UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

PROYECTO FIN DE CARRERA

Diseño de un sistema de lectura y procesado para múltiples sensores embebidos en una FPGA

Marina Aparicio Rodríguez

Diciembre 2007

Diseño de un sistema de lectura y procesado para múltiples sensores embebidos en una FPGA

AUTOR: Dª. Marina Aparicio Rodríguez

TUTOR: D. Eduardo Boemo / Gustavo Sutter

Dpto. de Ingeniería Informática

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Octubre 2007

PROYECTO FIN DE CARRERA

Título:

Diseño de un sistema de lectura y procesado para múltiples sensores embebidos en una FPGA

Autor: Dª. Marina Aparicio Rodríguez

Tutor: D. Eduardo Boemo / Gustavo Sutter

Tribunal:

Presidente: D. Ángel de Castro Martín

Vocal: D. Elías Todorovich

Vocal secretario: D. Eduardo Boemo Scalvinoni

Fecha de lectura: Madrid, a de de 200

Calificación:

Palabras clave:

Prototipo, FPGAs, sensores, sistema de control, microprocesadores embebidos.

Prototype, FPGAs, sensors, control system, embedded microprocessor.

Resumen

Este proyecto describe un prototipo de un módulo de control para un telescopio de electroscopia del IAC. Este nuevo sistema remplazará al anterior sistema que era demasiado lento y manual. El sistema se encuentra dentro de un entorno de criogenización, lo que implica trabajar en sistema con parámetros inestables. Para generar un sistema automático se desarrollará un sistema de control realimentado que permita posicionar las barras (que determinan la apertura de una lente) mediante un microprocesador embebido en una FPGA.

Este documento analiza los diferentes dispositivos electrónicos y herramientas disponibles en el mercado, describe el sistema de control prototipo implementado y el interfaz gráfico desarrollado para mostrar el movimiento mecánico del sistema.

Abstract

This project is based on the developments of a control prototype for a spectrum telescope of the IAC. This new control system will replace the last system used in the telescope to take samples of the universe's radiation. It attempts to improve the current system by eliminating the manual steps necessary and by generally speeding up the entire configuration process. The main objective is to reduce the layout when a new sample is taken.An automatic control system is designed using a microprocessor embedded in a FPGA to achieve this objective

The control system will be placed in a cryogenic system where the required constraints, i.e. the election of the sensor and the actuator, will be determined by the temperature.

This document analyzes the microprocessors and tools that currently exits in the marketplace. It then goes on to give an overview of the design of the control system prototype. The system that is described provides a parameterized and a feedback control.The project also includes the development of a graphic interface that allows to simulating the movements of the mechanic system.

Agradecimiento

Me gustaría agradecer este proyecto a Eduardo Boemo y Gustavo Sutter, que me han dedicado su tiempo y ayudado durante este proyecto fin de carrera, además de haber sido unos excelentes profesores durante mis años de estudiante. También recordar a todos los profesores de la escuela que han contribuido con su esfuerzo y cercanía a formarnos.

Querría dedicar este proyecto a mis padres, por su cariño y apoyo, y en especial a mi madre, por aguantar los momentos de estrés y desesperación estoicamente. A mis amigos, que supieron escucharme cuando lo necesité y corregirme cuando me equivocaba, y a todos ellos que decidieron dedicarme su tiempo y me enseñaron lo poco que sé.

Marina Aparicio Rodriguez

Diciembre del 2007

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Origen del proyecto

Este proyecto se ubica dentro de un proyecto en colaboración con la IAC (Instituto de Astrofísica de Canarias), para el desarrollo de un prototipo para el control de un telescopio de gran envergadura en la isla de Tenerife.

Todos los astros que conforman el universo transmiten ondas electromagnéticas. El espectro de estas ondas dependerá de aspectos como la composición química, la temperatura, la densidad, la presión o el grado de ionización. Por ello, se han desarrollado telescopios que captan el espectro de diferentes áreas del universo, permitiendo así realizar un análisis espectral para obtener información sobre los astros contenidos en estas áreas. A esta técnica se le denomina espectroscopia. La principal característica de este telescopio es que se encuentra en un entorno a temperaturas criogénicas. El medio criogénico dificulta el control de los parámetros y el manejo de los sistemas electrónicos, y determinará de manera notable el diseño del sistema de control.

El telescopio estará compuesto por una placa pixelada que recibirá las radiaciones electromagnéticas a estudiar. Para determinar el área del espectro electromagnético sometido al estudio del telescopio el sistema dispondrá de un mecanismo situado encima de la placa pixelada, formado por un conjunto de barras pareadas, que permitirán describir el espacio de toma de datos de la lente. El sistema describirá pequeñas franjas por las que pasaran las radiaciones electromagnéticas al área de captación.

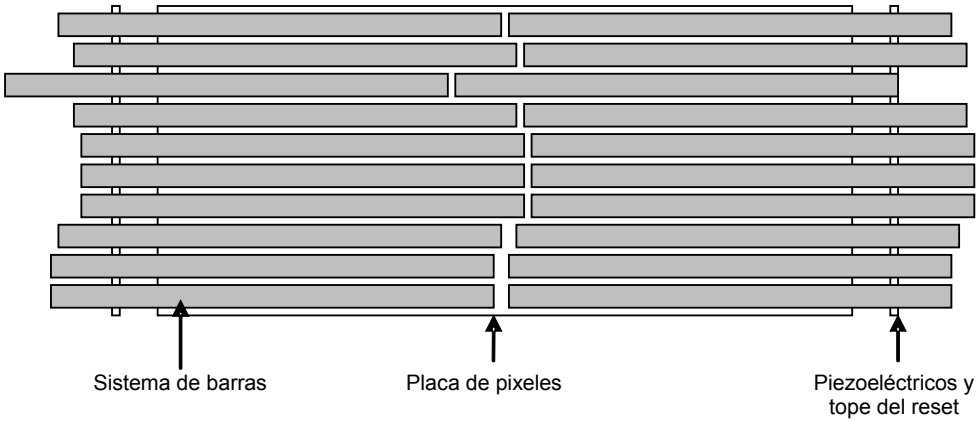


Figura 1: Esquema del sistema de barras.

Los anteriores sistemas disponían un mecanismo de bandejas. Cada vez que se deseaba tomar una nueva medida se debía de sustituir la bandeja por otra nueva. Sin embargo, los cambios de bandejas se debían de realizar a temperatura ambiente, lo que encarecía y ralentizaba el proceso. El proyecto surge por la necesidad de renovar el antigua sistema utilizado para la toma de datos. El nuevo sistema está formado por entre 50 y 55 pares de barras. Estas barras están hechas en aluminio y diseñadas de modo que se pueden desplazar con la menor fricción posible en el sistema. El objetivo del proyecto será desarrollar el sistema de control que permita controlar y modelar la toma de datos del telescopio mediante el control de las barras.

1.2 Antecedentes

Los anteriores telescopios basados en la técnica de la espectroscopia utilizaban como sistema de captura de datos una bandeja que se situaban encima de la placa de pixeles. El antiguo sistema tenía como principal desventaja la necesidad de tener que cambiar las bandejas a temperatura ambiente cada vez que se deseara tomar una muestra distinta a la configurada en ese instante.

A continuación se analizan las principales desventajas que presenta el sistema debido a encontrase en un entorno a bajas temperaturas.

El telescopio se encuentra a 70 grados Kelvin. Teniendo en cuenta que el cambio de bandejas se debe realizar a temperatura, será necesario tener el recinto a temperatura ambiente y luego volver a enfriarlo. Esto supone un gasto energético muy elevado (del orden de miles de euros) además de que el proceso de cambio de bandeja y enfriamiento podía llegar a durar varios días.

El sistema permite un número limitado de enfriamientos, ya que el mecanismo se puede ver afectado por los cambios de temperatura constantes.

1.3 Objetivos

En lugar de este sistema, se desea instalar un nuevo sistema formado por barras conectadas un actuador piezoeléctrico. Estas barras se desplazaran colocándose en las posiciones necesarias para realizar las medidas. Con este nuevo sistema se desea conseguir reducir el tiempo de configuración al orden de 2 minutos (frente al retardo de días que suponía le antiguo sistema).

Los objetivos o hitos propuestos para la implementación del prototipo:

Análisis de las características del diseño y las especificaciones del prototipo.

Análisis y documentación sobre las posibles opciones en el mercado para la implementación del sistema de control en una FPGA.

Descripción de sistema cuya estructura básica deberá estar formada por un módulo capaz de leer varios sensores, en principio se considerarán 4 asociados a cada actuador, y que combina sus valores aplicando un algoritmo, y que genere una señal de control sobre dos actuadores, de manera realimentada, acorde a la lectura obtenida de los sensores. Este bloque básico deberá ser replicado 50 veces y embebido en una única FPGA de tal manera que el sistema total constará de 400 sensores y 100 actuadores.

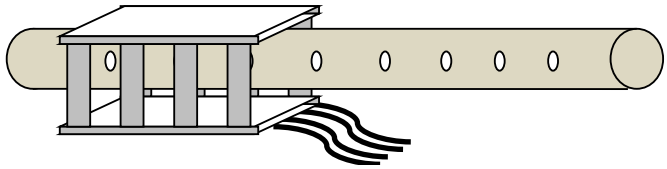


Figura 2: Descripción sistema de sensores por barra.

Descripción de un algoritmo inicial de control de las barras: algoritmo de control realimentado que basan los parámetros de desplazamiento en función de la posición actual de las barras.

Descripción de un programa de simulación que permita comunicarse mediante el puerto serie con la FPGA.

Obtención de los datos de ocupación del sistema.

Como resultado se debe obtener u prototipo que permita controlar el sistema, completamente parametrizable y que disponga de capacidad de adaptación a nuevas funciones de transferencia y parámetros debido a las condiciones variables del sistema criogenizado.

1.4 Organización de la memoria

Esta memoria está distribuida en seis capítulos:

Este primer capítulo comprende una breve descripción del proyecto, sus orígenes, objetivos y organización del mismo.

En el segundo capítulo se analizará la situación del estado del arte en los ámbitos vinculados al proyecto:

- Los sistemas criogenizados y más especialmente las implicaciones en los sistemas electrónicos.
- Características de las FPGAs, principales tecnología de fabricación y principales fabricantes en el mercado. Además, se realiza un breve repaso sobre los productos disponibles en el mercado aplicados a esta tecnología.

El tercer capítulo versa sobre las especificaciones del sistema, las posibles opciones disponibles en el mercado para la implementación del sistema y la justificación de las decisiones de diseño tomadas.

En el cuarto capítulo se describen todos los aspectos de desarrollo llevados a cabo en el proyecto. Se describe minuciosamente los periféricos diseñados, el microprocesador y al interfaz grafica de control del sistema.

En el quinto capítulo se abordan las características del sistema obtenido, las necesidades de ocupación y las características del mismo más restrictivas y que determinaran la elección del modelo de FPGA a utilizar.

En el sexto capítulo se describen las conclusiones obtenidas, las FPGAs que se pueden utilizar para implementar le prototipo y las posibles mejoras que se podrían incluir en el diseño.

2 Estado del arte

2.1 Introducción

A continuación se hace una breve descripción del estado del arte en los ámbitos más destacables reaccionados con el proyecto: las características de los sistemas de criogenización, una breve introducción a los dispositivos programables (fabricantes y tecnología de fabricación) y los tipos de procesadores embebidos en FPGAs.

2.2 Entornos de criogenización

Un aspecto determinante del proyecto está relacionado con las condiciones físicas en las que se encuentra inmerso el sistema. Este entorno determinara el diseño mecánico y también el diseño del controlador. Actualmente, la tecnología relativa a los entornos de criogenización costa de cuatro ámbitos de estudio fundamentales:

Sistemas de enfriado: Uso de gases licuados como Nitrógeno o Helio , y la utilización de sistemas mecánicos como los compresores en ciclo cerrado (Closed Cycle Coolers "CCC")

Aislantes térmicos: Vacío y cámaras de vacío. Criostatos (Cryostats). El diseño de sistemas criogenizados en entornos al vacio implica la aparición de la desgasificación en el alto vacio ("*outgassing"*). La NASA tiene publicada una lista de los materiales que soportar menor desgasificación en el vacío [26] que serán utilizados para este tipo de sistemas.

Comportamiento de materiales a baja temperatura: Estas temperaturas extremas implican que los materiales sufran efectos como la fragilidad y la contracción. Además de estos efectos, se ha de sumar la desgasificación debida al alto vacio [24,25].

Limitaciones de circuitos a bajas temperaturas: la incursión de dispositivos electrónicos en sistemas criogénicos implica una variación del funcionamiento de los mismos. Los dispositivos compuestos por semiconductores, los controladores *pwm*, los osciladores de lata frecuencia, los conversores analógico digital, sensores y transductores ven alterado su comportamiento debido a las bajas temperaturas y las condiciones en alto vacio [25].

2.3 FPGAs

2.3.1 Evolución de los FPD

Se conoce como FPD (Field Programable Device) a todos los circuitos digitales utilizados para implementar hardware, donde el chip puede ser configurado por el usuario para realizar diferentes diseños.

Las FPLAs (Field Programable Logic Array) o PLAs utilizando su nombre simplificado fueron los primeros dispositivos programables que aparecieron. Estos sistemas consistían en dos niveles de puertas lógicas, un plano de AND y otro de OR. Ambos niveles podían ser programados y dan así comienzo a la era de los dispositivos de hardware programable. Los PLAs (Programable Array Logic) aparecieron a principio de los años setenta como una variante de los FPLAs.

Sin embargo, el salto cualitativo se produjo con la aparición de los CPLD (*Complex field programable desvice*). Estos dispositivos permitieron mejorar la velocidad mediante la utilización de *arrays* de PLDs organizados en bloques [22].

2.3.2 Introducción FPGAs

Las FPGA (*Field Programmable Gate Arrays*), en principio llamadas LCA (*Logic Cell Array*) nacen en 1985 con una idea sencilla: un Gate Array tolerante a errores de diseño y programable o reprogramable por el usuario.

Las principales características de las FPGAs son:

Alta complejidad (106 puertas)

Bajo costo de desarrollo.

Fácil de depurar.

Tolerante a errores.

Pocas unidades.

Tamaño reducido.

Fiabilidad alta.

Área-velocidad-potencia intermedio.

Confidencialidad baja.

Un FPGA es un componente estándar reprogramable. Esto supone que tanto las interconexiones como las entradas y salidas deben ser reprogramables. Para ello, las pistas de interconexión están compuestas por transistores de paso. Mediante una memoria auxiliar se almacenan los 1s y 0s que determinan cada conexión en particular. En la siguiente imagen se puede observar un esquema de una línea de interconexión formada por transistores.

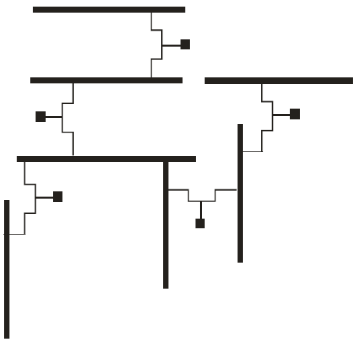


Figura 3: Esquema líneas de interconexión de una FPGA

Al igual que los Gate Arrays, las funciones lógicas se deben mapear en los transistores. Sin embargo, en este caso estas deben ser reprogramables. Por ello las funciones se mapean en una tabla “look-up” con la que podre emular cualquier puerta. Las direcciones son la entrada al circuito y el dato la salida del mismo. Al igual que en el caso de la interconexión, una memoria auxiliar y un circuito de configuración imponen los 1s y 0s de cada tabla.

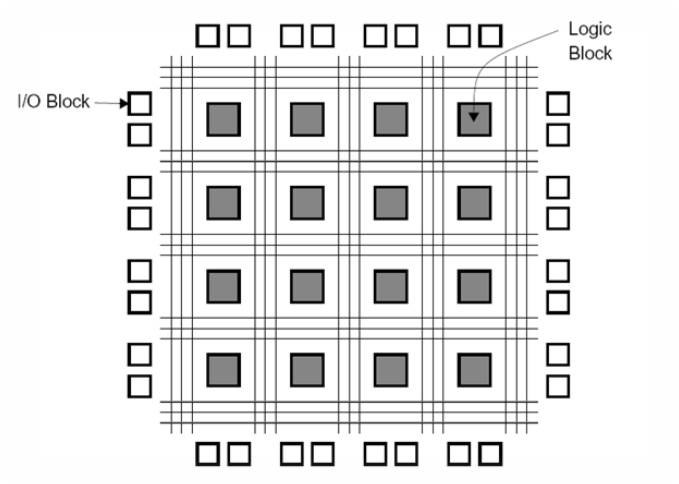


Figura 4: Estructura general de una FPGA.

2.3.3 Tecnologías de programación

A continuación se describen las principales tecnologías utilizadas para la configuración de los *array* de interconexión de las FPDs:

SRAM (Static random acess memory)[22,1]: la memoria SRAM es un tipo de memoria semiconductora. Los datos cargados en la memoria se mantienen mientras la alimentación se mantenga en la memoria. La siguiente figura muestra una celda de memoria SRAM. Cada bit se carga en cuatro transistores CMOS.

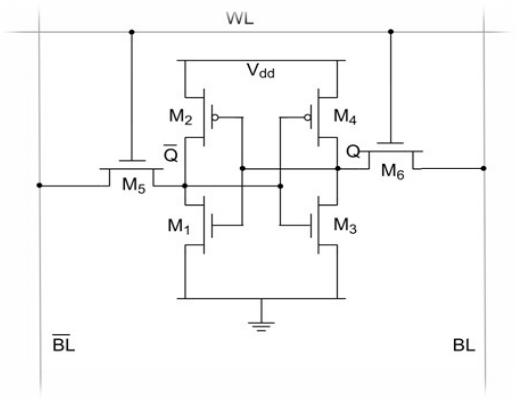


Figura 5: Estructura de una celda SRAM de memoria.

Las memorias SRAM se caracterizan por ser más caras que las memorias DRAM. Sin embargo, son más rápidas y tienen un menor consumo de energía, lo que las hace idóneas para ser utilizadas en sistemas embebidos y FPGAs.

La siguiente Figura muestra la estructura de bloques de una FPGA. Cada uno de los CLBs será configurado con la información contenida en las memorias SRAM de la FPGA.

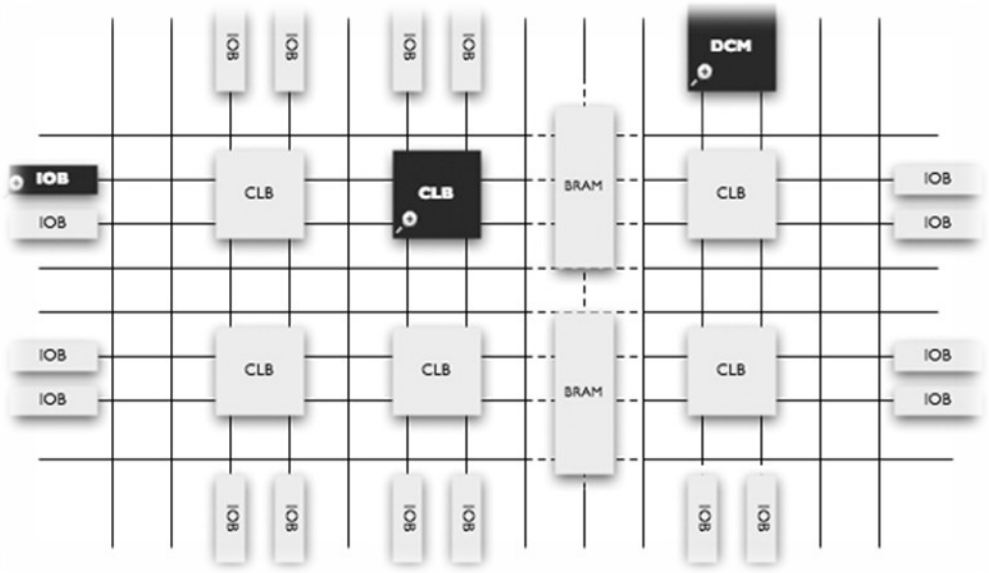


Figura 6: Estructura de una FPGA de Xilinx [1].

La siguiente figura muestra que las SRAM tienen dos aplicaciones: controlar los nodos de puertas y controlar la selección de líneas de los multiplexores que determinan como se conducen los bloques lógicos a las salidas del chip.

