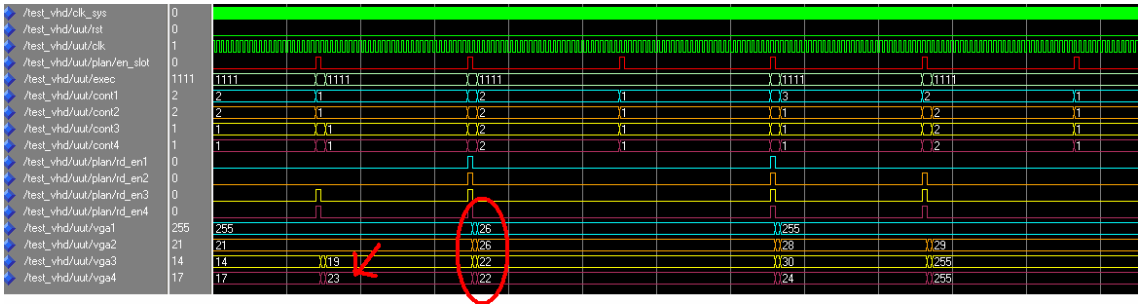


Figura 83: Ejecución de tareas desde el slot noveno al decimocuarto

Se aprecia que en las particiones 1 y 2, por una lado, y la y 4, por otro se ha producido la combinación anunciada.

También se observa que la tarea 23 se ha ejecutado antes que la 22. Esto es debido a que dicha tarea se está ejecutando en lugar del gap necesario para sincronización de la combinación anteriormente comentada.

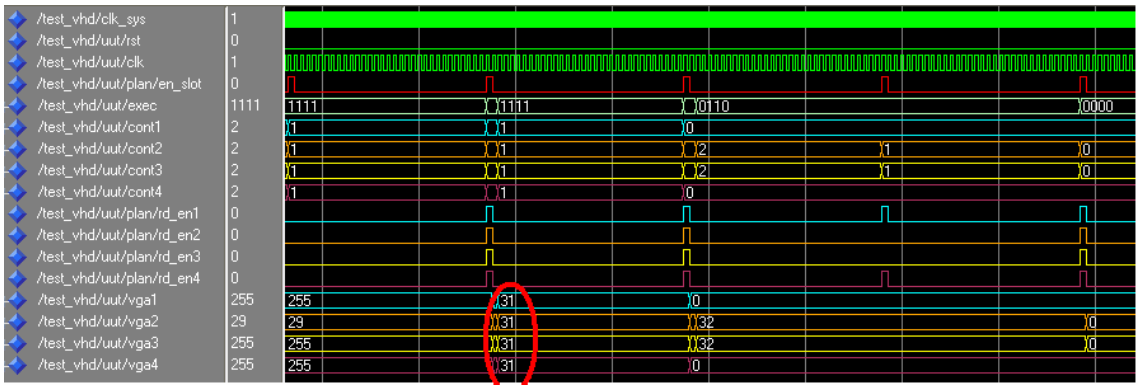


Figura 84: Ejecución de tareas desde el slot decimoquinto al decimoctavo

Aquí se ejecuta la tarea 31 de la que había comentado que necesitaba el área total. A continuación en las particiones 2 y 3 se lanza la última tarea planificada. En el resto el valor es 0, lo que quiere decir que ya no tienen más tareas esperando a ser ejecutadas. En los slots sucesivos se generan lecturas, pero sin resultado por haber finalizado la ejecución del lote de tareas enviado.

CAPÍTULO 5: DESARROLLO DEL PROYECTO

Desde que se comenzó el proyecto hasta su finalización se ha pasado por diversas etapas. Unas veces por encontrarme en un punto muerto y otras por haber finalizado con éxito el punto realizado.

Como ya se ha explicado con detalle los pasos realizados y la evolución sufrida por cada uno de los objetivos del proyecto en sus correspondientes capítulos, sólo comentaré la evolución general.

El primer objetivo en el que me puse a trabajar fue el controlador de memoria DDR SDRAM.

El trabajo inicial fue realizar un proceso de ingeniería inversa con el código del controlador generado con el MIG. Esto no fue una tarea sencilla ya que, debido a que la herramienta se encuentra en una fase relativamente temprana de su desarrollo, genera un código sucio, muy complejo y deja sin implementar funciones básicas de la memoria como el refresco de la misma (es un dispositivo volátil).