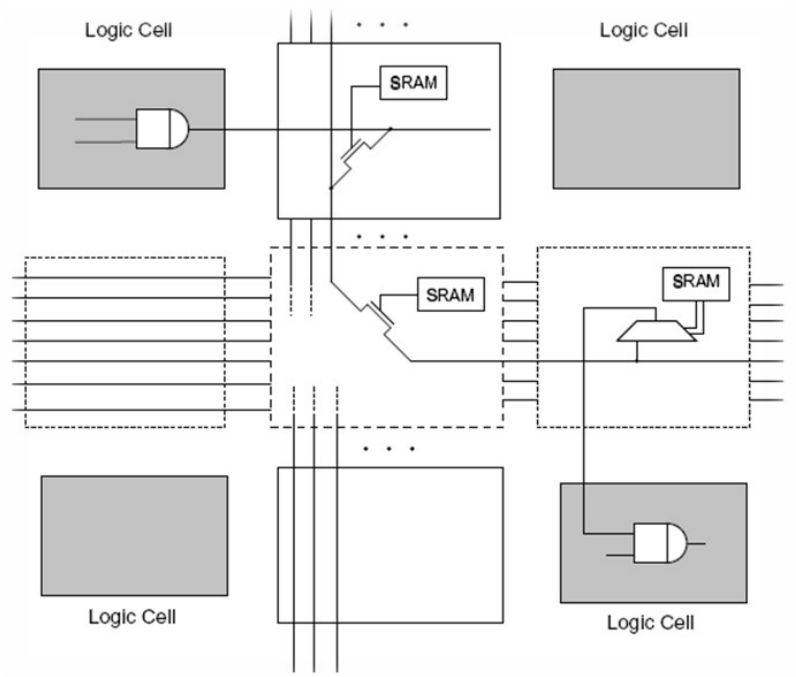


Figura 7: Programación mediante SRAM [22]

Antifuse [22,4]: se denomina *antifuse* a una tecnología de programación desarrollada por la compañía Actel para FPGAs. Mediante la disposición de matrices *antifuse* en el chip se podrán programar los *gate-array* de la FPGAs.

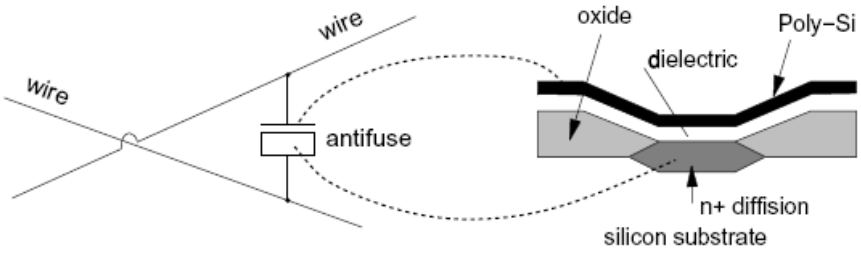


Figura 8: Estructura de un "antifuse" de Actel [22].

Las matrices anti-fusible consisten en líneas de conductores dispuestas en paralelo, generalmente de aluminio, recubiertas por una fina capa de dieléctrico (habitualmente oxido de silicio). Sobre el oxido se deposita otra capa de líneas conductoras esta vez colocadas perpendiculares a las anteriores. De esta manera, se pueden superponer varias capas de sistemas anti-fusible, lo que permite ahorrar al fabricante en área de silicio consumida por la FPGA, incluso permitiendo utilizar esta área para incluir más transistores.

Para programar la estructura se aplica una elevada tensión entre dos líneas que se cruzan. Si esta tensión es superior a la rigidez eléctrica del oxido, se producirá una ruptura de la estructura, y gracias a un pequeño arco situado entre ambos conductores se producirá la fusión de los dos líneas conductoras, soldándolas, y produciendo así una conexión permanente de ambas líneas [22].

EPROM: Estas memorias PROM son capaces de mantener al información cuando la alimentación de la memoria desaparece, es decir, son no volátiles. Una vez programada, una EPROM se puede borrar solamente mediante exposición a una fuerte luz ultravioleta. Por ello, un sistema que utilice esta tecnología solo podrá ser reprogramado fuera del propio circuito. La siguiente figura muestra la utilización de memorias EPROM para la conexión en un plano AND de un CPLD.

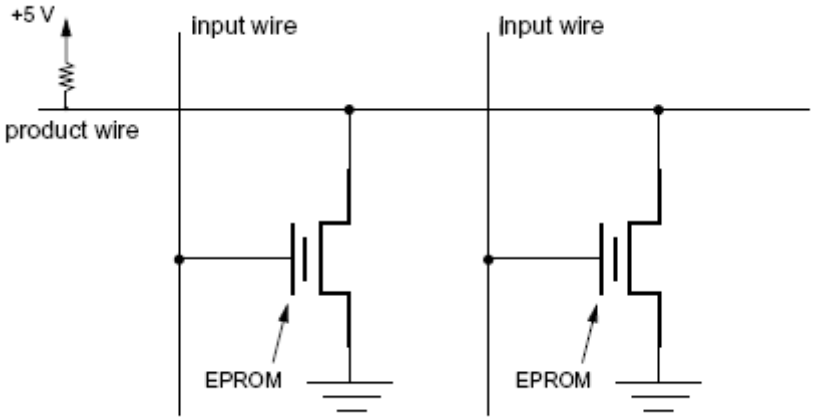


Figura 9: Programación mediante EPROM [18]

Flash: la memoria flash es una memoria no volátil. Las memoria son volátiles no necesitan mantener la alimentación de la memoria para mantener los datos cargados en ellas.

EEPROM: este tipo de memoria es no volátil, como la memoria flash. Es un tipo de memoria ROM que puede ser programado, borrado y reprogramado eléctricamente, a diferencia de la EPROM.

Fuse: se denomina fusible a un dispositivo, formado por un filamento o lámina de un metal o aleación de bajo punto de fusión que se intercala entre las interconexiones de rutado de un FPD. La programación se realiza fundiendo aquellas interconexiones que no serán necesarias en al diseño. Esta tecnología no permite la reprogramación del sistema.

Estas tecnologías permiten definir el rutado dentro de los dispositivos programables. Sin embargo, las tecnologías más populares en el caso de los dispositivos CPLD son las memorias EPROM y EEPROM, y para las FPGA la SRAM y la tecnología Anti-fuse [22]. A continuación se incluye un esquema resumen sobre las tecnologías de programación.

Nombre	Reprogramable	Volátil	Tecnología
Fuse	No	No	Bipolar
EPROM	Si, fuera del circuito	No	UVCMOS
EEPROM	Si, dentro del circuito	No	EECMOS
SRAM	Si, dentro del circuito	Si	CMOS
Antifuse	No	No	CMOS+

Tabla 1: Tecnologías de conexión en FPDs

2.3.4 Principales fabricantes

A continuación se enumeran los fabricantes de FPGAs de mayor peso en el mercado, los dipositivos electrónicos que fabrican y las herramientas, buses y microprocesadores que ofrecen:

Xilinx [1]:

Esta compañía es una de las líderes en el mercado de FPGAs y CPLD. Estos sistemas proveen de dispositivos electrónicos como soluciones para múltiples aplicaciones como aeroespaciales y de defensa, automovilísticas, procesado de datos, industria, ciencia o medicina.

Xilinx ofrece productos y servicios del tipo:

- Componentes electrónicos clasificados en tres familias: una familia de *glue logic* (CoolRunner™), FPGAs de bajo coste (Spartan™) y otra gama de alta calidad (Virtex™).
- Herramientas de diseño: para diseño lógico (ISE™), para diseño de sistemas embebidos (EDK™) y para procesado digital de señales (*System Generators for DPS* para tratamiento de señales con Matlab).
- IPs y diseños de referencia: dentro de estas IPs los más requeridos son el microcontrolador PicoBlaze™, el *soft-core* MicroBlaze™ y el *hard-core* PowerPC™.

La estructura básica de las FPGAs de Xilinx está formada por *array* de bloques lógicos que pueden ser interconectados vertical o horizontalmente mediante canales de rutado controlados por memorias BRAM.

Altera [2]:

Esta compañía es la principal competidora de Xilinx en volumen de mercado. Los principales productos que ofrece:

- Componentes electrónicos clasificados en tres familias: una familia de *CPLDs* (Max), FPGAs de bajo coste (Cyclone y Arria) y otra gama de alta calidad (Stratix).

- Herramientas de diseño: para FPGAs ofrece la herramienta MAX+PLUSII y la herramienta Quantus II.
- IPs de referencia: procesadores el procesador soft core Nios II y otros interfaces de comunicación.

Lattice Semiconductor [3]:

Una de las compañías fabricantes de FPGAs destacable debido a que la tecnología de programación utilizada es una combinación de la tecnología SRAM y Flash no volátil.

Los principales productos de la compañía:

- Componentes electrónicos: FPGAs de bajo coste (LatticeSC/M™) y otra gama de alta calidad (LatticeECP2M™).
- IPs de referencia: dentro de los procesadores, el procesador soft core más popular es Micos32 y otros interfaces de comunicación.

Actel [4]:

Entre los productos más populares de esta compañía cabe destacar:

- Procesadores ARM
- Procesor core Leon3 (*soft core*)
- On chip bus interface: AMBA

Esta compañía fue la impulsora de la tecnología Anti-fuse anteriormente descrita.

Atmel [5]:

Los productos más populares de esta compañía son:

- Componentes electrónicos: las FPGAs fabricadas más populares son AT40K y AT40KAL.
- IPs de referencia: dispone de microprocesadores soft-core MARC4, AVR8, AVR32 de 4,8 y 32 bits respectivamente.

Otras compañías:

- **Quicklogic [6]:** Desarrolla FPGAs mediante la tecnología *antifuse* (solo programables una vez). Sus productos actualmente están centrados en dar soluciones militares.
- **Achronix Semiconductor [7]:**Esta compañía desarrolla FPGAs de alta velocidad (del orden de 2GHz de velocidad)
- **MathStart [8]:** desarrolla una familia de FPGAs llamadas FPOA

Fabricante	FPGAs	Procesadores	Buses
Xilinx	Spartan™	MicroBlaze y	OPB,FSL,LMB

	Virtex™	PowerPC 405	
Altera	Cyclone, Arria Stratix	Nios II	Avalon
Lattice Semiconductor	LatticeECP2M™,LatticeSC/M™	MICO32	--
Actel	Axcelerator, Fusion,M1 Fusion, M7 Fusion	ARM7, Leon3	AMBA
Atmel	AT40K y AT40KAL	MARC4, AVR8, AVR32	--

Tabla 2: Fabricantes de FPGAs

2.4 Sistemas embebidos

El método de diseño denominado *System on Chip* (SoC) consiste en incluir IP (*Intelectual Properties*) dentro del propio chip. La inclusión de los IP facilita el diseño de los sistemas y lo optimiza gracias a un diseño óptimo y modular.

El tipo de IP que se suelen incluir en el chip suelen ser procesadores, que permiten realizar operaciones de procesado de señal. También es frecuente la incursión de memorias, que disponen de funciones especiales de almacenamiento y carga de datos.

Los procesadores se clasifican en tres grupos [23]:

> **Hard Core**: los procesadores "*Hard Core*" están optimizados para una tecnología específica y no pueden ser modificados por el diseñador que los utiliza. Estos procesadores tienen un *layout* predefinido y un *floorplan* que están incluidos en la arquitectura del diseño. Tiene la ventaja de el *timing* es fijo y puede ser modelado como librería de elementos. Las fundamentales desventajas de los "*hard cores*" es que no pueden ser modificados por el diseñador ni se pueden ajustar los *timing* al diseño utilizado. Sin embargo, presentan una frecuencia de trabajo mayor que los "*soft core*".

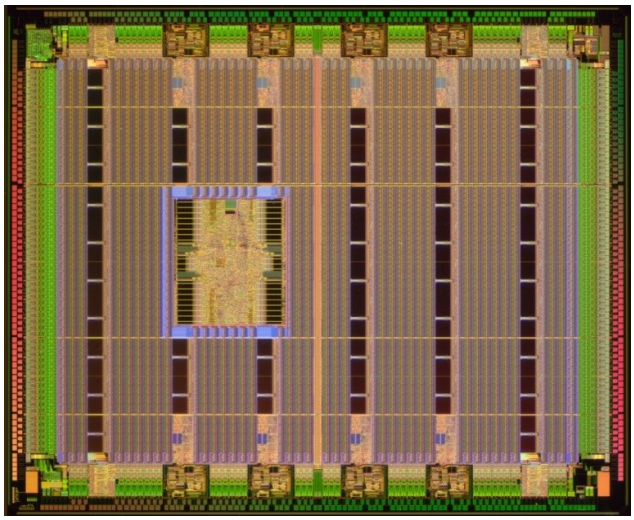


Figura 10: Aspecto de una FPGA con un procesador "Hard Core" [1].

Firm cores: los procesadores "*firm cores*" están definidos como una mezcla de código fuente y *netlist* dependientes de la tecnología utilizada. En este tipo de procesadores, el código fuente es visible para el diseñador y específicas partes del mismo podrán ser modificadas por el diseñador. Sin embargo, el *netlist* está determinado por la tecnología, y por tanto, el usuario se verá obligado a utilizar estos procesadores sobre los chips del mismo fabricante.

Soft Core: los procesadores "*Soft Core*" se caracterizan por estar definidos como sistemas mediante código sintetizable HDL. Esto permite que el diseñador pueda modificar algunas de las características del procesador, esto permite expandir su funcionalidad y flexibilizar el diseño. Sin embargo, tienen un mayor costo y una potencia de procesamiento menor.

	Hard cores	Firm cores	Soft cores
Dureza	*Layout* predefinido	Mezcla de código fuente y tecnología dependiente de la *netlist*	Dependiente del comportamiento del código
Modelado	Modelado como librería de elementos	Mezcla de bloques fijos y sintetizables que pueden ser compartidos por otros *cores*	Sintetizable con otra lógica
Flexibilidad	No puede ser modificado pro el diseñador. La utilización de varios hard core en un chip puede resultar ineficiente	Tecnología dependiente	El diseño puede variarse
Predictibilidad	Garantiza los *timing*	Camino critico es fijo	El *timing* no está garantizado.
Coste	bajo	medio	alto
Descripción	Ficheros *layout* y *timing information*	Código sintetizable HDL y ficheros *layout* y *timing information*	Código sintetizable HDL

Tabla 3: Tabla resumen de los tipos de "cores"[24]

3 Diseño

3.1 Introducción.

En este capítulo se analizaran las opciones de diseño del proyecto. Para ello, primero se realiza una descripción de las especificaciones del sistema. En función de estas especificaciones se determina el tamaño de nuestros datos y el tipo de operaciones necesarias. Este análisis será la base para la toma de decisiones de diseño.

3.2 Especificaciones del sistema.

El diseño del sistema se va a realizar siempre haciendo módulos parametrizables. A pesar de ello, será necesario determinar los requerimientos mínimos del sistema. El sistema se diseñará de módulo que se diseñara un bloque básico que se replicara tantas veces como sea necesario para controlar todas las barras. El bloque básico controlará tan solo 8 barras (4 pares de barras pareadas).

3.2.1 Descripción del sistema

Se trata de un mecanismo que dispone de 100 barras, enfrentadas 50 a 50, y dotadas de 4 sensores de posición y un actuador piezoeléctrico por barra. Dichas barras han de desplazarse, atendiendo a un control realimentado, y ser posicionadas con precisión en las posiciones indicadas.

Por tanto el mecanismo total consta de 50 pares de barras enfrentadas donde el control de cada par de barras es independiente del resto. Se propone por tanto que se genere el sistema global de control del mecanismo como una réplica de elementos de control asociados a cada par de barras enfrentadas. Inicialmente, se propuso un esquema en el cual cada par de barras se controlara mediante un microcontrolador PicoBlaze. El esquema del sistema propuesto se puede ver en la Figura 11.

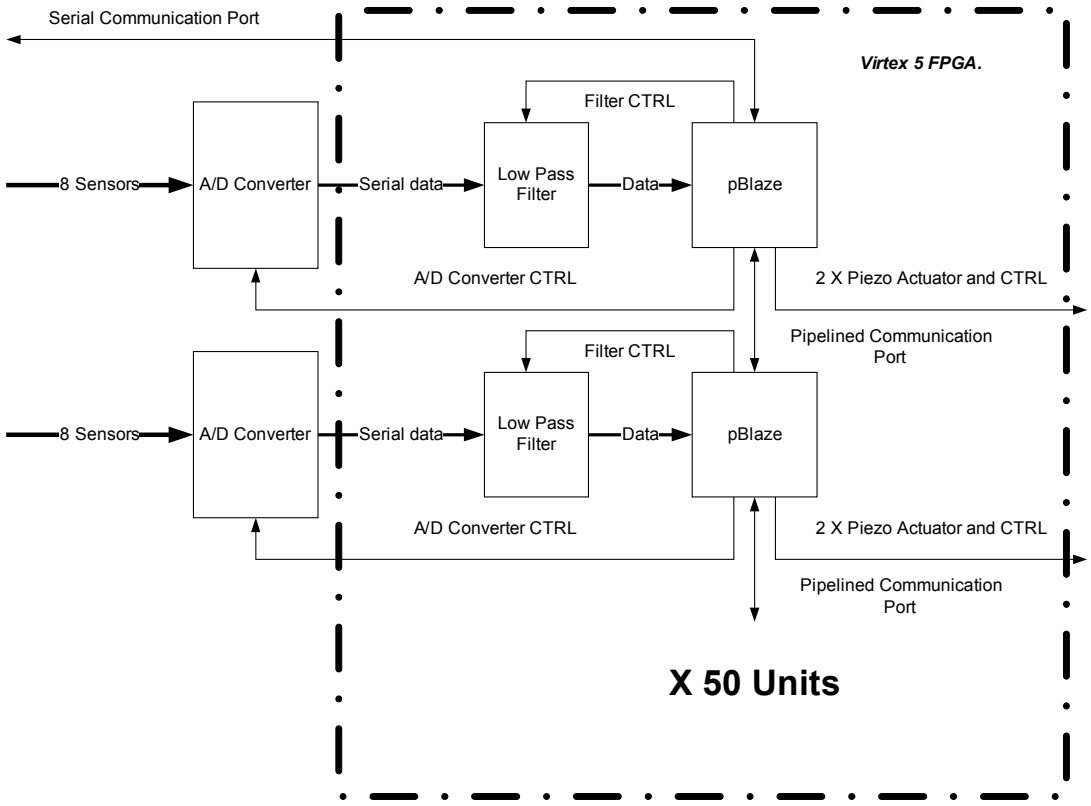


Figura 11: Propuesta inicial de diseño.

3.2.2 Sensores

El sistema cuenta con 4 sensores asociados a cada actuador que determinan la posición de la barra. Dichos sensores, tras combinar sus valores, proporcionan la información de posición necesaria para determinar la señal a aplicar al actuador.

Es importante considerar que el tipo de sensores utilizados vendrá determinado por el entorno criogénico en el que encuentra emplazado el diseño. Los sensores basados en capacidades variables tienen un funcionamiento lineal a bajas temperaturas frente a otro tipo de sensores de codificación, como los optoelectrónicos.

3.2.3 Combinación de sensores.

La combinación de los sensores es necesaria ya que, debido a su naturaleza, cada uno por separado no proporciona una información completa sobre la posición de la barra en cuestión. El algoritmo de recombinación ha de ser determinado, implementado y ejecutado por el sistema de control para así disponer de la información necesaria para el posicionado correcto de cada barra.

Puesto que no se dispone a lo largo del desarrollo del sistema de dichos sensores, se deberá crear un módulo que genere lecturas simuladas de éstos. Como regla general se asumirá que los valores de dichos sensores corresponden a valores de 16 bits.

La manera de combinar los sensores se estudiará a continuación con más detenimiento. Se propuso inicialmente que sea implementara el controlador mediante la utilización de en un microcontrolador (PicoBlaze). Inicialmente se informó de que el controlador seria sencillo, por ejemplo con una función de transferencia lineal, hasta que se determine con exactitud el algoritmo a implementar. Este punto ha de ser considerado con cautela ya que algún algoritmo podría requerir una potencia de cálculo excesiva, operaciones en punto flotante.

3.2.4 Señal sobre los actuadores

Las señales de control sobre los actuadores consistirán en 2 señales. Una primera señal determina el sentido del actuador y corresponde a una señal lógica 1 ó 0. La otra señal corresponde a la que determina la frecuencia de actuación y debe generarse una onda cuadrada para la que tanto la frecuencia como el ciclo de trabajo (*duty cycle*) debe ser programable.

Al igual que los sensores, que el sistema se encuentre a temperaturas criogénicas determinará la elección del actuador. En este caso, los piezoeléctricos suponen grades ventajas en este ámbito (tanto en consumo de potencia como en tamaño) frente a los motores [Apéndice D]

3.2.5 Control en lazo cerrado

El control se realizará en lazo cerrado. Se utilizará la señal obtenida de la combinación de sensores para determinar el sentido en el que ha de ejercer el actuador y la frecuencia con la que debe actuar. El sistema debería prever dos tipos de funcionamiento, uno burdo para cuando se está lejos de la posición deseada, y otro fino para cuando se está en las proximidades del punto final.

Los valores del ciclo de trabajo están relacionados con el desplazamiento que obtiene el piezoeléctrico sobre la barra. Esto deberá ser contemplado a la hora de simular los valores obtenidos por los sensores ya que los desplazamientos esperados deben depender del ciclo de trabajo aplicado.

3.2.6 Sistema de adquisición de datos

En previsión de que durante las pruebas del sistema será necesario transferir información entre el usuario y el sistema, se deberá disponer de una línea de comunicación. Esta línea de comunicación deberá ser una líneas serie RS-232 en cascada que fluya información entre los microcontroladores y el usuario y entre los propios microcontroladores.

3.3 Descripción de requerimientos de precisión del sistema.

Este sistema necesita tener la posibilidad de posicionar las barras con una precisión menor de 4 nm. Por tanto, será necesario conocer las dimensiones del sistema mecánico para determinar el número de bits necesarios para expresar una posición absoluta.

El sistema está compuesto por barras, que se moverán de manera independiente y dispondrán de 4 capacitores que realizan la función de sensores. El sistema de barras perforadas y los 4 sensores formados por capacitores determinan un sistema formado por dos *encoder*, uno relativo y otro absoluto.

Las perforaciones de las barras determinaran la posición absoluta entre dos perforaciones de la barra. Por tanto, este sistema es un *encoder* absoluto. Las perforaciones se encuentran situadas cada 11mm. Por tanto, si se necesita una precisión menor de 4 nm debe de ser posible definirá al menos 2,750 posiciones diferentes. Para describir la posición absoluta entre dos perforaciones de la barra serían necesarios un mínimo 12 bits.

Con relación al *encoder* absoluto, el ancho de la placa de pixel es de 33 cm, por lo tanto, cada una de las barras de aluminio que componen el sistema tendrá una longitud de 33 cm. Por tanto cada barra contendrá 30 vueltas diferentes, por tanto serán necesario 5 bits para describir el número de vuelta en el que se encuentran situados los capacitores.

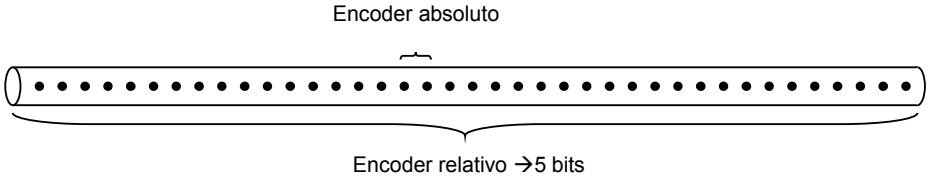


Figura 12: Descripción del* encoder *relativo y el* encoder *absoluto.

Por lo tanto, la serán necesarios al menos 17 bits para describir la posición absoluta dentro de una barra con una precisión menor de 4 nm.

3.4 Posibles opciones de diseño.

En este apartado se analizarán las diferentes posibilidades de diseño que actualmente nos oferta el mercado. En base a las especificaciones de diseño del sistema y las características tanto técnicas como conceptuales, se muestran las diferentes opciones, con sus desventajas y ventajas en relación con nuestro diseño.

Mediante le hardware a medida se podría diseñar un hardware de control PID (integral y diferencial). A pesar de que esta opción sería más sencilla de implementar, es probable que el diseño necesite de un algoritmo de control de movimiento más complejo. Por ello, es necesario incluir un microprocesador, que permita adaptar el algoritmo de control del sistema a las especificaciones particulares del sistema una vez instalado el mismo.

La primera decisión que se tomó fue la utilización de una FPGA como base para nuestro diseño. Las FPGAs tienen como principal característica que son reprogramables, lo que dotará a estos sistemas de una gran flexibilidad, además de poseer un precio notablemente inferior frente a un ASICs (siempre considerando el precio unitario de una FPGA y la fabricación de un ASIC a medida). Frente a estas ventajas, las FPGAs son sistemas más lentos y consumen más potencia que un ASIC.

Utilizando una FPGA se tiene a su vez dos posibilidades. Se puede diseñar un hardware a medida o utilizar un microprocesador embebido en la FPGA. En base a estas dos opciones, se puede realizar diseños que combinen ambas opciones. En este diseño se opto por una combinación de microprocesadores embebidos y de hardware a medida, denominado co-diseño hardware software.

Frente a los hard-core, los microprocesadores soft-cores son más flexibles y portables, lo cual son características básicas para el diseño a realizar. La inclusión de microprocesadores embebidos nos permite replicar el mismo microprocesador tantas veces como permita la capacidad de la placa. Así se obtienen sistemas que funcionan de manera paralela e independiente.

El hardware a medida permite diseñar periféricos para nuestro microprocesador. En el caso de este diseño, se puede incluir mediante el hardware a medida un módulo que simule el movimiento del sistema.

A continuación se muestran una breve descripción de las posibles opciones de microprocesadores aplicables a nuestro sistema:

3.4.1 Utilización de un microcontrolador.

Una de las opciones de diseño pasaría por la utilización de microcontroladores. A continuación se analizan las características de uno de los microcontroladores de código libre de Xilinx, ‘PicoBlaze’ [21].

PicoBlaze es un “*soft core*” de 8 bits, es decir, un microcontrolador diseñado para ser empotrado íntegramente dentro de una FPGA (Virtex-E, Virtex-II, Virtex-II/Pro, Virtex-4, Spartan-II, Spartan-IIE y Spartan-3).

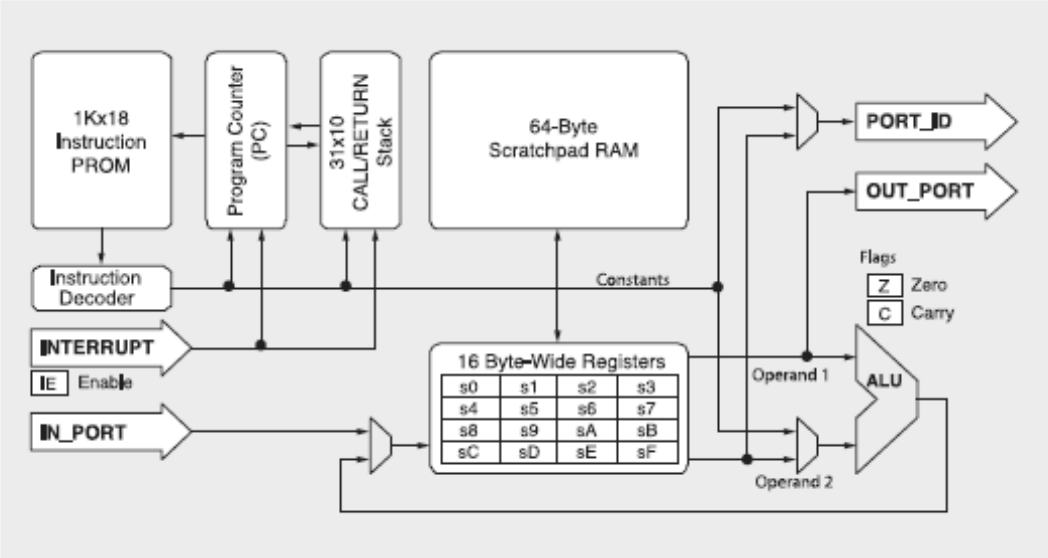


Figura 13: Diagrama de bloques del microntrolador PicoBlaze de Xilinx [28].

Existen diferentes tipos de arquitecturas del microcontrolador para optimizarlo para cada tipo de FPGA. A continuación se muestra una tabla comparativa con las principales características de PicoBlaze en los diferentes tipos de FPGA [28].

Características	PicoBlaze para Virtex-E y Spartan-IIE	PicoBlaze para Virtex-II y Virtex-II/Pro	PicoBlaze para Spartan-3 y Virtex-II/Pro y Virtex-4
Tamaño de instrucción	16 bits	18 bits	18 bits
Registros de 8 bits	16	32	16
Espacio de programa	256 instrucciones	1024 instrucciones	1024 instrucciones
Profundidad de la pila	15	31	31
Ensamblador	KCPSM	KCPSM2	KCPSM3
Tamaño	76 slices de una Spartan-IIE	84 slices de una Virtex-II	96 slices de una Spartan-3
Rendimiento	37 MIPS (Spartan-IIE)	55.8 MIPS (Virtex-II)	44 MIPS (Spartan-3) y 100 MIPS(Virtex-II/Pro)
Memoria	--	--	64 bytes

Scratch-Pad			

Tabla 4: Principales características de PicoBlaze en diferentes FPGAs de Xilinx[28]

Pero PicoBlaze presenta algunas desventajas con respecto a las especificaciones de este diseño:

Número limitado de instrucción (en el caso de utilizar una Virtex-4 es de 1024).

El fabricante propone varias soluciones para poder aumentar el número de instrucciones posibles, pudiendo llegar a obtener una capacidad de 2048 instrucciones. Sin embargo, el número de instrucciones permitidas es bastante escaso, lo que podría impedir incluir una mayor funcionalidad en el sistema, e incluso, podría ser insuficiente para implementar el diseño.

En su juego de instrucciones no dispone de multiplicación ni división. Este aspecto es de gran importancia si se considera que puede ser necesario en el procesado de los datos la utilización de estas instrucciones. Por tanto, las instrucciones que no están implementadas en el microcontrolador tendrían que ser implementarlas en código ensamblador en el propio programa, lo cual resulta costoso y reduce la cantidad de memoria disponible para el propio programa.

En PicoBlaze los datos son de 8 bits, mientras que en nuestro diseño los datos tienen un tamaño mayor.

3.4.2 Utilización de microprocesador de 16 bits.

Existen gran variedad de microprocesadores de código libre, que disponen de un amplio conjunto de instrucciones y funcionalidad.

Sin embargo, esta opción tiene varias desventajas:

Estos sistemas no disponen de herramientas para su diseño y programación como las que soportadas por el fabricante de FPGAs para sus microprocesadores.

Al igual que en caso anterior, el sistema necesita de datos mayor de 16 bits, lo que haría más complejo el diseño.

3.4.3 Utilización de Micros de 32 bits

Existen gran variedad de microprocesadores de 32 bits (gratuitos, de pago y embebidos).a continuación se describe el "*soft core*" de Xilinx.

3.4.3.1 MicroBlaze

MicroBlaze [14] es un procesador "*soft core*" con un número reducido de instrucciones (RISC) optimizado para ser implementado en FPGAs. Permite un alto grado de configuración.

Sus principales características son:

Datos de 32 bits
Instrucciones de 32 bits por palabra(con 3 operados y dos modos de direccionamiento)
32 registros de propósito general de un tamaño de 32 bits.
Un pipeline de 3 estados a 5 estados (para la versión v6.00 de MicroBlaze).
Soporta interrupciones y excepciones hardware.

Dispones de buses:

Arquitectura completamente Harvard
Bus OPB, de instrucciones y de datos.

A continuación se muestra una pequeña tabla con las características de tamaño y velocidad correspondientes a las diferentes FPGAs existentes en el mercado que soportan MicroBlaze.

Familia	Velocidad	Operaciones	Celdas lógicas
Virtex-II Pro	150 MHz	102 D-MIPS	900
Virtex-II	125 MHz	82 D-MIPS	900
Virtex-E	75 MHz	49 D-MIPS	1050
Spartan-II	65 MHz	43 D-MIPS	1050
Spartan-IIE	75 MHz	49 D-MIPS	1050

Tabla 5: Características de MicroBlaze[14]

Esta opción de diseño presenta como desventajas que es:

MicroBlaze tiene una estructura mucho más compleja que los anteriores sistemas mencionados.

Mayor tamaño. Esto se debe a que su estructura es más compleja. Puede resultar una grave desventaja si se quiere incluir varios microprocesadores dentro de una misma FPGA.

Podría ser necesario incluir memoria externa, lo que encarecería el valor final de la FPGA.

Sin embargo, tiene como mejora frente al microcontrolador "PicoBlaze":

Más velocidad.

El microprocesador puede ser programado en lenguaje de alta programación (lenguaje C++) frente al microcontrolador PicoBlaze que se debe programar en ensamblador.

Dispone de un mayor juego de instrucciones y funcionalidad. Permitirá controlar varias barras a la vez.

Dispone de librerías específicas para aritmética de datos enteros y de datos en punto flotante [31].

Más información con respecto al "*soft core"* MicroBlaze en el apéndice A del documento.

4 Desarrollo

4.1 Descripción del sistema

Una vez analizadas las especificaciones básicas del sistema se debe de concretar la estructura general del diseño.

El siguiente esquema muestra la jerarquía de diseño. En la base se la pirámide de jerarquía se encuentran los periféricos del sistema. Estos módulos son diseños a medida, diseñados en lenguaje de HDL (en este caso mediante la utilización de VHDL) mediante la utilización de la herramienta ISE de Xilinx. Estos periféricos se comunicaran mediante el bus OPB de Xilinx a los microcontroladores.

El segundo nivel de la jerarquía estará compuesto por los microntroladores embebidos en la FPGA, y el nivel superior de diseño estará formado por la FPGA.

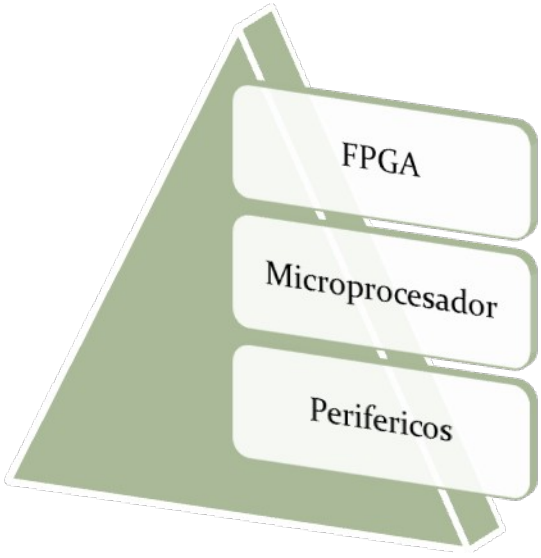


Figura 14: Esquema del diseño.

En base a las características repetitivas del sistema se opta por describir un diseño mediante estructuras replicadas. Se diseñará un bloque básico que controlara 8 barras (se opta por este número de barras debido a que permite agrupar los datos en bytes), el cual se replicara dentro de

la FPGA tantas veces como sea necesario, hasta controlar con el todas las barras que compongan el sistema de control del telescopio.

Sin embargo, la replicación de módulos se realizara en varios de los niveles de diseño del sistema:

Un primer nivel estará comprendido por los módulos específicos de control individual de cada barra. Estos módulos serán agrupados en un único periférico que a su vez estará comunicado con un microcontrolador.

El segundo nivel de complejidad estará formado por el conjunto de microprocesadores MicroBlaze, que tendrán la función de controlar cada uno un grupo de barras.

Por último, todos los microprocesadores estarán embebidos en una única FPGA que controlará todo el sistema.

A continuación se muestra un pequeño esquema explicativo sobre la estructura general.

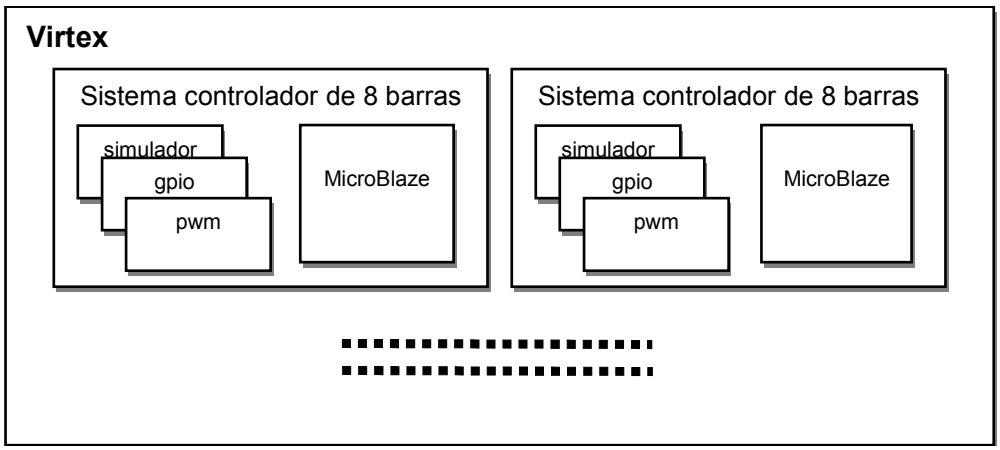


Figura 15: Esquema explicativo del diseño.

Para un mejor control del sistema y permitir al diseñado simular el movimiento real de la barras, el funcionamiento del microcontrolador se puede controlar mediante una aplicación java que se comunica con el sistema mediante el puerto serie. La aplicación java permite comunicarse con el sistema de simulación, controlando el arranque del sistema, la posición en la que se situarán las barras, y mostrando una simulación visual del moviendo del sistema.

4.2 Descripción del bloque de control básico del sistema

A continuación se muestra la estructura básica del sistema para controlar ocho barras. El sistema estará formado por un microcontrolador del tipo MicroBlaze de Xilinx. Al microprocesador se conectaran los periféricos que a continuación se describen:

Periférico de 8 *pwm*: Este periférico tiene la capacidad de generar hasta ocho señales moduladas en ancho de banda (señales *pwm*). De este modo, se podrán generar las ocho señales necesarias para mover las ocho barras que serán controladas por cada

microprocesador. Este módulo se conectará con el microcontrolador mediante el bus OPB. El microprocesador determinará los parámetros de tiempo que debe cumplir la señal *pwm* y la dirección de movimiento. Estos datos se transmitirán mediante el bus OPB de Xilinx.

Periférico simulador del sistema de barras: debido a que no se dispone del sistema real de barras para poder comprobar el funcionamiento del sistema, es necesario diseñar un módulo que permita hacer una representación aproximada del comportamiento del sistema de barras real. Para ello se diseña un módulo de simulación del movimiento de una barra, y se incluirán ocho de ellos dentro del periférico de simulación.

Este módulo de simulación de las barras recibe tan solo las señales que recibirá el piezoeléctrico encargado del movimiento de las barras (las señales *pwm* y la señal de indicación de dirección). Sin embargo, a pesar de que en principio con estas señales debería de ser suficiente para realizar una simulación del movimiento, hay que considerar el tope existente al final de cada barra. Se considera que cuando se llega al tope, la barra deja de moverse. Además, hay que considerar el hecho de que los periféricos de simulación permiten obtener una respuesta casi instantánea del resultado del movimiento, pero en la realidad, el movimiento conlleva un retardo. Por tanto, se necesitará para realizar una simulación veraz del sistema incluir una señal de retardo en este periférico.

Para poder controlar estos aspectos de la simulación, este periférico se comunicará con el microcontrolador, del cual obtendrá la información de inicialización del sistema. Estos datos de inicialización consisten en:

- Vuelta y posición de la barra en el momento de arrancar el sistema. A partir de esta información se irá controlando el número vuelta y posición de movimiento de la barra e se impedirá que se actualice el desplazamiento producido por la señal *pwm* cuando la barra se encuentre en el extremo final del sistema.
- Valor de retardo, expresado en pulsos de reloj.

Hay que tener en cuenta, que si estos datos no se inicializan el periférico considerará que las barras se encuentran en el extremo (ya situadas en el tope del sistema físico en el que están insertadas las barras) y que no se añade ningún retardo al sistema.

La salida de mi periférico de simulación será la posición absoluta entre dos perforaciones de la barra.

Periférico GPIO: se implementará un módulo GPIO (general purpose input/output) que permita la comunicación entre los sensores del sistema de barras y el microcontrolador. Hay que destacar que aunque el periférico de simulación ya permite leer el microcontrolador el resultado de la posición de las barras simuladas, este no sería le montaje definitivo. En lugar del periférico estará el sistema de barras, y para poder realizar la lectura de los sensores es necesario incluir un sistema GPIO que permita la comunicación entre el microcontrolador y el sistema de barras.

Además de estos periféricos, el sistema contendrá un IP *timer* de MicroBlaze, que permitirá tomar medidas de tiempo sobre el tiempo de ejecución del sistema implementado [32].

Estos periféricos estarán conectados al microprocesador del sistema. Al ser un sistema de lazo cerrado, el microprocesador analizará los resultados que obtiene del periférico de simulación y mediante el procesado de esta información determinara el *pwm* que actuará desplazando las barras hacia la posición determinada.