Una vez sacados los rasgos generales de la arquitectura, y tras comprobar su no funcionamiento, se procedió a la depuración del código. Esta tarea me tuvo varios meses ocupada debido a la alta complejidad del controlador (tiene un número elevado de componentes), el desconocimiento del funcionamiento de este tipo de memorias y a que al no ser un código propio me costó mucho comprender la función de las diferentes señales.

Tras hacer varias modificaciones y ver que el controlador seguía sin funcionar se procedió a su depuración con un analizador lógico. Tras aprender su funcionamiento y coger la soltura necesaria para muestrear las señales necesarias, se comprobó que el controlador funcionaba correctamente. La conclusión a la se llegó fue que el controlador VGA que se utilizaba para ver por una pantalla el contenido de la memoria no funcionaba.

Se procedió a la búsqueda de un controlador en Internet. Tras encontrar varios se optó por uno que se podía adaptar a nuestras necesidades y se amplió con todas las funcionalidades necesarias.

Una vez se incorporó el nuevo VGA al proyecto y se comprobó el correcto funcionamiento de éste módulo y de la memoria fue el momento de eliminar el banco de pruebas que tenía incorporado el controlador DDR. Para realizar el nuevo controlador se tuvo que volver a estudiar el código para ver que señales de control y datos estaban generadas por el banco de pruebas.

Una vez visto el código no necesario había dos posibilidades: dejar la máquina de estado que genera el código de operación dentro del controlador y meter las peticiones de lectura y escritura, o por el contrario dejarla fuera y pasarle al controlador el código de operación como una nueva entrada. Se optó por la primera opción debido a que esta máquina ya estaba implementada dentro, como parte del banco de pruebas, y suponía menos retoques en el código.

Tras varias pruebas y largo proceso de depuración se obtuvo un controlador que funciona para tamaños de ráfaga cuatro.

Después de conseguir que funcionara el controlador de memoria se empezó con el diseño del planificador.

Para el planificador se comenzó con varios diseños funcionales basados en el artículo dado como referencia en su capítulo.

El primer componente a desarrollar fue el sistema de almacenamiento del planificador. Inicialmente se implementó como una cola fifo simple pero se fue modificando para adaptarla a las funcionalidades impuestas por la especificación.

A continuación se diseñó el módulo de planificación. Este componente sufrió varios procesos de rediseño debido a que la sincronización de las colas daba problemas por los retardos en las operaciones de lectura y escritura en las mismas.

Se realizaron simulaciones y el planificador funcionaba bien pero al cargarlo en la FPGA el funcionamiento de alguna de las colas no era el esperado. Se realizaron simulaciones temporales y se vieron problemas en la inicialización de algunas señales que no aparecían en las simulaciones de comportamiento. Tras depurar estos errores se obtuvo el diseño final.

Excepto para el controlador de memoria, en el que comencé a partir de un código dado, el diseño de todos los módulos y sus componentes se han realizado sobre papel antes de comenzar a escribir código. Esto ha simplificado mucho las fases de rediseño y las tareas de depuración.

CAPÍTULO 6: CONCLUSIONES

Aunque ya había trabajado con FPGAs en alguna asignatura, nunca había sido a este nivel de complejidad, tanto en lo referente a lo sistemas diseñados como en los dispositivos utilizados. Me ha servido para conocer más el mundo del diseño hardware y aprender sobre todo técnicas de depuración.

He aprendido a trabajar con las herramientas y librerías que proporciona Xilinx y he adquirido mayor soltura con el código VHDL sintetizable.

Los problemas encontrados que han sido numerosos se han producido tanto a nivel técnico como organizativos.

En el desarrollo del planificador he aprendido a especificar módulos comunes como una BRAM o una ROM.

Aunque se ha cumplido con los requisitos especificados, el planificador admite muchas ampliaciones que le proporcionen más funcionalidades. Por ejemplo, se ha dejado preparado para adaptarlo a una división del área de ejecución en 2D o incorporar un módulo que ajuste en tiempo de ejecución el tamaño de las particiones.

Por otro lado, el controlador DDR RAM también admite mejoras como hacerlo funcionar para más tamaños de ráfaga.

APÉNDICES

A. Controlador vga

Este módulo ha sido utilizado para la depuración tanto del controlador DDR SDRAM como del planificador.