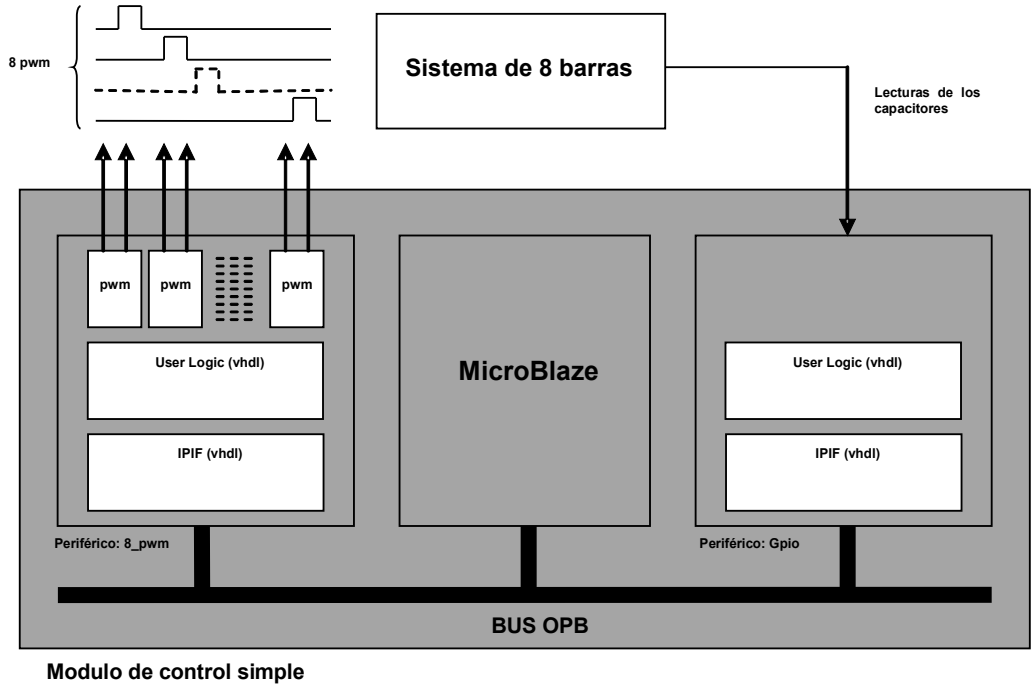


Figura 16: Descripción de la estructura de un módulo de movimiento sin el bloque de simulación

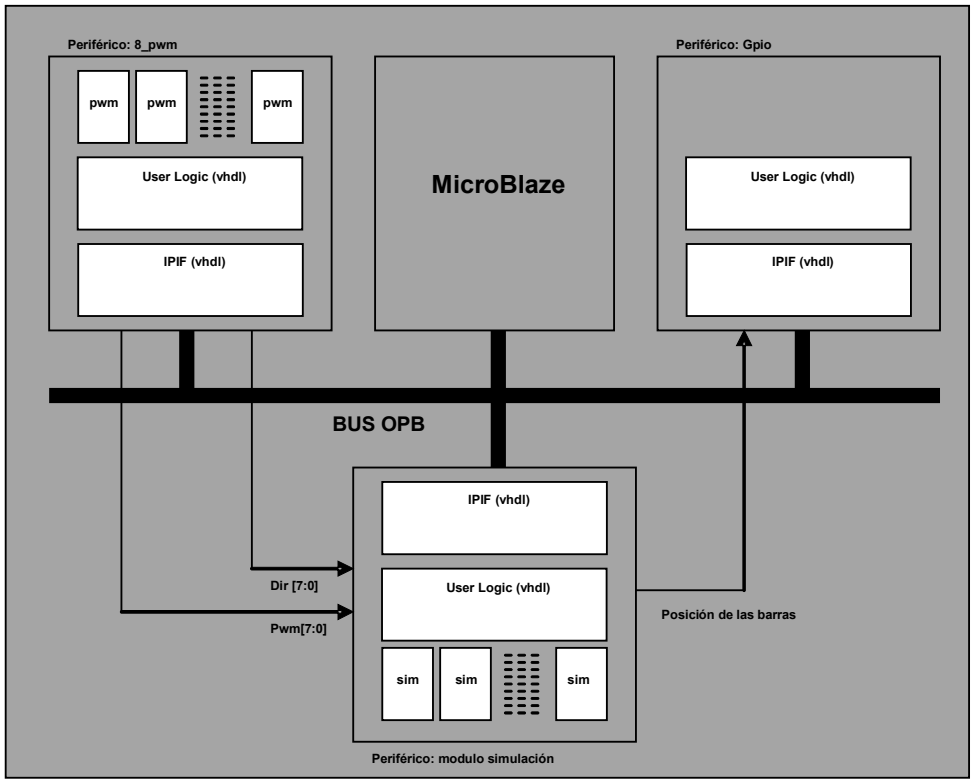


Figura 17: Descripción de la estructura de un módulo de movimiento con el bloque de simulación

4.3 Descripción de los periféricos

A continuación se hace una descripción técnica de la funcionalidad de los periféricos y su estructura. La siguiente figura muestra el flujo de información intercambiado por cada uno de los elementos. Hay que recalcar que parte de esta información se transmite mediante la utilización del bus OPB (aquellas señales que comunican el microcontrolador y los periféricos), mientras que otras lo hacen mediante rutado directo de la señal.

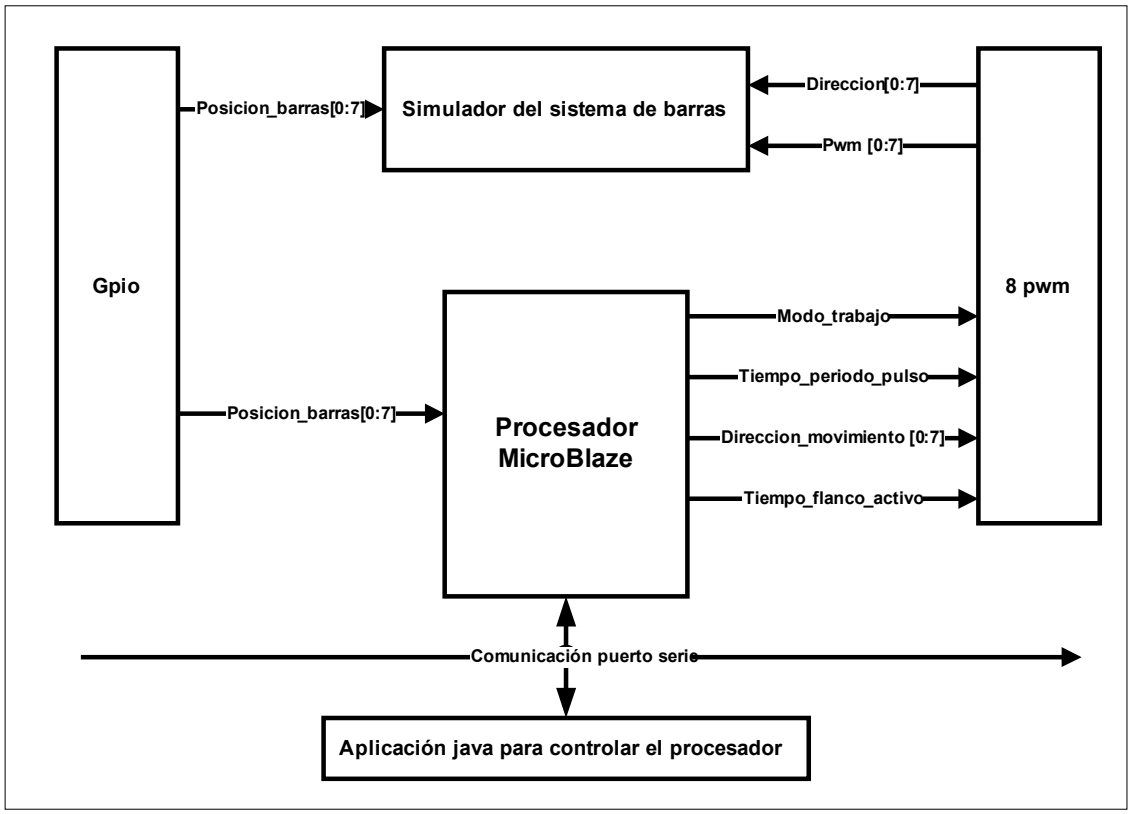


Figura 18: Descripción de la estructura de un módulo de movimiento para 8 barras.

4.3.1 Módulo PWM: Opb_8pwm

4.3.1.1 Introducción

Este periférico ha sido diseñado para controlar ocho piezoeléctricos. Cada uno de estos piezoeléctricos necesita para describir su movimiento dos parámetros:

- Una señal que determina la dirección del movimiento
- Una señal *pwm* (*pulse width modulation*) que activa el movimiento. Esta señal puede que ser continua en el tiempo, o que sea tan solo un pulso, por lo que este sistema incluye dos modos de trabajo que permitirán ajustar el movimiento del piezoeléctrico según las necesidades del sistema.

Como resultado se obtienen ocho señales *pwm* y ocho direcciones de movimiento que serán conectadas con los respectivos piezoeléctricos.

4.3.1.2 Descripción general

Este periférico se comunica con el microcontrolador utilizando el bus OPB. Para su descripción se utiliza el módulo IPIF del bus OPB, que describe un *interfaz* entre el bus y el módulo periférico. A su vez, este módulo incluye un fichero VHDL denominado “User Logic”, a partir del cual se puede capturar los datos enviados por el bus OPB.

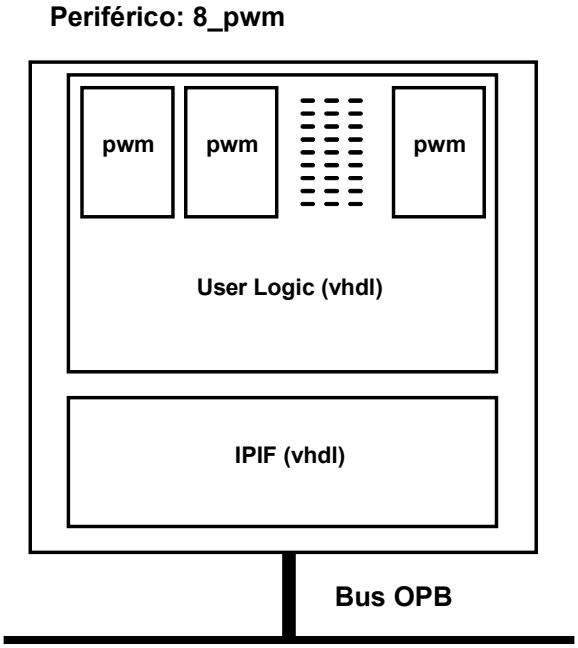


Figura 19: diagrama de bloques del periférico 8_pwm

El sistema está conformado por ocho módulos *pwm* independientes que serán instanciados dentro de módulo "User Logic" y que comparten una misma base de tiempos. Cada *pwm* recibe además de los tiempos de duración de los pulsos y los parámetros de control del sistema, el tipo de modo de trabajo asignado de manera independiente.

Tiene como salidas externa las señales:

- pwm_out (8 bits)
- direccion_out(8 bits)

Correspondientes a las direcciones y señales *pwm* obtenidas para cada una de los módulos simples.

4.3.1.3 Descripción funcional del módulo simple

El módulo pwm simple permite generar una señal *pwm* en función de la base de tiempos indicada mediante la señal de reloj del sistema. El módulo recibe como entradas:

- **clk :** señal de reloj del sistema y en función de la cual se expresarán los parámetros de tiempo del *pwm*.
- **rst :** señal de reset asíncrono interna del módulo *pwm***.**
- **capData :** señal de captura de los parámetros de tiempo de entrada del *pwm*.
- **ce :** señal de modo de trabajo.
- **disp:** señal de disparo. Determinará cuando comienza el funcionamiento del *pwm* en el modo sencillo.

- **pwm_data:** parámetro de entrada que indica el número de pulsos de reloj que la señal *pwm* está activo alto.
- **pwm_length:** parámetros de entrada que indica el número de pulsos de reloj del periodo de la señal *pwm*.

Las salidas del módulo serán:

- **disp_ack:** señal de salida indicando que ha llegado el disparo.
- **pwm_out:** señal *pwm* obtenida por el módulo.

NOMBRE	TIPO	MODO	TAMAÑO	DESCRIPCION
clk	I	--	1	Señal de reloj del sistema
rst	I	Alto	1	Señal de reset del sistema
capData	I	Alto	1	Señal de captura
ce	I	Alto	1	Señal de modo de trabajo.
disp	I	Alto	1	Señal de disparo.
pwm_data	I	--	24	Periodo de activo alto
pwm_length	I	--	32	Periodo del pulso
disp_ack	O	Alto	1	Señal disparo de salida
pwm_out	O	Alto	1	Señal *pwm* obtenida.

Tabla 6: **Pin description *del módulo pwm***

Tanto "**pwm_data**" como "**pwm_length**" expresan parámetros temporales. Como no se pueden utilizar unidades temporales directamente, estos datos vendrán parametrizados en función del periodo de reloj, es decir, cada uno de ellos determinará el tiempo como un número de periodos de reloj.

El módulo simple *pwm* puede trabajar en dos modos de trabajo:

- **modo sencillo:** Se obtiene un único pulso activo del *pwm*. Cuando el sistema está en modo sencillo de trabajo la señal de entrada ce debe estar activo bajo (ce='0').
- **modo continuo:** Se obtiene a la salida una señal *pwm* continua**.** Cuando el sistema está en modo continuo de trabajo la señal de entrada ce debe estar activo alto (ce='1').

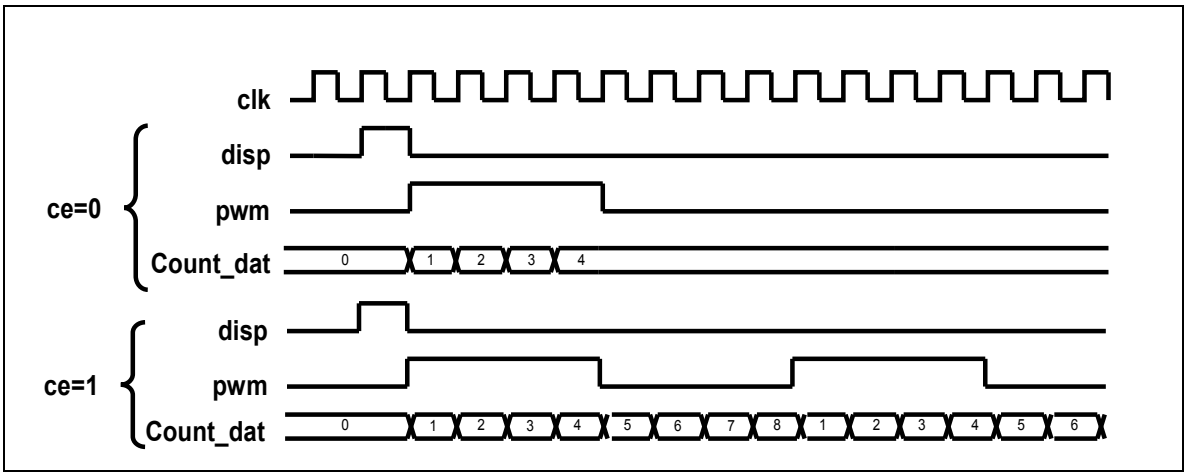


***Figura 20: descripción del funcionamiento del módulo* pwm.**

Este sistema no dispone de sistema de parada ni de arranque, por tanto, siempre estará activo. Para conseguir que no produzca una señal *pwm* el parámetro de tiempo que determina el tiempo que debe estar en activo el *pwm* debe ser nulo. El módulo dispone de un reset asíncrono que inicializa los parámetros de tiempo del sistema a nulo e impide el funcionamiento del mismo.

Inicialmente, cuando el microprocesador se arranque, se ejecutará el reset del sistema. Desde ese momento, el módulo se mantendrá parado hasta que los parámetros tengan nuevos valores.

Para la inicialización del módulo se deben de inicializar los parámetros de tiempo deseados para la señal *pwm*. Estos parámetros vendrán expresados en función de la señal de reloj de entrada del módulo. También será necesario indicar el modo de trabajo deseado mediante la señal de entrada **"ce"**. Cuando la señal de entrada **"capData"** se active, los parámetros de tiempo y control del módulo se cargarán y estarán activos hasta que otra señal **"CapData"** llegue o se active la señal de "**reset**" del módulo.

El módulo comienza su funcionamiento cuando detecta la llegada de un disparo. En ese momento el contador del módulo comenzará a contar. El sistema dispone de dos comparadores. El primero de ellos compara la señal del contador con el parámetro de periodo del *pwm* (la señal **"pwm_length_init"**) de modo que cuando ambas sean iguales se reseteé el contador. El segundo comparador compara la salida del contador con la señal **"pwm_data_init"**, de modo que si la señal es menor, la salida del *pwm* estará activa, mientras que si es mayor, la salida del pwm se mantendrá inactiva.

El código detallado está incluido en el apéndice I.1.

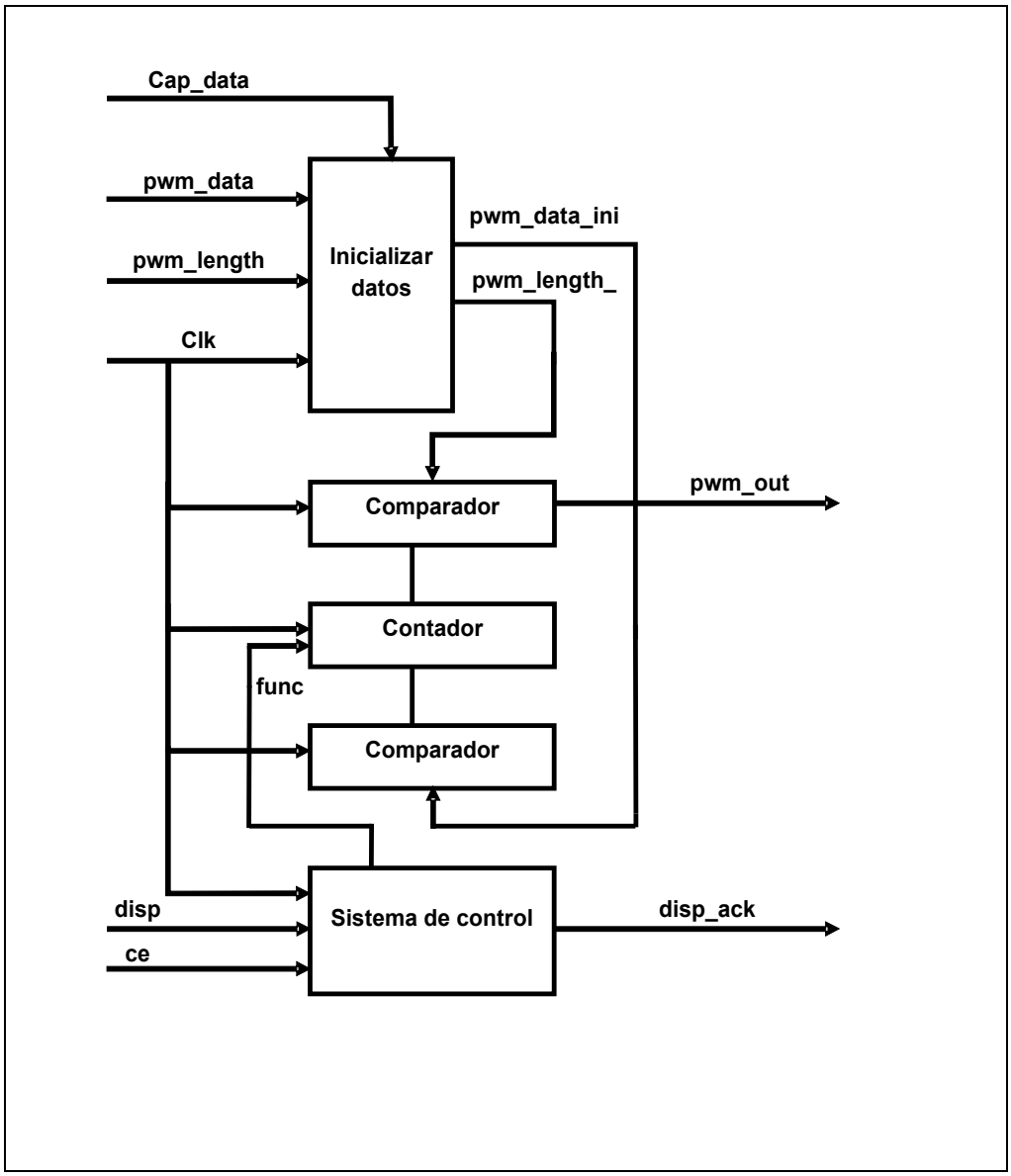


Figura 21: descripción modular del módulo pwm.

La diferencia de funcionamiento debido al modo de trabajo reside en que cuando el módulo está programado en modo sencillo, al detectar que el comparador de periodo de señal se activa, el contador se para y deja de contar. Por lo tanto, tan solo se habrá activado la señal *pwm* durante un periodo. En el modo continuo el contador tan solo parará cuando se vuelva a detectar una carga de datos o un reset.

4.3.1.4 Interfaz

A continuación se describe la estructura de los registros de comunicación con el periférico. Cada uno de los registros definidos está formado por 32 bits.

Registro de habilitación

- Dirección : OPB_8PWM BASEADDR

- Registro solo de escritura

---	---	---	---	CE	DISP

- CE(8 bits): señal de activación de modo de cada *pwm*.
- DISP(8 bits): señal de disparo de cada *pwm*.

Registro de periodo activo del pwm

- Dirección : OPB_8PWM BASEADDR + 0x00000004
- Registro de escritura

CAP	FLANCO

- CAP (8 bits): señal “capData” o de activación de captura de datos para los 8 *pwm* del periférico.
- FLANCO (24 bits): número de pulsos de reloj que debe estar activa la salida del *pwm*. En este caso, cuando active la señal de captación de datos se tomara el flanco indicado. De esta manera se puede cargar a la vez en todos los módulos simples de *pwm* el mismo flanco, por el contrario, cargar un diferente.

Registro de periodo total del pwm

- Dirección : OPB_8PWM BASEADDR + 0x00000008
- Registro de escritura

PERIODO TOTAL

- PERIODO TOTAL (32 bits): número de pulsos de reloj que dura el periodo de un pulso *pwm*. Este dato será común para los 8 *pwm* generados por el periférico.

Registro de direcciones.

- Dirección : OPB_8PWM BASEADDR + 0x00000012
- Registro de escritura

---	---	---	---	---	---	DIR

- DIR (8 bits): Indica las posiciones de movimiento de los 8 pwm.

4.3.2 Módulo de simulación: Opb_simular

4.3.2.1 Introducción

La creación de este periférico surge de la necesidad de poder hacer una simulación del movimiento de las barras. Con este periférico se consigue hacer una aproximación del movimiento de ocho barras, teniendo como entrada las señales de dirección y *pwm* generadas por el módulo *pwm*.

El periférico también recibe información de microcontrolador, a fin de poder inicializar variables del sistema, como el retardo al que está sometido el movimiento de las barras.

4.3.2.2 Descripción del periférico

El periférico está formado por ocho módulos de simulación simple, los cuales simulan de manera independiente el movimiento que realizaría una barra del sistema. El sistema comparte una misma base de tiempo y todos los módulos tienen el mismo retardo.

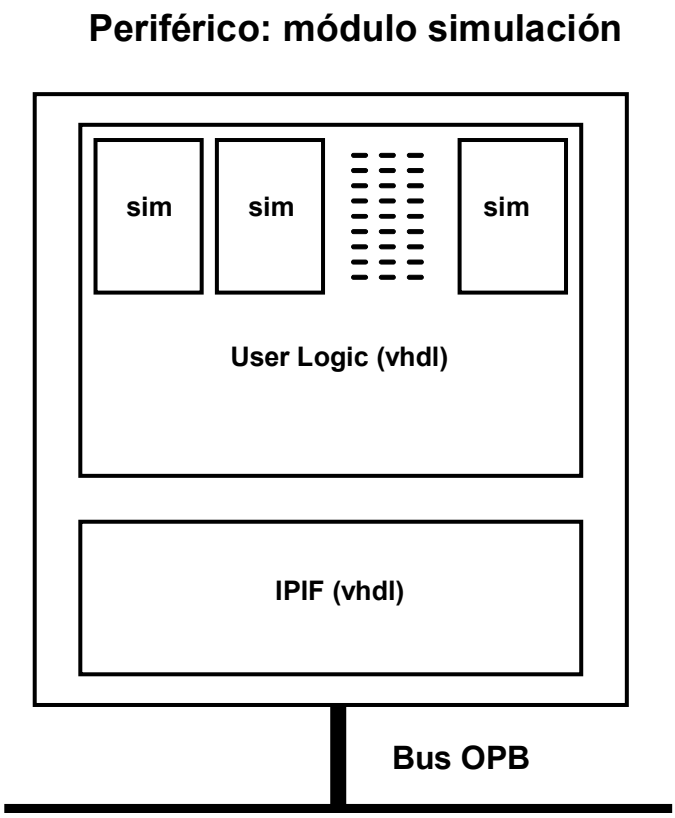


Figura 22: diagrama de bloques del periférico de simulación

Al igual que en periférico 8pwm, este módulo necesita conectarse mediante le bus OPB al microcontrolador. Para ello utiliza de nuevo el módulo IPIF como interfaz y el módulo “User Logic” para instanciar los módulos simples de simulación.

4.3.2.3 Descripción del módulo simple de simulación

El módulo de simulación simple permite simular el movimiento de una barra. Como señales de entrada recibe:

clk : señal de reloj del sistema. Todas las medidas de tiempo vendrán en función del periodo de la señal de reloj.

rst: señal de reste del sistema. Activa un reset asíncrono.

cap: señal de captura de datos de entrada.

pwm: señal de *pwm* que afecta a la barra simulada.

dir : señal de dirección de movimiento de la barra simulada.

retardo: retardo al que está sometido en sistema en su movimiento. Este parámetro vendrá expresado en número de pulsos de reloj del sistema.

vuelta: número de vuelta inicial en la que se encuentra el sistema antes de comenzar su simulación.

posicion_inicial: posición inicial dentro de una vuelta en la que se encuentra el sistema antes de comenzar su simulación.

Las salidas del módulo serán:

dat: posición dentro de la vuelta en la que se encuentra la barra.

cap_aux : Señal auxiliar de desactivación de captura de datos.

NOMBRE	TIPO	MODO	TAMAÑO	DESCRIPCION
clk	I	--	1	Señal de reloj del sistema
rst	I	Alto	1	Señal de reset asíncrona.
cap	I	Alto	1	Señal de captura de los parámetros iníciales.
pwm	I	Alto	1	Señal *pwm.*
dir	I	Alto	1	Señal de dirección.
retardo	I	--	8	Tiempo de retardo.
vuelta	I	--	8	Número de vuelta inicial.
posición_inicial	I	--	32	Posición inicial dentro de una vuelta.
dat	O	--	32	Posición dentro de la vuelta en la que se encuentra la barra.

cap_aux	O	Bajo	1	Señal auxiliar de desactivación de captura de datos

Tabla 7: **Pin description** ***del módulo de simulación***

El sistema dispone de un reset asíncrono, el cual pone los contadores del sistema a cero y además, inicializa la posición de las barras a la posición inicial.

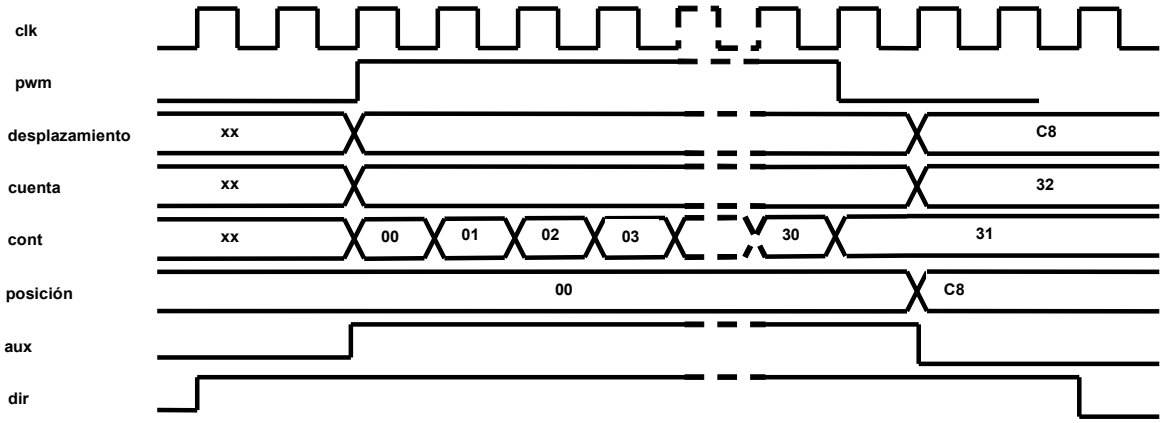


Figura 23: descripción del funcionamiento del módulo de simulación.

El sistema se inicializa al activarse la señal "**cap**", cargando los valores de vuelta y posición inicial en los registros internos del sistema. Desde este instante, el sistema considerara esos valores la posición de inicio de movimiento y solo se modificarán si vuelve a activarse la señal de captura de datos ("**cap**"). En caso de que el sistema no esté inicializado, las barras se consideraran inicializadas a la posición de reset. No es indispensable la inicialización del sistema, pero es adecuado para ajustarse a la realidad.

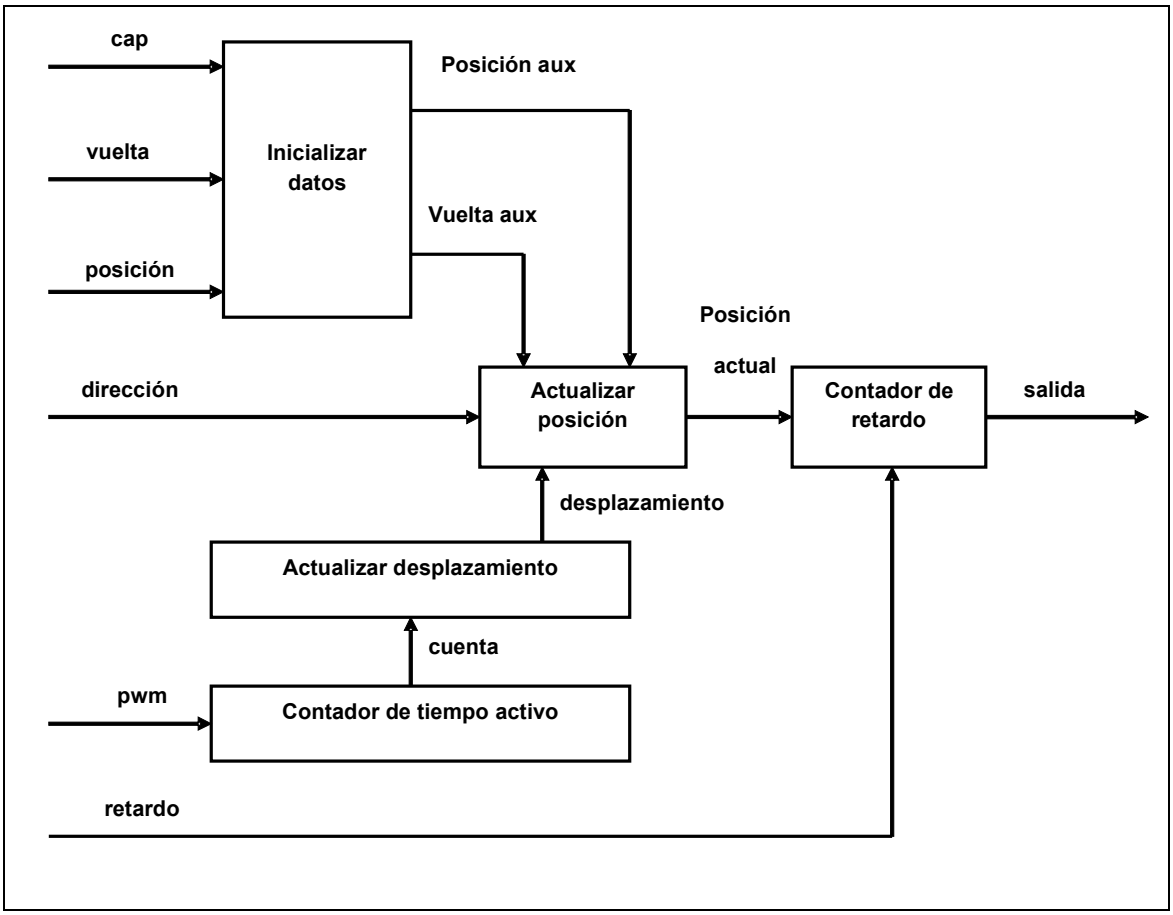


Figura 24: Esquemático del módulo simulación

El sistema se simulación contabiliza el periodo de tiempo que la señal pwm se encuentra activa y en función del mismo determina el desplazamiento que se habría producido en la placa. De esta manera actualiza la posición de las barras y el número de vuelta en el que se encuentra. Hay que tener en cuenta que el simulador solo nos dará como salida la posición dentro de la vuelta, sin embargo, es importante conocer el número de vuelta, ya que cuando la barra llegue al extremo, se encontrará con el tope. Por tanto, cuando la barra llegue a la vuelta cero y se encuentre en la última posición la actualización de la posición de la barra se deberá parar.

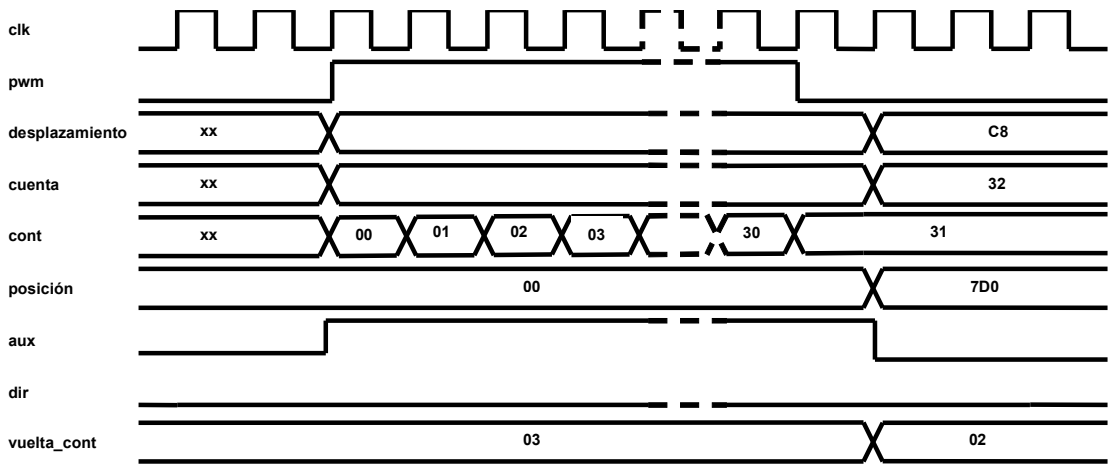


Figura 25: descripción del control de vueltas en el módulo de simulación.

En caso de que exista retardo, el sistema tardara tantos periodos de reloj como se indique en la señal retardo en actualizar la posición de las barras en la salida del módulo de simulación.

El código detallado está incluido en el apéndice I.2.

4.3.2.4 Interfaz

A continuación se describe la estructura de los registros de comunicación con el periférico. Cada uno de los registros definidos está formado por 32 bits

Registro de retardo.

- Dirección : OPB_SIMULAR BASEADDR
- Registro de escritura

RETARDO	---	---	---	---	---	---

- RETARDO (8 bits): valor del número de pulsos de reloj de retardo ha los que se ve sometido la salida de la posición actual de la barra simulada. El valor del retardo es común para las ocho barras simuladas
-

Registros de posición precisa.

- Existen 8 registros de posición precisa (tantos como barras simula el sistema) comenzando la numeración de las mismas por el valor '1'.
- Dirección : OPB_SIMULAR BASEADDR + 4 * (número de barra)
- Registros de escritura

POSICION INICIAL

- POSICION INICIAL (32 bits); parámetro que indica la posición inicial en la que se encuentra la barra cuando comienza la simulación.

Registros de posición actual.

- Existen 8 registros de posición actual (tantos como barras simula el sistema) comenzando la numeración de las mismas por el valor '1'.
- Dirección : OPB_SIMULAR BASEADDR + 28+4 * (número de barra)
- Registros de lectura

POSICION ACTUAL

- POSICION ACTUAL (32 bits); devuelve la posición actual de la barra.

Registros de número de vueltas y captura.

- Existen 8 registros número de vuelta y captura (tantos como barras simula el sistema) comenzando la numeración de las mismas por el valor '1'.
- Dirección : OPB_SIMULAR BASEADDR + 68 + 4* (número de barra)
- Registros de escritura

CAP	-	-	-	---	---	---	---	---	VUELTA

- CAP (1 bits: bit más significativo): señal de captura de los datos iníciales del sistema de barras.
- VUELTA (8 bits): parámetro que indica la vuelta inicial en la que se encuentra el sistema.

4.3.3 GPIO: Opb_my_gpio

4.3.3.1 Introducción

Se implementa un módulo propio GPIO (general purpose input/output) que servirá de puente entre los sensores del sistema de barras y los pines de entrada al microprocesador.

Hay que considerar que aunque el periférico de simulación ya permite leer al microcontrolador el resultado de la posición de las barras simuladas, este no sería posible en el montaje definitivo. El lugar del periférico estará el sistema de barras, y para poder leer los sensores es adecuado incluir un sistema GPIO que permita la comunicación entre el microcontrolador y el sistema de barras.

4.3.3.2 Descripción del periférico

Los módulos GPIO son módulos asíncronos muy extendidos en los sistemas embebidos. Permiten proporcionar al sistema un conjunto de entras salidas configurables por el mismo sistema.

En este caso el módulo está diseñado como entradas salidas de 32bits.

4.3.3.3 Interfaz

Registros de lectura de posición.

- Existen 8 registros de lectura de posición comenzando la numeración por el valor '1'.
- Dirección : OPB_MY_GPIO BASEADDR + 4 * (número de barra)
- Registros de escritura

POSICION BARRA n

- POSICION BARRA N (32 bits): salida del banco del banco de registros

4.3.4 Instanciación de los periféricos en el sistema

Como ya se describió brevemente en el apartado 2.4.2, la herramienta EDK dispone de una aplicación que permite diseñar periféricos [12]. Mediante esta herramienta se puede diseñar un periférico para MicroBlaze, que disponga del interfaz de conexión con el bus OPB. De esta manera, los periféricos diseñados pasan a formar parte del catalogo de IP disponibles para ser instanciados en el procesador.

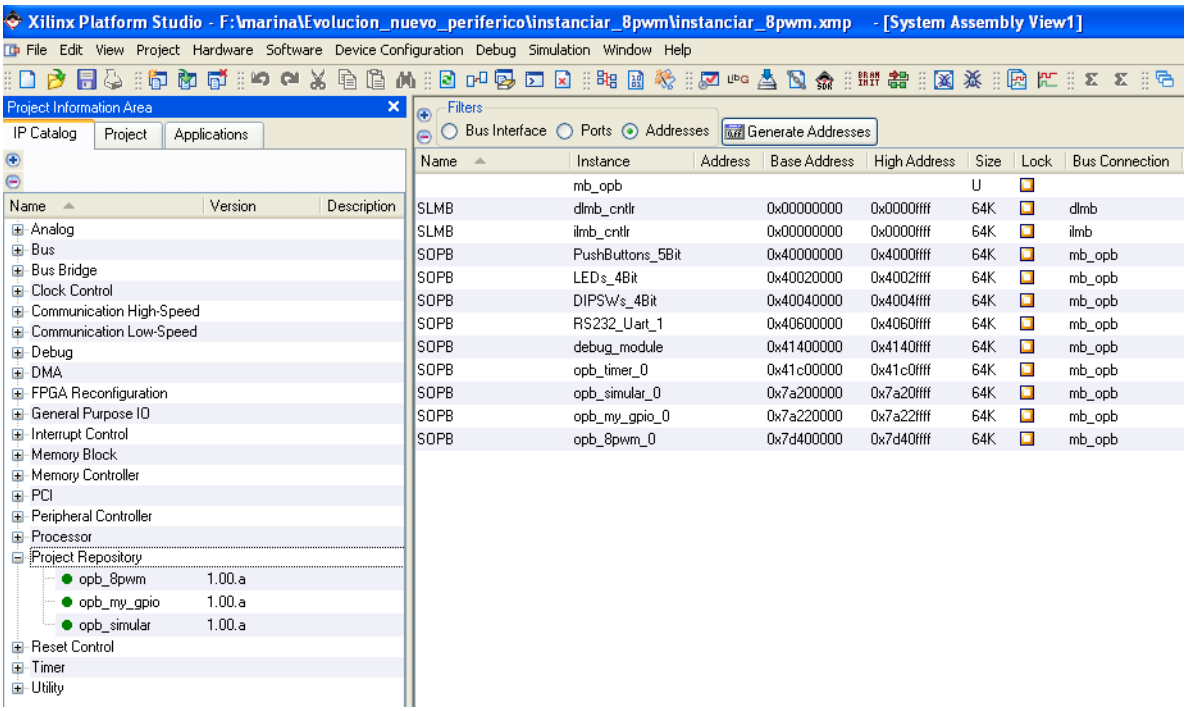


Figura 26: Catálogo de IP disponibles para ser instanciados en el procesador.

En la Figura 26 se puede ver todo el catalogo de IPs del sistema, organizadas según su funcionalidad. En el caso de los periféricos diseñados por el propio usuario, estos se encuentran en el '*proyect repository*'.

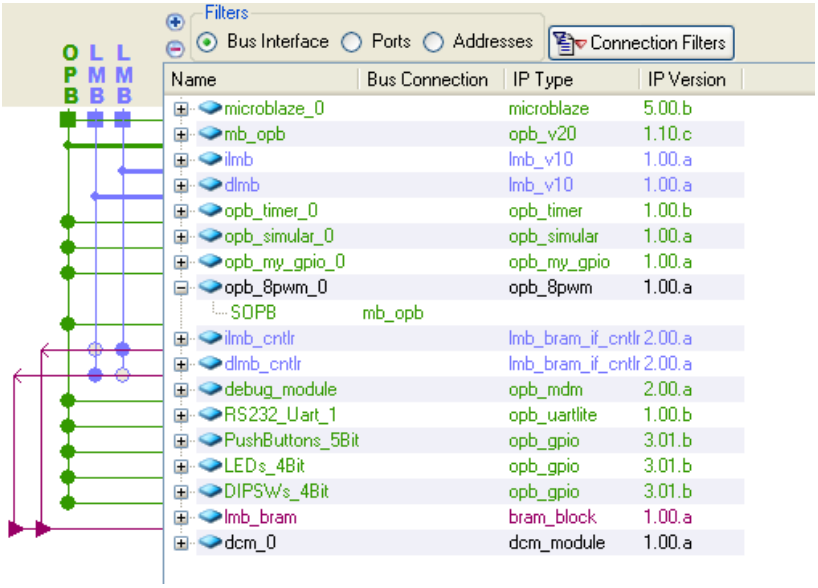


Figura 27: Conexiones de los IPs a los buses del microprocesador.

Cuando se instancia uno de estos IPs, se indica el bus al cual se desea conectar. Además del interfaz con el bus, el periférico puede disponer de señales externas, que se comuniquen con otros periféricos, sin necesidad de pasar por el interfaz del procesador. En el caso de este diseño, es necesario interconectar los periféricos como se indicaba en la Figura 27. Como se puede observar en la Figura 28, las puertas de los periféricos se pueden conectar directamente con las de otros periféricos con tan indicar el nombre de la señal interna con la que se desea hacerlo.