El módulo controlador del VGA ha sido escrito en Verilog ya que en la red se encontraba parte del código [16] y funcionaba correctamente. El código encontrado no realizaba exactamente las funciones necesarias para la depuración, por ello se decidió ampliar dicho código con funcionalidades propias del proyecto. Las modificaciones y ampliaciones llevadas a cabo son las enumeradas a continuación:

- Ajuste del reloj para poder usarlo sin problemas con una frecuencia de 100 MHz ya que sólo funcionaba correctamente a una frecuencia de 25 MHz.
- Modificación de las condiciones del trigger que actualiza la pantalla a las requeridas por el planificador y el controlador DDR.
- Modificación de la información que se muestra en pantalla aumentando líneas de datos de entrada para su visualización.
- Inclusión de señales de control de entrada para poder modificar el texto y la información a mostrar por pantalla (lectura/escritura en la memoria DDR, mostrar trabajos en cada partición…)
- Inclusión de un envoltorio VHDL para poder usarlo con el resto del proyecto.

A.1 Descripción

La idea de un controlador de pantalla es muy sencilla. Simplemente tenemos que recorrer cada uno de los píxeles de la pantalla y asignarle un determinado valor RGB. Para ello utilizamos un contador que vaya incrementando la columna y una vez acabada ésta otro contador incrementa la línea. El valor del límite de fila y de columna es lo que determina la resolución de pantalla y la velocidad con la que va el contador es la que determina el refresco de pantalla que obtendremos. Sin embargo este controlador está enfocado a representar caracteres en pantalla, por ello lo que hacemos es ir avanzando con un contador de carácter aunque más internamente tengamos que recorrer cada bloque correspondiente al carácter a nivel pixel.

La estructura interna del controlador es la siguiente. Cada carácter tiene una dimensión de 8 x 8 píxeles. Tenemos una ROM en la que se encuentra el estado en el que tiene que estar cada pixel de la matriz 8 x 8 para formar cada uno de los caracteres que queramos representar en pantalla. Para almacenar los caracteres que hay en cada momento en pantalla utilizamos una memoria RAM. Por lo tanto, lo que almacenamos son caracteres, no los pixeles que los forman. Éstos ya los tenemos de forma permanente en la ROM que se encuentra en el fichero CHAR_GEN_ROM.

Dependiendo de la resolución que elijamos en el fichero SVGA_DEFINES.v podremos mostrar la siguiente cantidad de caracteres y líneas:

- En modo 640 x 480 se muestran 80 caracteres en 60 líneas.

- En modo 800 x 600 se muestran 100 caracteres en 75 líneas.
- En modo 1024 x 768 se muestran 128 caracteres en 96 líneas.

Además de la resolución de pantalla, en este fichero también seleccionamos la frecuencia de refresco de pantalla ya que esto dictaminará la velocidad con la que tendremos que avanzar los contadores que avanzan líneas y columnas.

En el fichero SVGA_DEFINES.v ya vienen calculadas las constantes necesarias para el funcionamiento del controlador con las siguientes resoluciones y frecuencias de refresco:

Resolución	Frecuencia	Frecuencia de reloj a nivel pixel
640 x 480	60 Hz.	25.175 MHz
640 x 480	72 Hz	31.000 MHz
640 x 480	75 Hz	31.500 MHz
640 x 480	85 Hz	36.000 MHz
800 x 600	56 Hz	38.100 MHz
800 x 600	60 Hz	40.000 MHz
800 x 600	72 Hz	49.500 MHz
800 x 600	75 H	50.000 MHz
800 x 600	85 Hz	56.250 MHz
1024 x 768	60 Hz	65.000 MHz
1024 x 768	70 Hz	75.000 MHz
1024 x 768	75 Hz	78.750 MHz
1024 x 768	85 Hz	45.500 MHz

Tabla 17: Resoluciones y frecuencias del controlador vga

En el fichero donde indicamos en código ASCII hexadecimal los caracteres que queramos mostrar es CHAR_DISPLAY.v. Dada la estructura que necesitamos le pasamos 4 parámetros desde fuera para poder mostrar cada una de las tareas correspondiente a cada partición. Se puede elegir el color de fondo de cada uno de los bloques 8 x 8 píxeles así como el color del carácter en sí.