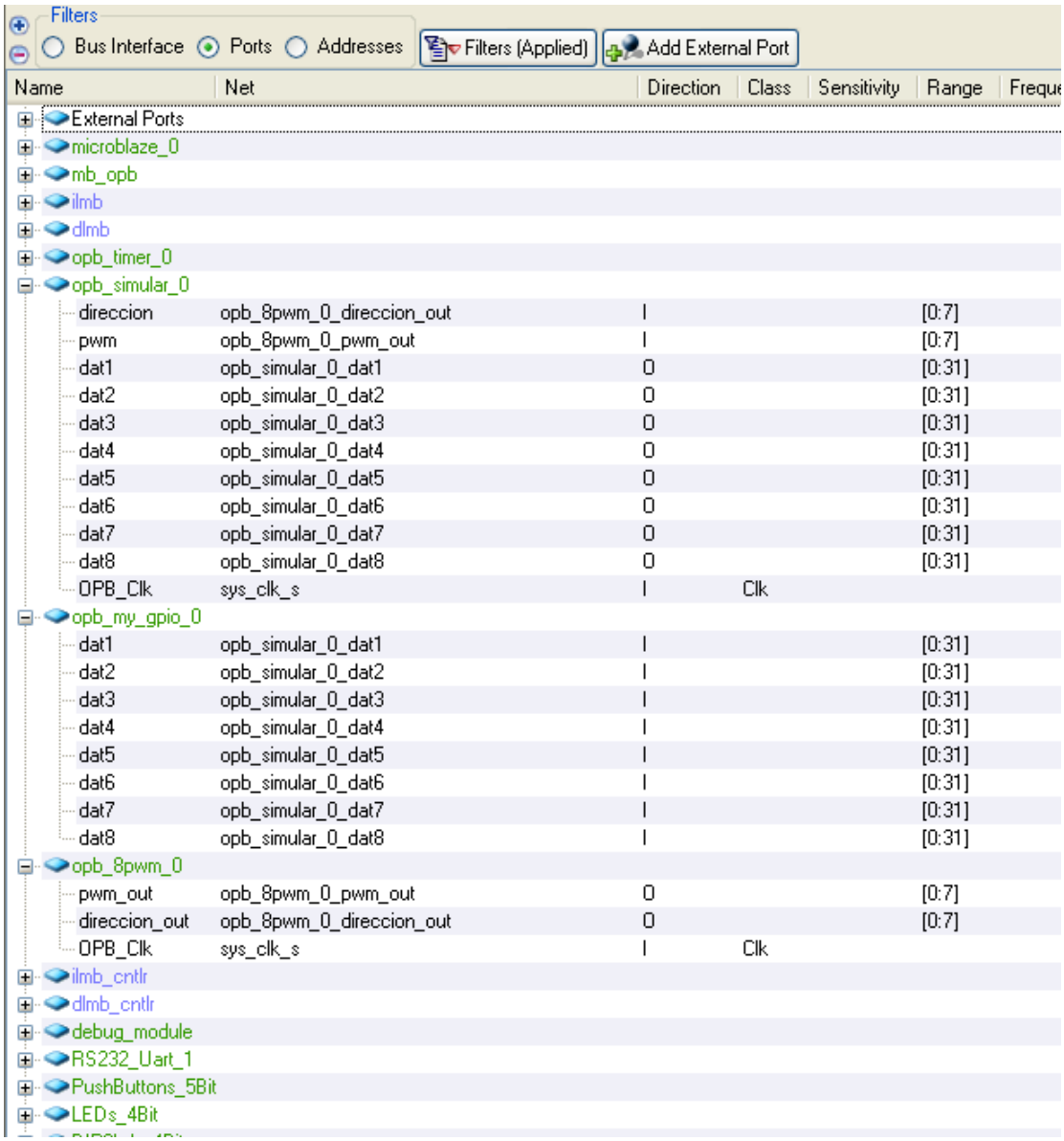


Figura 28: Conexiones de señales externas de periféricos.

Estas conexiones realizadas mediante la herramienta son descritas en el fichero .mhs (*microprocessor hardware specification*) junto con todos los detalles de instanciación de los periféricos incluidos en el procesador. Además de este fichero, existen otros ficheros que describirán nuestro sistema hardware, como se puede observar en la Figura 29. Estos ficheros están explicados con mayor detalle en el apéndice B

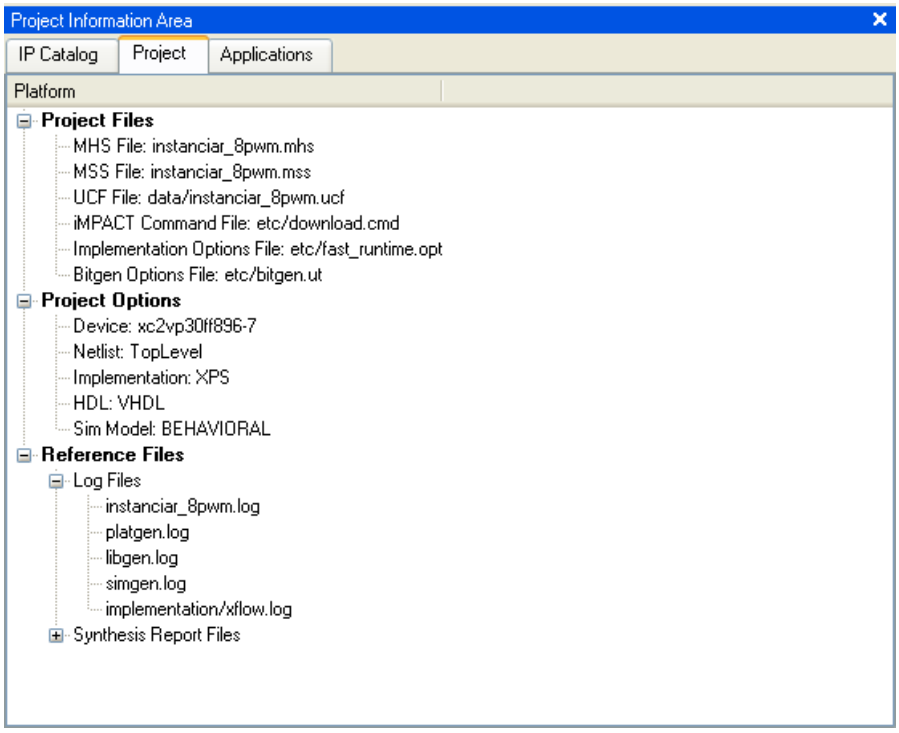


Figura 29: Ficheros hardware del proyecto.

4.4 Descripción del código software

En función del diseño de un bloque básico del sistema, se diseña un software de programación para el control de 8 barras del sistema. También se podrá utilizar para ello la herramienta de diseño EDK de Xilinx.

4.4.1 Instanciación de nuevo proyecto en el diseño

Se crea un proyecto de aplicación nuevo en el procesador. La herramienta genera automáticamente una serie de ficheros y carpetas que permitirán instanciar los ficheros de en código C en el procesador. El más destacado será el mapa de direcciones de memoria de los periféricos conectados al procesador, que se encuentra descrito en el fichero 'xparameter.h'.

Como se observa en la Figura 30, al generar una nueva aplicación aparece una nueva pestaña en la cual se indica la posición en la cual se deben incluir los ficheros fuente y los ficheros cabecera.

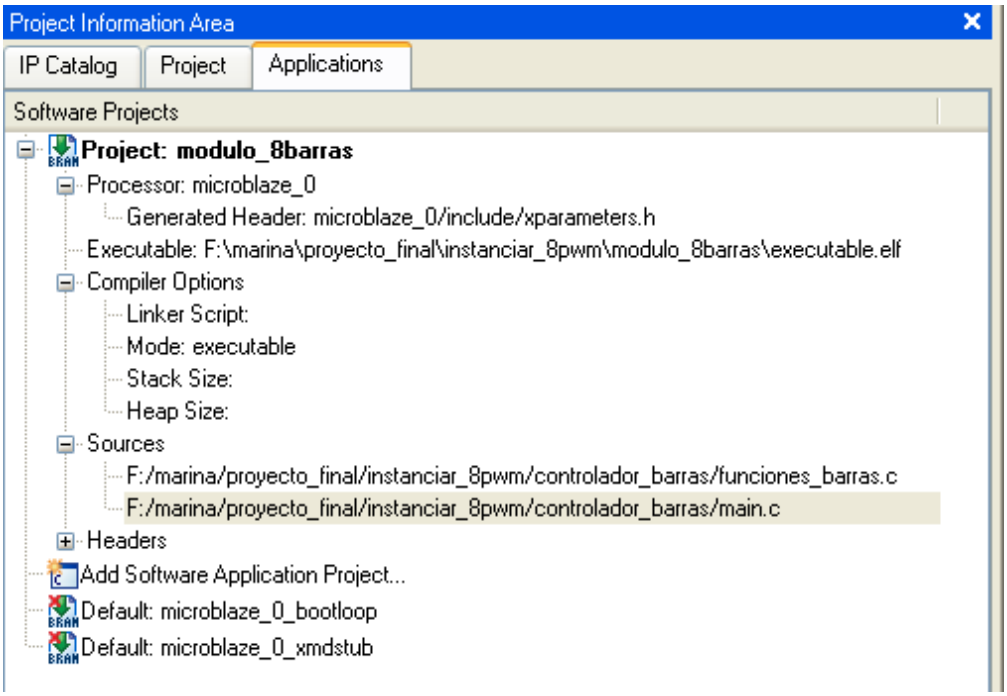


Figura 30: Descripción de la estructura de una aplicación sobre MicroBlaze

4.4.2 Algoritmo de control del sistema

A continuación se describe el algoritmo de control utilizado para controlar y actualizar el movimiento de las 8 barras.

Como se ha comentado anteriormente, las barras contienen una serie de rendijas equidistantes. Mediante la utilización de 4 capacidades y midiendo la tensión existente en ellas se puede conocer la posición exacta entre dos de los huecos de la barra. Sin embargo, será necesario que el software mantenga la cuenta de las veces que la barra avanza entre dos rendijas, ya que esta no se puede conocer mediante la medida de las capacidades (*encoder* relativo). Esto tendrá dos implicaciones: será necesario resetear siempre que aparezca un error en el sistema y cuando se reinicie el sistema para poder asegurarnos que el sistema se encuentra en la posición cero.

Para cada barra se define una estructura básica que permite el manejo sencillo de los datos asociados a cada barra.

```
typedef struct
{
      Xuint16 precision;
      Xuint8 vuelta;
      } Posicion_barra;
```

La precisión representa la posición de la barra entre dos de las rendijas de medida. Este dato se obtiene mediante la medida directa de la tensión de las capacidades.

La variable vuelta permite mantener la cuenta del número de pasos por diferentes rendijas de medida se han realizado. A diferencia de la variable precisión, esta medida solo se puede inicializar mediante un reset y siempre tomará el valor cero. Posteriormente, será actualizando en función del valor de precisión leída por el sistema.

Se diseñan una serie de funciones para el control y manejo de cada barra.

```
int actualizarIzquierda(Posicion_barra *,int,Xuint32);
int actualizarDerecha(Posicion_barra *,int,Xuint32);
int control(Posicion_barra *,Posicion_barra *,Xuint32 *,int *);
int reset(Posicion_barra*);
```

La función “**actualizarIzquierda**” permite actualizar la posición de una barra situada a la izquierda. Las variables a tener en cuenta para la actualización es la dirección de movimiento y la precisión leída por el sistema (en nuestro caso, la posición la obtendrá del periférico de simulación).

La función “**actualizarDerecha**” permite actualizar la posición de una barra situada a la derecha. El funcionamiento de la función es idéntico al de la función “actualizarIzquierda” con la variación de la variable dirección.

La función “**control**” permite determinar el siguiente movimiento a realizar por la barra para llegar a la posición ideal. La función recibe como parámetros la posición actual de la barra y la posición ideal a la que debe llegar. Comparando estos parámetros se determina la dirección de movimiento que se enviara la periférico pwm y si el movimiento será burdo o más preciso. Siempre que la barra se encuentre en un número de vuelta diferente a la ideal, el sistema determinara que el movimiento debe ser burdo (el actuador esta en movimiento más tiempo), mientras que siempre que se mueva dentro de la vuelta asignada el movimiento será corto. El sistema no se moverá cuando se detecta que la posición de la barra está a menos distancia de la precisión asignada.

La función “**reset**” permite resetear la posición de las barras. Esta función se debe ejecutar siempre que se reinicie el sistema, ya que es la función que nos permite inicializar el valor del número de vuelta de cada barra (ya que esta información no se puede obtener directamente de la lectura del sistema).

La función de reset utiliza el modo continuo del *pwm*, y activa al mismo tiempo el movimiento de las ocho barras hasta sus extremos. La función finaliza de resetear después de varias actualizaciones que las barras se encuentran dentro de un rango de precisión similar.

Debido a la utilización del módulo de simulación, será necesario inicializar los parámetros de simulación de nuestro módulo. Se podrá inicializar la vuelta en la que se encuentra cada barra y un retardo del sistema. El retardo se expresará en función de ciclos de reloj.

Cada uno de los bloques básicos del sistema está compuesto por un microprocesador, el cual tendrá como código el algoritmo de control implementado. Este algoritmo en lazo cerrado (o sistema realimentado) determina el movimiento de las barras en función de la posición actual de cada barra. A continuación se describen los aspectos más importantes del código:

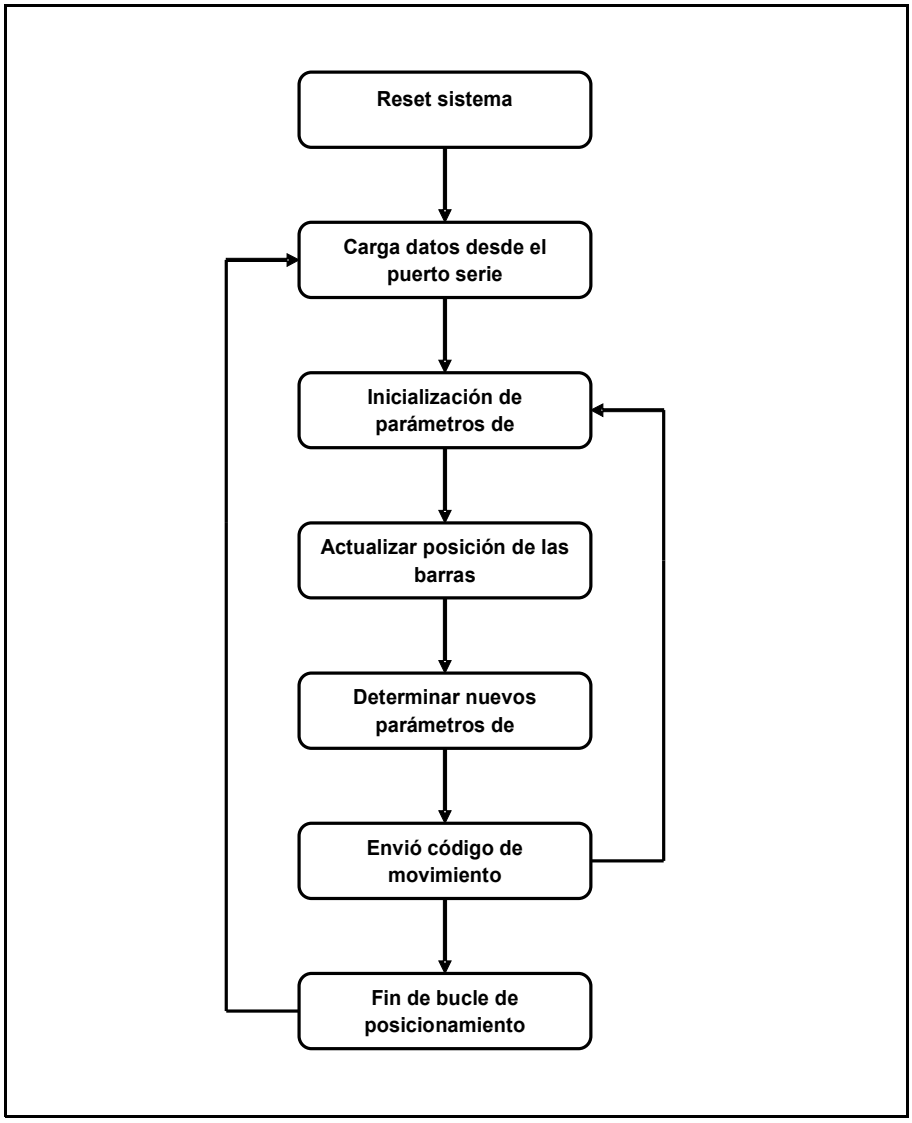


Figura 31: Flujo del programa software del microprocesador

- **ETAPA INICIAL**: esta etapa comprende la declaración de las variables del sistema, que permiten el control del mismo, y su inicialización.

  Se declaran dos tablas de tamaño fijo, igual al número de barras del sistema, de tipo "Posicion_barra". Como se explicó anteriormente, esta estructura permite almacenar la información de la posición de una barra, separando la información en dos campos, el campo "posicion" correspondiente a la información obtenida directamente de la

lectura de los sensores, y el campo “vuelta”, el cual se actualiza de manera inteligente por parte del sistema. Una de las tablas será la que contenga la posición actual de las barras, y que se irá actualizando con el tiempo, mientras que la otra tabla contendrá la posición en la que se deberán colocar las barras, y se toma como referencia para determinar la velocidad y dirección del movimiento de la barra.

Se definirán dos tablas de mascaras. La primera de la tabla se llamara “disp” y contendrá las ocho mascaras correspondientes al interfaz *pwm* correspondiente al sistema. Con objeto de recordar, la señal disparo del periférico *pwm* era la que activaba la producción de un pulso *pwm*. Además de los disparos simples, se define una constante que dispare al mismo tiempo los ocho *pwm* capaz de producir de manera simultánea el periférico *pwm*.

```
//CONSTANTES DE DISPARO

#define DISP_1              0x00000001;
#define DISP_2              0x00000002;
#define DISP_3              0x00000004;
#define DISP_4              0x00000008;
#define DISP_5              0x00000010;
#define DISP_6              0x00000020;
#define DISP_7              0x00000040;
#define DISP_8              0x00000080;
#define DISP                0x000000FF;
```

Del mismo modo que los mascaras de disparo, se generara una máscara de tipo de pwm para cada barra y otra general (movimiento continuo o limitado).

Se declaran dos tablas de tamaño fijo, igual al número de barras del sistema, del tipo “Posicion_barra”. Como se explicó anteriormente, esta estructura permite almacenar la información de la posición de una barra, separando la información en dos campos, el campo “posicion” correspondiente a la información obtenida directamente de la lectura de los sensores, y el campo “vuelta”, el cual se actualiza de manera inteligente por parte del sistema. Una de las tablas será la que contenga la posición actual de las barras, y que se irá actualizando con el tiempo, mientras que la otra tabla contendrá la posición en la que se deberán colocar las barras, y se toma como referencia para determinar la velocidad y dirección del movimiento de la barra.

- **ETAPA DE RESET**: una vez declaradas todas las variables del sistema necesarias, se debe resetear el sistema. El reset consiste en desplazar las 8 barras a los extremos de la ventana, dejando así entrar todo el espectro.

  A priori, se podría concluir que el hecho de resetear el sistema podría resultar poco eficiente, ya parece más sencillo comenzar a mover las barras desde su posición actual. Ciertamente, desplazar las barras al extremo de la ventana supondrá de media tener que realizar más pasos hasta llegar a la posición deseada, lo que supone una

pérdida de tiempo y recursos. Sin embargo, como se comento brevemente, la función reset tiene como principal objetivo permitir al sistema inicializar el campo vuelta, que define la posición de una barra. Si no se realizara un reset, sería imposible determinar la posición real de las barras, y por tanto, no se podrían asegurar el correcto funcionamiento del algoritmo.

- **ETAPA DE CARGA DE DATOS**: al igual que la posición actual de las barras, la variable que contiene la posición en la cual se deben situar las barras debe ser inicializada.

  Es el usuario del sistema el que debe de introducir cual es la posición en la cual se posicionaran las barras. Para ello se utiliza un interfaz gráfico que se comunica con el microprocesador mediante un puerto serie. Para el microprocesador la comunicación mediante un puerto serie es transparente, es decir, el sistema tan solo leerá y escribirá mediante las funciones “printf” y “gets” adaptadas para Xilinx.

- **ETAPA DE REALIMENTACION Y ACTUALIZACION DEL SISTEMA:** Esta etapa se encarga de leer la posición de las barras, procesar esta información y determinar la duración y dirección del movimiento.

  Esta etapa se caracteriza por ser un bucle que se repite tantas veces como el número de barras del sistema. A continuación se muestra la descripción del mismo en pseudo código:

```
Mientras i < número de barras
        Actualizo la posición de la barra i.
        Determino el desplazamiento a realizar por el sistema.
        Muevo la barra i.
```

  Para realizar este proceso, se utilizan las funciones definidas en ‘funciones_barras.h’.