A.1.1 Esquema

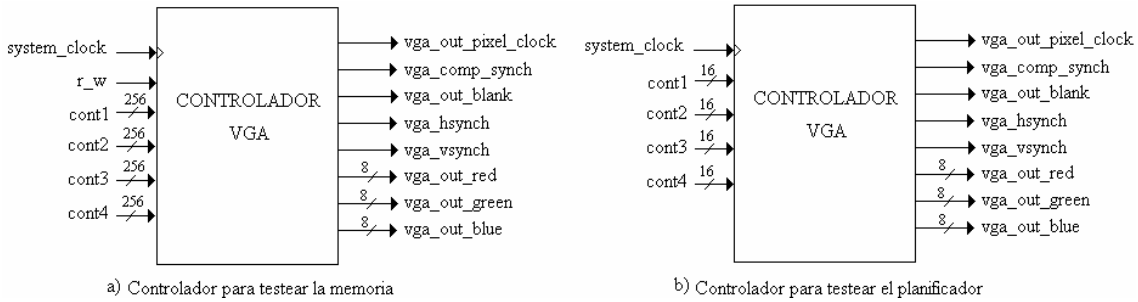


Figura 85: Esquema del controlador vga

Este controlador se ha utilizado para depurar el controlador DDR SDRAM y el planificador. Para cada una de las depuraciones se necesitaba mostrar diferentes elementos por lo que los datos que se envían al controlador vga son diferentes. En la figura 85 se muestran los datos que se envían en el controlador utilizado con la memoria (a) y el empleado en el planificador (b).

El significado de las señales se muestra en las siguientes tablas:

Entrada	Descripción
system_clock	- Señal de reloj del sistema
r_w	- Esta señal solo está en el controlador utilizado para la depuración del controlador de la memoria. - Indica si se está haciendo una lectura o una escritura. En función de esta señal se mostrarán unos datos u otros.
cont1 cont2 cont3 cont4	- Datos a mostrar por pantalla

Tabla 18: Entradas del controlador vga

Salida	Descripción
vga_out_pixel_clock	- Reloj de píxel
vga_comp_synch	- Sincronización compuesta para el vga
vga_out_blank	- Señal que anula las entradas de RGB para no mostrar en el display
vga_hsynch	- Sincronización horizontal para el conector de salida del VGA
vga_vsynch	- Sincronización vertical para el conector de salida del VGA
vga_out_red	- Señal de rojo
vga_out_green	- Señal de verde
vga_out_blue	- Señal de azul

Tabla 19: Salidas del controlador vga

Para ver mejor la forma en que se muestran los datos, se incluyen unas imágenes de lo que se vería por pantalla:

Figura 86: Imagen mostrada por el vga para la memoria

La primera línea (figura 86) muestra la operación que se está realizando que puede ser reading o writing. A continuación se muestran el dato a escribir o leído y la dirección de la memoria a la que se está accediendo.

Figura 87: Imagen mostrada por el vga para el planificador

En la figura 87, caada franja de color se corresponde con una partición del área de ejecución de la FPGA. Se muestra el número de la partición y el identificador de la tarea que se está ejecutando en cada una de ellas..

B. Bibliografía

[1] "Microelectrónica: Evolución del diseño", Antonio Adrián Quijano

http://www.ciencias.org.ar/%5Cuser%5Cfiles%5CAnales05_Quijano.pdf

[2] "Desarrollo de un entorno hardware para la ejecución de aplicaciones de propósito general sobre FPGAs", L. Sánchez, R. Sánchez, J. Valero. Trabajo fin de carrera Facultad de Informática UCM 2005/2006

[3] "FPGA: nociones básicas", M.L. López y J.L.Ayala

http://www-lsi.die.upm.es/~marisa/docencia/fpga_a2_2004.pdf

[4] The Virtex-II Pro / Virtex-II Pro X FPGA Family user guide

http://www.xilinx.com

[5] "Tema 1. Diseño automático", J.M. Mendías

http://www.dacya.ucm.es/mendias/143/143.html

[6] XUP2P user guide

http://www.xilinx.com

[7] Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs

http://www.xilinx.com

[8] "Auto-Reconfiguración sobre FPGAs", J. Castillo, I. González, P. Huerta y J.I. Martínez. http://www.escet.urjc.es/~phuerta/pdf/jcra_2005.pdf