  El sistema se moverá con movimiento burdo y continúo si la barra no se encuentra en el mismo número de vuelta que el deseado. En caso contrario, el movimiento será preciso y no continuo, hasta encontrar a una distancia menor que la precision, en cuyo caso el sistema no se moverá.

  Una vez recorrido el bucle para todas las barras del sistema, se comunica por el puerto serie la posición actual de las barras. Para ello se utiliza el siguiente código:

```
cd:b1v02p0003
```

  cd: → Cabecera del código, permite al sistema que recibe la información determinar que la siguiente información se refiere a la posición actual de una barra.

b → indica que el siguiente byte determina el número de la barra. Las barras se comienzan a numerar por el valor cero.

v → indica que los siguientes dos bytes de información se corresponden con la vuelta en la que se encuentra la barra. En este ejemplo, la barra número 1 se encuentra en la segunda vuelta.

p → indica que los siguientes cuatro bytes de información se corresponden con la posición dentro de una vuelta en la que se encuentra la barra. En este ejemplo, la barra número 1 se encuentra en la posición 0003.

- **ETAPA DE CONTROL DE FIN DE BUCLE:** esta etapa se encarga de determinar si el sistema de barras a terminado su desplazamiento, y por tanto, se puede pasar de nuevo a la etapa de carga de datos, o si por el contrario, es necesario volver a la etapa de actualización del sistema.

  El sistema considerara terminado el desplazamiento de las barras, cuando todas las barras se encuentren paradas. Si esta condición no se cumple, volverá a la etapa de actualización. En el caso de que se cumpla, volverá a la etapa de carga de datos, para que el usuario enviara la nueva posición en la que se deben colocar las barras.

El código detallado está incluido en el apéndice H.

4.4.3 Comunicación mediante el puerto serie

Xilinx provee de librerías en C optimizadas en ocupación y velocidad para sus microprocesador (tanto MicroBlaze como PowerPC). En la librería “stdio.h” para EDK, están implementadas las funciones tradicionales de entrada y salida “printf” y “scanf”. Sin embargo, también está definida la función “Xil_printf”, similar a la clásica función “printf” pero que ocupa tan solo 1kB. Sin embargo, esta función no permite imprimir funciones en punto flotante [31].

4.5 Descripción del interfaz gráfico

El interfaz gráfico de simulación de las barras ha sido descrito mediante el lenguaje java, utilizando la herramienta NetBeans IDE 5.5.1 que utiliza la versión Java 1.6.0_02.

4.5.1 Comunicación mediante el puerto serie en Java

Para la comunicación por el puerto serie de la FPGA será necesario incluir en nuestro programa el paquete del API de comunicaciones de java (package javax.comm [10]) compatible con JDK1.2. Las características de este paquete y las funciones definidas en base al mismo están detalladas en el apéndice G de este documento.

4.5.2 Descripción funcional del interfaz gráfico

Para controlar la posición la posición adecuada de las barras y permitir mostrar una representación grafica del movimiento de las barras, se crea una aplicación java que se comunica mediante un puerto serie con la FPGA. En concordancia con las características del diseño, esta aplicación ha sido diseñada para comunicarse con una de los microprocesadores embebidos en la placa, el cual controla el movimiento de ocho barras (las cuatro barras situadas a la derecha y las otras cuatro barras situadas a la izquierda).

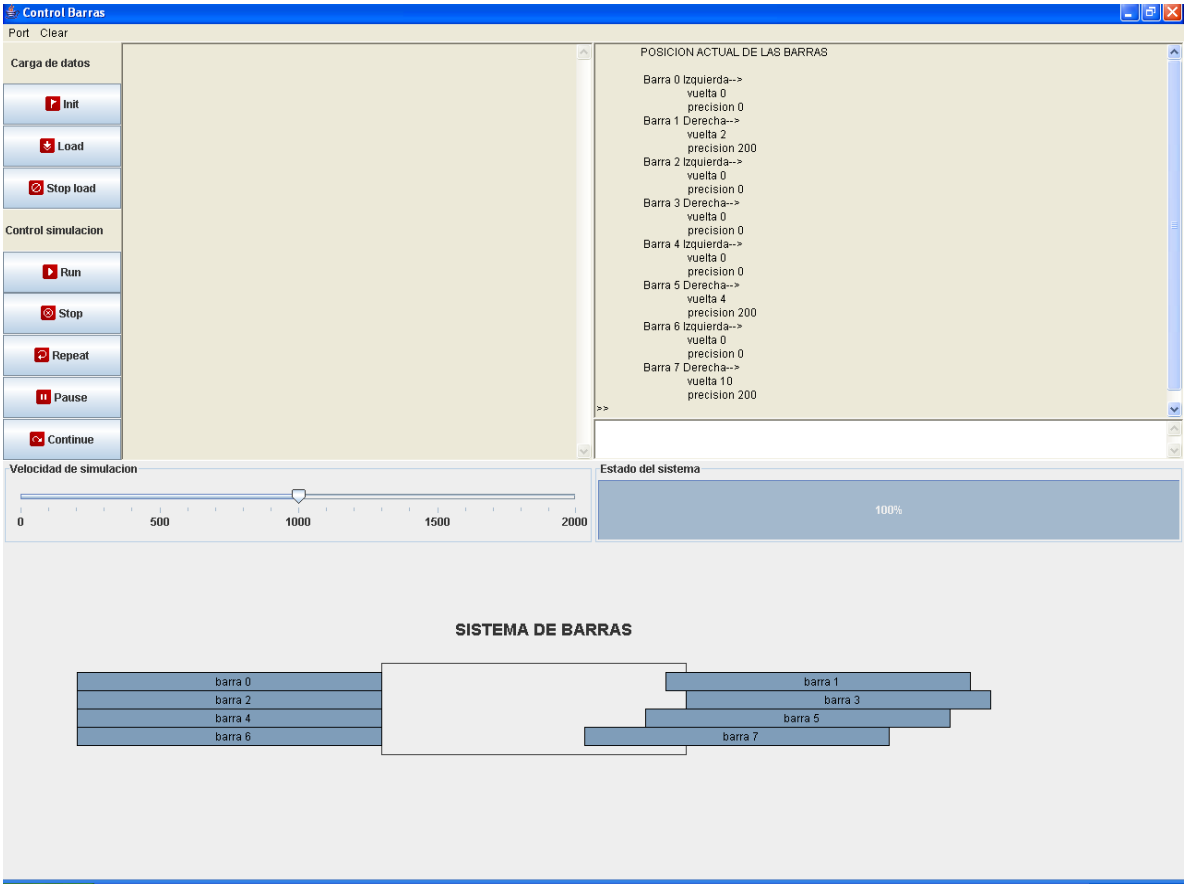


Figura 32: Aspecto del interfaz grafico en java.

El interfaz java permite interactuar con el microprocesador embebido en la FPGA mediante la comunicación mediante el puerto serie. Para ello dispone de los siguientes comandos de ejecución:

Init: comando que abre la ventana para inicializar la posición a la que deben llegar las barras.

Load: comando que manda la posición en la que debe colocarse las barras y comienza a leer la progresión de movimientos obtenida mediante la comunicación con la FPGA.

Stop load: finalizar la carga de datos obtenidos del microprocesador.

Run: comenzar la simulación del movimiento de las barras.

Stop: finalizar la simulación.

Repeat: repetir la simulación.

Pause: parar la simulación.

Rerun: reanudar la simulación en la posición actual.

Figura 33: Detalle de la parte superior del interfaz grafico

La pantalla está compuesta por dos paneles de texto situados en la parte superior, uno a la derecha y otro a la izquierda. En estos dos paneles de texto no está permitida la lectura ya que tiene carácter informativo.

En el panel situado a la derecha, se muestra la información leída directamente del programa software que se está ejecutando en tiempo real en la FPGA. Básicamente, esta información se reduce a los códigos de información enviados por la FPGA y que la aplicación java utiliza para representar el movimiento de las barras. El panel situado en la izquierda es una consola, en la cual se muestran los comandos que se están ejecutando y la posición de las barras cuando se realiza la simulación de las barras. Situado debajo del panel informativo de simulación se encuentra un panel de texto que permite introducir los comandos para controlar el sistema. En la parte superior derecha se puede observar los botones de control. Por tanto, es posible manejar la aplicación tanto con la consola de comandos como mediante los botones.

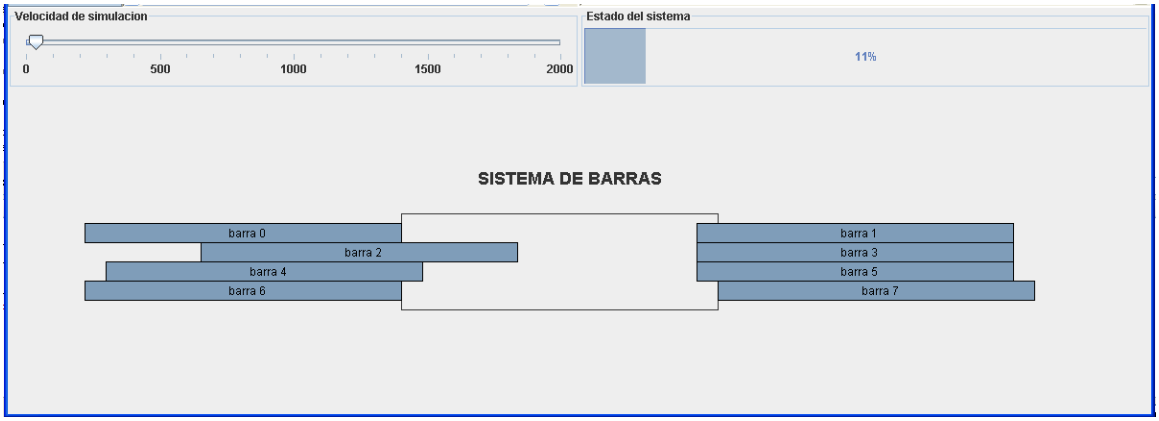


Figura 34: Detalle de la parte inferior del interfaz grafico

En la parte inferior de la aplicación se puede observar: un panel en el cual se muestra la imagen de 8 barras que se van desplazando con la simulación. En el panel de "velocidad de simulación" puedes determinar la velocidad a la que se realiza la simulación, mientras que en el panel "Estado del sistema" se puede observar el porcentaje de proceso simulado hasta el momento.

4.5.3 Inicialización de los datos

El objetivo principal de la aplicación java es poder controlar el movimiento de las barras. Para ello, cuando se introduce el comando "init" por la línea de comandos (o en su defecto se utilicen los botones de apoyo existentes) se abre una pantalla en el sistema en la cual se debe especificar la posición y la vuelta en la cual se desea colocar la barra.

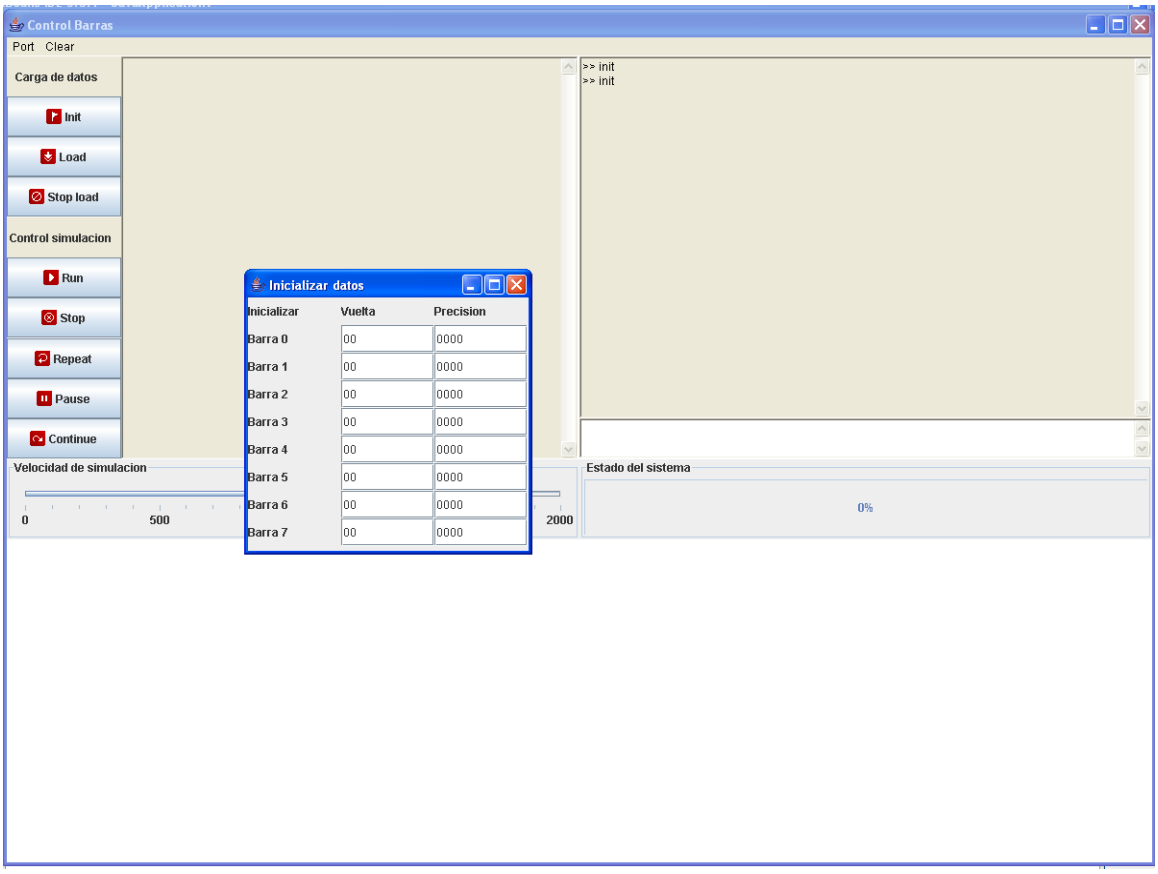


Figura 35: Detalle inicialización de datos

4.5.4 Carga de datos

Cuando se activa la carga de datos los datos inicializados se cargan en la FPGA. Si la posición de las barras no ha sido inicializada, los valores por defecto serán los correspondientes a la posición de reset de las barras. Tras cargar los datos, el sistema comienza a desplazar las barras y a mandar un código de posición a la aplicación java. La aplicación guarda toda la información que recibe y la muestra por uno de los paneles.

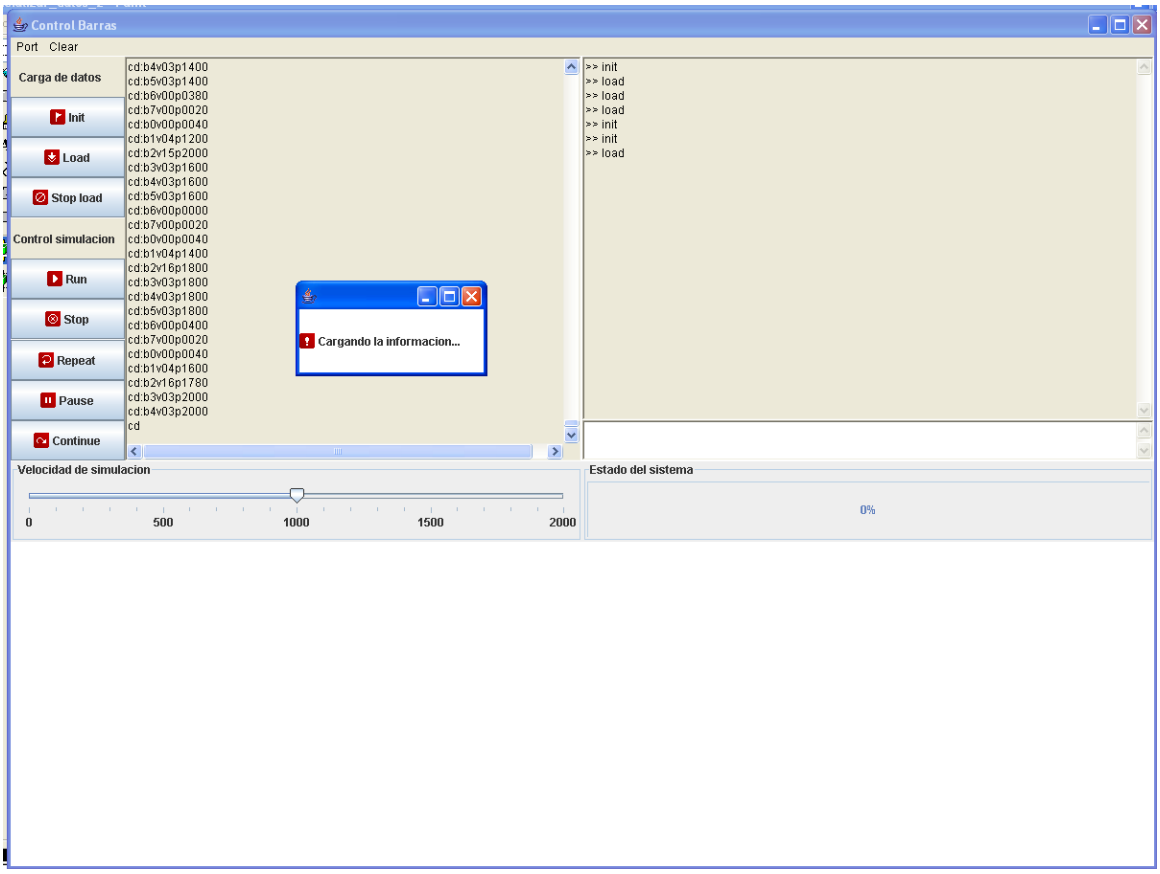


Figura 36: Detalle carga de datos

4.5.5 Simulación

Una vez recibida la información de la placa, se puede hacer una representación del movimiento de las barras. Para ello se utilizaran los códigos obtenidos en la etapa descarga de datos. Estos datos son procesados por el sistema el cual va mostrando la evolución del movimiento mediante un diagrama en la parte inferior de la pantalla, y los datos numéricos en uno de los paneles. El usuario puede consultar la barra de proceso, en la cual se indica el porcentaje de instrucciones realizadas, y pude también elegir la velocidad de la simulación.

Además, la aplicación dispone de una serie de comandos de control de la simulación que permiten terminar la simulación (stop), parar la simulación (pause), reanudar la simulación (rerun) o repetir la ultima simulación realizada (repeat).

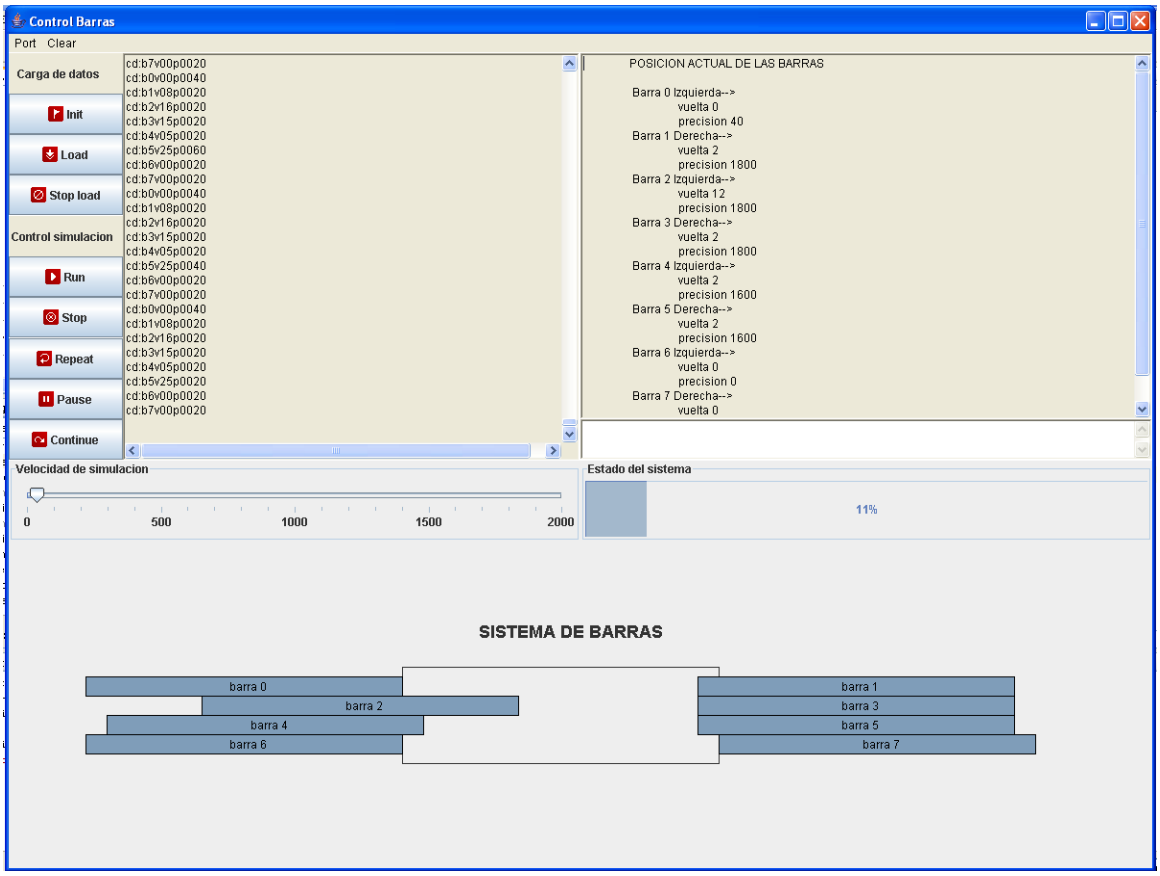


Figura 37: Detalle de ejemplo de simulación.