[9] 16700 Series Logic Analysis System

http://www.home.agilent.com/agilent/product.jspx?nid=-536902443.0.00&c=153491.i.2&to=79837.g.1&cc=ES&lc=spa&pageMode=LB&no=0

[10] http://www.valueram.com/datasheets

[11] DDR SDRAM Data sheet

http://www.micron.com

[12] ChipScope Pro Serial I/O Toolkit User Guide

http://www.xilinx.com

[13] Memory interface generator user guide

http://www.xilinx.com

[14] Synthesizable 400 Mb/s DDR SDRAM Controller
http://www.eetasia.com/ARTICLES/2004DEC/A/2004DEC09_MPR_AN.PDF?SOURCES=DOWNLOAD

[15] Artículo: "Constant complexity scheduling algorithm for HW multitasking in 2D Reconfigurable FPGAs", S. Román, H. Mecha, D. Mozos, J. Septién, Lecture Notes in Computer Science Vol.: 3985 Pg.:187-192 Ed. Springer 2006

[16] http://embedded.olin.edu/xilinx_docs/projects/realtimewatch-v2p.php

C. Glosario

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

Almacenamiento volátil: dispositivo que pierde su contenido cuando se suspende la alimentación.

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".

BRAM (Block RAM): bloques de memoria integrados en las FPGAs que funciona a la misma frecuencia que el bus de datos y es capaz de realizar un acceso por ciclo. Su funcionamiento es equivalente a una cache de nivel 1.

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 conjunto 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.

Gap: es una no operación que se inserta en las colas del planificador para sincronizarlas.

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.

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 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 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.

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 desarrollo o 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.

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.

Slot: unidad de tiempo de ejecución de las tareas lanzadas por el planificador-ubicador.

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.

D. Índice de figuras

Figura	Página
Figura 1: Esquema general del sistema	7
Figura 2: Estructura interna de una FPGA	9
Figura 3: Estructura de un IOB (Xilinx Virtex2P)	10
Figura 4: Estructura de un CLB (Xilinx Virtex2P)	10
Figura 5: Estructura de un LUT	11
Figura 6: Estructura de la interconexión de una FPGA	11
Figura 7: foto XUP	12
Figura 8: Conexiones Virtex II Pro con la XUP	13
Figura 9: Estructura interna de Virtex II Pro	14
Figura 10: Procesadores empotrados en Virtex II Pro	14
Figura 11: Relojes generados por el DCM	15
Figura 12: Foto del analizador lógico 16702B	16
Figura 13: Sistema de ventanas del analizador 16702B	16
Figura 14: Sistema multiventana del analizador 16702B	17
Figura 15: Análisis de onda del analizador 16702B	17
Figura 16: Esquema general de un módulo DDR SDRAM	20
Figura 17: Registro de modo	22
Figura 18: Registro de modo extendido	23
Figura 19: Ráfaga de lectura	24
Figura 20: Ráfaga de escritura	25
Figura 21: Diagrama inicialización de la DDR	26
Figura 22: Flujo de diseño de circuitos digitales	27
Figura 23: Interfaz Project Navigator	28
Figura 24: Ventana de creación de un proyecto	29
Figura 25: Cuadro de procesos	30
Figura 26: Ventana principal de Modelsim	32

Figura 27: Chipscope Pro Core Inserter (Paso 1)	33
Figura 28: Chipscope Pro Core Inserter (Paso 2)	34
Figura 29: Chipscope Pro Core Inserter (Paso 3)	34
Figura 30: Chipscope Pro Core Inserter (Paso 4)	35
Figura 31: Modo de uso del Chipscope Pro Analyzer	36
Figura 32: Interfaz Chipscope Analyzer	36
Figura 33: Selección de tipo de conexión con la FPGA	37
Figura 34: Visualización de señales en Chipscope Analyzer	37
Figura 35: Generación del controlador con MIG: paso 1	39
Figura 36: Generación del controlador con MIG: paso 2	40
Figura 37: Generación del controlador con MIG: paso 3	41
Figura 38: Generación del controlador con MIG: paso 4	42
Figura 39: Generación del controlador con MIG: paso 5	42
Figura 40: Ventana principal de iMPACT	43
Figura 41: Jerarquía del controlador DDR SDRAM con banco de pruebas	45
Figura 42: Máquina de estados de controller_64bit_00	47
Figura 43: Máquina de estados del módulo cmd_fsm (banco de pruebas)	48
Figura 44: Máquina de estados del módulo cmd_fsm (banco de pruebas) con refresco	50
Figura 45: Jerarquía del controlador DDR SDRAM sin banco de pruebas	53
Figura 46: Esquema del controlador de memoria	54
Figura 47: Esquema del registro de configuración	57
Figura 48: Máquina de estados del módulo cmd_fsm sin banco de pruebas	58
Figura 49: División del área de ejecución	59
Figura 50: Combinación de particiones	60
Figura 51: Esquema del módulo planificador	61
Figura 52: Conexiones entre el planificador y el entorno	63
Figura 53: Protocolo de planificación de tareas	63
Figura 54: Protocolo de ubicación de tareas	64

Figura 55: Campos en que se divide una tarea	65
Figura 56: Visión general del planificador	66
Figura 57: Maquina de estados de la unidad de planificación	68
Figura 58: Primera etapa de la ruta de datos de la unidad de planificación	70
Figura 59: Segunda etapa de la ruta de datos de la unidad de planificación	71
Figura 60: Tercera etapa de la ruta de datos de la unidad de planificación	72
Figura 61: Esquema de una cola fifo	75
Figura 62: Arquitectura general de la cola fifo	76
Figura 63: Arquitectura de la fifo en la primera fase de desarrollo	77
Figura 64: Contenido del registro que almacena el último gap	78
Figura 65: Valores que identifican un gap	78
Figura 66: Arquitectura de la fifo en la segunda fase de desarrollo	78
Figura 67: Opciones para tratar la inserción en gap	79
Figura 68: Fifo implementada como una lista dinámica	79
Figura 69: Dato almacenado en la BRAM	80
Figura 70: fifo simple implementada como una lista dinámicamente enlazada	80
Figura 71: Ruta de datos final de la fifo	81
Figura 72: Máquina de estados del controlador de la fifo	82
Figura 73: Simulación 1 de la fifo	83
Figura 74: Simulación 2 de la fifo	84
Figura 75: Diseño del banco de pruebas	85
Figura 76: Planificación de las tareas 1 a la 7	86
Figura 77: Planificación de las tareas 8 a la 14	87
Figura 78: Planificación de las tareas 15 a la 21	88
Figura 79: Planificación de las tareas 22 a la 26	88
Figura 80: Planificación de las tareas 27 a la 32	89
Figura 81: Ejecución de tareas en los dos primeros slots	89
Figura 82: Ejecución de tareas desde el slot tercero al octavo	89

Figura 83: Ejecución de tareas desde el slot noveno al decimocuarto	90
Figura 84: Ejecución de tareas desde el slot decimoquinto al decimoctavo	90
Figura 85: Esquema del controlador vga	94
Figura 86: Imagen mostrada por el vga para la memoria	97
Figura 87: Imagen mostrada por el vga para el planificador	98

E. Índice de tablas

Tabla	Pagina
Tabla 1: Chips y módulos de memoria DDR	19
Tabla 2: Entrada/Salida de un módulo DDR SDRAM	20
Tabla 3: Entradas de un módulo DDR SDRAM	21
Tabla 4: Tabla de verdad de los comandos DDR	24
Tabla 5: Códigos de operación	49
Tabla 6: Entrada/Salida del controlador de memoria	55
Tabla 7: Entradas del controlador de memoria	55
Tabla 8: Salidas del controlador de memoria	56
Tabla 9: Entradas del planificador	62
Tabla 10: Salidas del planificador	62
Tabla 11: Entradas de la unidad de planificación	67
Tabla 12: Salidas de la unidad de planificación	68
Tabla 13: Salidas Moore de la unidad de planificación	73
Tabla 14: Salidas Mealy de la unidad de planificación	74
Tabla 15: Entradas de la fifo	75
Tabla 16: Salidas de la fifo	76
Tabla 17: Resoluciones y frecuencias del controlador vga	95
Tabla 18: Entradas del controlador vga	96
Tabla 19: Salidas del controlador vga	97

F. Palabras clave

- Cola fifo
- DCM
- DDR
- FPGA
- Planificador
- Reconfiguración
- Ubicador
- Verilog
- VGA
- VHDL
- Vitex II PRO
- Xilinx
- XUP