5 Integración, pruebas y resultados

5.1 Introducción

Para analizar los parámetros de ocupación y realizar las simulaciones mediante el interfaz de simulación Java, se ha utilizado una placa VirtexII Pro de Xilinx, donde se ha cargada el diseño de un módulo simple:

Figura 38: Placa XUP Virtex II Pro de Xilinx.

5.2 Pruebas

Durante el diseño del sistema, para cada uno de los periféricos será necesario realizar un seguimiento de su funcionamiento mediante la utilización del programa de simulación ModelSim. Este programa permite simular el comportamiento general de los periféricos que se encuentran incluidos en el diseño. De esta manera se puede observar el flujo de datos en el bus OPB y el funcionamiento de los módulos. La siguiente figura muestra una simulación del funcionamiento del módulo opb_8pwm.

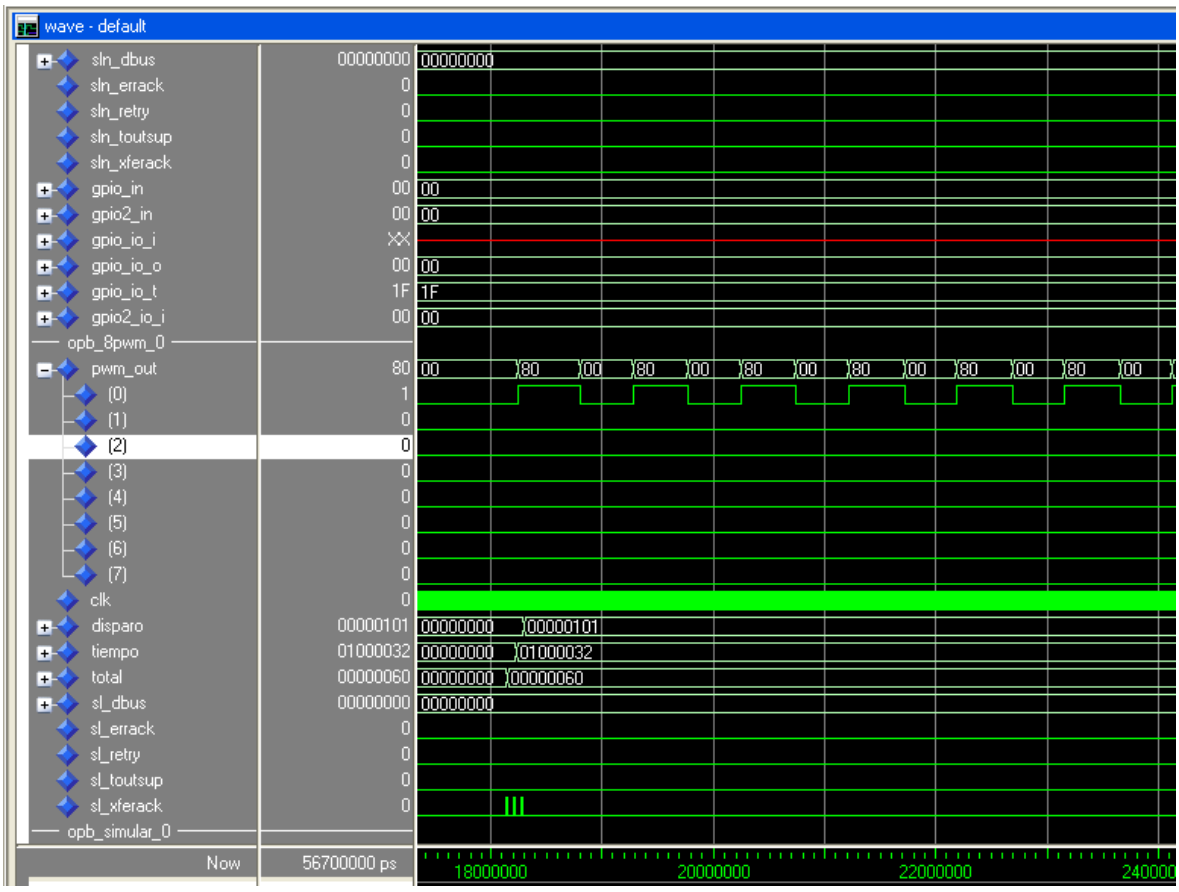


Figura 39: Simulación del módulo opb_8pwm.

No solo permitirá analizar el funcionamiento de los periféricos, sino que también permite observar procesos dominados por el código software contenido en el microprocesador, como la evolución de un reset del sistema. La siguiente figura es un ejemplo del mismo.

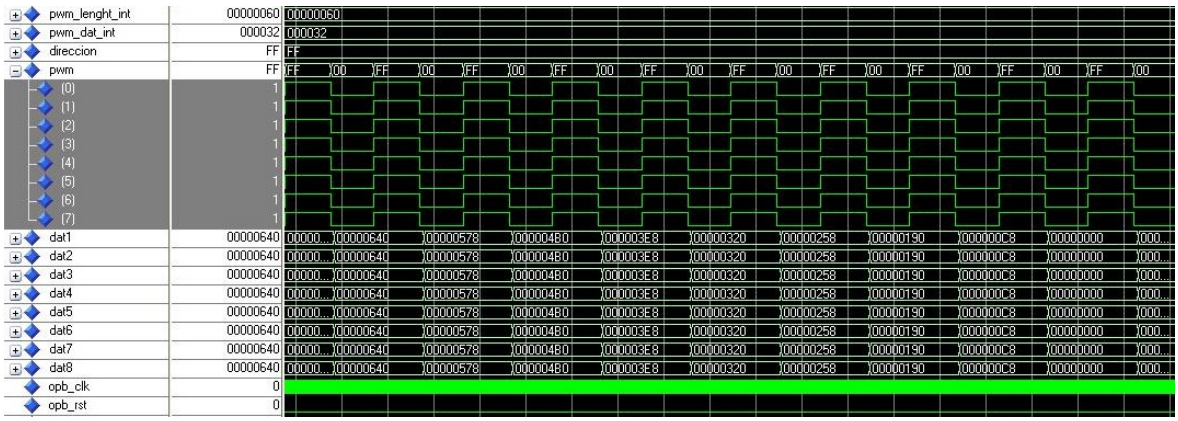


Figura 40: Ejemplo de simulación de reset.

5.3 Datos de integración del sistema de pruebas

5.3.1 Datos de ocupación de los periféricos.

Una vez implementado el diseño e incluidos los periféricos en el microprocesador, se instancia el módulo simple de movimiento de 8 barras mediante la utilización de la herramienta EDK.

A continuación se muestran los datos de ocupación de cada uno de los periféricos incluidos en el sistema. Estos datos son obtenidos de los ficheros *map report* (.mrp).de la herramienta ISE con al que los se han diseñado.

	Opb_8pwm	Opb_simular	Opb_my_gpio
Número de Slices:	203	1145	252
Número de Slice Flip Flops:	214	1113	53
Número de 4 input LUTs:	310	1071	476
Número de IOs:	125	381	365
Número de bonded IOBs:	118	377	327
Número de GCLKs:	1	1	1

Tabla 8: Datos de ocupación de los periféricos.

5.3.2 Datos de ocupación del módulo de movimiento

Los datos de ocupación totales obtenidos al descargar un módulo completo de control de ocho barras en una FPGA VIRTEX-II Pro. Estos datos se han del fichero *map report* (.mrp).de la herramienta EDK.

COMPONENTES	DATOS DE OCUPACIÓN		
Número de BUFGMUXs	2 de	16	12%
Número de DCMs	1 de	8	12%
Número de External IOBs	32 de	556	5%
Número de LOCed IOBs	32 de	32	100%
Número de MULT18X18s	3 de	136	2%
Número de RAMB16s	32 de	136	23%
Número de SLICEs	6346 de	13696	46%

Tabla 9: Datos de ocupación de un módulo simple de control

Estos resultados también incluyen el módulo de simulación, de módulo de simulación. Por tanto, nuestro módulo simple ocuparía realmente tan solo 5201 Slices. Este el parámetro más comprometido con respecto a la ocupación del sistema y que determinara el tipo de FPGA que deberá utilizar.

5.3.3 Características del código software

El tamaño del fichero .elf obtenido al compilar el código es:

```
mb-size módulo_8barras/executable.elf
   text    data     bss     dec     hex filename
  14197    1840    1096   17133    42edmódulo_8barras/executable.elf
```

5.3.4 Características temporales

El módulo implementado en la Virtex II Pro utilizada en el laboratorio tiene un periodo mínimo de 9,959 ns, lo que implica una frecuencia máxima de 100.412 MHz.

5.3.5 Mapeado de direcciones

De esta implementación también se obtiene la información referente al mapeado de memoria:

```
Address Map for Processor microblaze_0
     (0x00000000-0x0000ffff) dlmb_cntlrdlmb
     (0x00000000-0x0000ffff) ilmb_cntlrilmb
     (0x40000000-0x4000ffff) PushButtons_5Bit  mb_opb
     (0x40020000-0x4002ffff) LEDs_4Bit mb_opb
     (0x40040000-0x4004ffff) DIPSWs_4Bit  mb_opb
     (0x40600000-0x4060ffff) RS232_Uart_1 mb_opb
     (0x41400000-0x4140ffff) debug_module mb_opb
     (0x41c00000-0x41c0ffff) opb_timer_0  mb_opb
     (0x7a200000-0x7a20ffff) opb_simular_0   mb_opb
     (0x7a220000-0x7a22ffff) opb_my_gpio_0   mb_opb
     (0x7d400000-0x7d40ffff) opb_8pwm_0mb_opb
```

El mapa de direcciones muestra el area de direcciones asignada a cada periférico incluido en el microcontrolador.

5.4 Datos de integración del sistema sencillo

A continuación se analiza las características de ocupación del sistema real del prototipo que sería implementado. En este sistema el módulo de simulación del sistema desaparece, y también se elimina el control y simulación del movimiento mediante la aplicación java. Esto supone una evidente reducción tanto del número de LUT ocupadas por los periféricos, como de las BRAMs necesarias para la programación del microprocesador.

La disminución del número de BRAMs es debido a que la comunicación con la aplicación java se utiliza la función “xil_printf”, la cual es una operación sumamente costosa [31].

5.5 Necesidades del diseño

Una vez conocía las necesidades de ocupación del módulo simple, se deben calcular las necesidades de ocupación totales del diseño completo.

Si el sistema está formado por 100 pares de barras, se necesita incluir dentro de misma FPGA al menos 13 módulos de 8 barras de control. Así, los datos de ocupación totales serian:

	Módulo simple	Total ocupación
Número de DCMs	1	13
Número de MULT18X18s	3	39
Número de RAMB16s	32	416
Número de SLICEs	5201	67.613

Tabla 10: Tabla de ocupación del sistema.

La memoria RAMB será la restricción de ocupación del diseño tras eliminar el modulo de simulación.

5.6 Datos de ocupación y rutado sobre diferentes FPGAs

A continuación se muestran los datos de ocupación del sistema embebidos en diferentes diseños de placas. Sobre la FPGA se instancia tan solo el microcontrolador y el módulo PWM y el módulo GPIO.

Además, este nuevo sistema implementado no contendrá las instrucciones de de conexión con el puerto serie que permitían el control mediante el programa de simulación. También estos diseños tendrán una restricción del reloj de la FPGA de 20 ns.

Ocupación en Spartan y VirtexII Pro						
Tipo de arquitectura	**Tipo de dispositivo**	**Número de DCMs**	**Número de MULT18X18s**	**Número de RAMB16s**	**Número de SLICEs**	**Mínimo retardo**
Spartan3	Xc3s2000	1	3	32	6,446	19.800ns
Virtex II Pro	xc2v2000	1	3	32	6440	16.625ns

Ocupación en Virtex IV y Virtex V				
Tipo de arquitectura	**Tipo de dispositivos**	**Número de Slice Flip-Flop**	**Número de LUT 4 entradas**	**Número de DCMs**
Virtex IV	xc4vfx100	5,555	8,699	1
	Números de RAMB 16s	**Número de Slices ocupados**	**Número total de LUT de entrada**	**Retardo mínimo**
	32	6,640	9,883	10.941ns

Tabla 11: Tabla de datos de ocupación según la FPGA utilizada.

Estos datos de ocupación directa se obtienen mediante de los ficheros de simulación .par (*Place and Route*) y .mrp (*Map Report*), obtenido tras ejecutar la generación de bitstream [Apéndice F].Los periféricos producirán una restricción de la velocidad del sistema, ya que dentro de los mismo se encuentra el camino crítico. Esta restricción será menor conforme se aumente la calidad y velocidad del módulo de FPGA utilizado.

5.7 : Análisis de posibles soluciones de diseño.

En base a los datos de ocupación obtenidos y analizados en el apartado anterior se deberá elegir una FPGA adecuada a las necesidades tanto de memoria del sistema. A continuación se incluyen las tablas que muestran la información de la gama de FPGAs ofrecida por Xilinx [1] en el mercado:

Las FPGAs **Spartan™** de Xilinx representa la gama más económica de FPGAs de Xilinx. Existen varios modelos centrados en diferentes aplicaciones:

- **Spartan / XL** : modelo con alta densidad (entre 5000 y 4000 de sistemas de puertas)
- **Spartan-II** / **Spartan-IIE**: modelo clásico de FPGA de bajo coste, que permite una solución económica para los diseños.
- **Spartan-3** [17]**:** alta densidad de lógica y pines de entrada y salida y altamente preparado para procesado de datos.
    - Spartan-3E/Spartan-3A/Spartan-3AN/Spartan-3A DSP

Device	System Gates	Equivalent Logic Cells[1]	CLB Array (One CLB = Four Slices)			Distributed RAM Bits (K=1024)	Block RAM Bits (K=1024)	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs						
XC3S50[2]	50K	1,728	16	12	192	12K	72K	4	2	124	56
XC3S200[2]	200K	4,320	24	20	480	30K	216K	12	4	173	76
XC3S400[2]	400K	8,064	32	28	896	56K	288K	16	4	264	116
XC3S1000[2]	1M	17,280	48	40	1,920	120K	432K	24	4	391	175
XC3S1500	1.5M	29,952	64	52	3,328	208K	576K	32	4	487	221
XC3S2000	2M	46,080	80	64	5,120	320K	720K	40	4	565	270
XC3S4000	4M	62,208	96	72	6,912	432K	1,728K	96	4	712	312
XC3S5000	5M	74,880	104	80	8,320	520K	1,872K	104	4	784	344

Tabla 12: Tabla Spartan 3 de Xilinx [17]

Utilizando una Spartan3 sería necesario utiliza una cuya de modelo XC3S4000 u otro modelo con mayor capacidad de memoria. La utilización de este tipo de FPGA abarataría costes pero sería necesario utilizar una placa por cada módulo controlador de 8 barras.

Las FPGAs **Virtex™** de Xilinx representa la gama alta de FPGAs de Xilinx. Existen varios modelos centrados en diferentes aplicaciones:

- **Virtex / E / EM** : primeras Virtex de Xilinx que suponen una mayor densidad de lógica,
- **Virtex-II Platform FPGA:** mejora de la productividad, eficiencia y flexibilidad de la FPGA.
  - **Virtex-II Pro Platform FPGA** [18]**:** plataformas con FPGA Virtex II diseñadas para ser utilizadas como medio de aprendizaje en las universidades. Este modelo de FPGA ha sido el utilizado para el desarrollo del proyecto.

Device(1)	RocketIO Transceiver Blocks	PowerPC Processor Blocks	Logic Cells(2)	CLB (1 = 4 slices = max 128 bits)		18 X 18 Bit Multiplier Blocks	Block SelectRAM+		DCMs	Maximum User I/O Pads
				Slices	Max Distr RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)		
XC2VP2	4	0	3,168	1,408	44	12	12	216	4	204
XC2VP4	4	1	6,768	3,008	94	28	28	504	4	348
XC2VP7	8	1	11,088	4,928	154	44	44	792	4	396
XC2VP20	8	2	20,880	9,280	290	88	88	1,584	8	564
XC2VPX20	8(4)	1	22,032	9,792	306	88	88	1,584	8	552
XC2VP30	8	2	30,816	13,696	428	136	136	2,448	8	644
XC2VP40	0(3), 8, or 12	2	43,632	19,392	606	192	192	3,456	8	804
XC2VP50	0(3) or 16	2	53,136	23,616	738	232	232	4,176	8	852
XC2VP70	16 or 20	2	74,448	33,088	1,034	328	328	5,904	8	996
XC2VPX70	20(4)	2	74,448	33,088	1,034	308	308	5,544	8	992
XC2VP100	0(3) or 20	2	99,216	44,096	1,378	444	444	7,992	12	1,164

Tabla 13: Tabla Virtex II de Xilinx [18]

Esta gama no permitiría incluir todos los módulos de control de 8 barras que deben componer el sistema en una única FPGA, sin embargo, sería posible contener los 13 módulos en dos Virtex II Pro XC2VPX70 y XC2VP100. La utilización de este modulo de FPGA podría ser útil para continuar el desarrollo del prototipo.

- **Virtex-4 Multi-Platform FPGA Family** [19]: las FPGAs más utilizadas actualmente en mercado, con unas completas prestaciones de procesado de datos y memoria.

  - **Virtex-4 LX:** altas prestaciones lógicas.
  - **Virtex-4 SX:** altas prestaciones en el procesado de señales.
  - **Virtex-4 FX:** procesado embebido y conexión serie.

Device	Configurable Logic Blocks (CLBs)(1)				XtremeDSP Slices(2)	Block RAM		DCMs	PMCDs	PowerPC Processor Blocks	Ethernet MACs	RocketIO Transceiver Blocks	Total I/O Banks	Max User I/O
	Array(3) Row x Col	Logic Cells	Slices	Max Distributed RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)							
XC4VLX15	64 x 24	13,824	6,144	96	32	48	864	4	0	N/A	N/A	N/A	9	320
XC4VLX25	96 x 28	24,192	10,752	168	48	72	1,296	8	4	N/A	N/A	N/A	11	448
XC4VLX40	128 x 36	41,472	18,432	288	64	96	1,728	8	4	N/A	N/A	N/A	13	640
XC4VLX60	128 x 52	59,904	26,624	416	64	160	2,880	8	4	N/A	N/A	N/A	13	640
XC4VLX80	160 x 56	80,640	35,840	560	80	200	3,600	12	8	N/A	N/A	N/A	15	768
XC4VLX100	192 x 64	110,592	49,152	768	96	240	4,320	12	8	N/A	N/A	N/A	17	960
XC4VLX160	192 x 88	152,064	67,584	1056	96	288	5,184	12	8	N/A	N/A	N/A	17	960
XC4VLX200	192 x 116	200,448	89,088	1392	96	336	6,048	12	8	N/A	N/A	N/A	17	960

Device	Configurable Logic Blocks (CLBs)(1)				XtremeDSP Slices(2)	Block RAM		DCMs	PMCDs	PowerPC Processor Blocks	Ethernet MACs	RocketIO Transceiver Blocks	Total I/O Banks	Max User I/O
	Array(3) Row x Col	Logic Cells	Slices	Max Distributed RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)							
XC4VSX25	64 x 40	23,040	10,240	160	128	128	2,304	4	0	N/A	N/A	N/A	9	320
XC4VSX35	96 x 40	34,560	15,360	240	192	192	3,456	8	4	N/A	N/A	N/A	11	448
XC4VSX55	128 x 48	55,296	24,576	384	512	320	5,760	8	4	N/A	N/A	N/A	13	640
XC4VFX12	64 x 24	12,312	5,472	86	32	36	648	4	0	1	2	N/A	9	320
XC4VFX20	64 x 36	19,224	8,544	134	32	68	1,224	4	0	1	2	8	9	320
XC4VFX40	96 x 52	41,904	18,642	291	48	144	2,592	8	4	2	4	12	11	448
XC4VFX60	128 x 52	56,880	25,280	395	128	232	4,176	12	8	2	4	16	13	576
XC4VFX100	160 x 68	94,896	42,176	659	160	376	6,768	12	8	2	4	20	15	768
XC4VFX140	192 x 84	142,128	63,168	987	192	552	9,936	20	8	2	4	24	17	896

Tabla 14: tabla de Virtex IV de Xilinx [19].

Dentro de la gama Virtex IV, sería aconsejable la utilización de la familia LX, ya que dispone de FPGAs con mayor capacidad lógica (Slices), lo que permitirá incluir todos los módulo de control en una única FPGA si se utiliza el modelo XC4VLX200 (con 89,088 Slices). El resto de familias de la gama, pueden ser utilizadas pero al igual que las gamas anteriores necesitaran incluir al menos dos FPGAs para controlar todo el sistema de barras.

- **Virtex-5 Multi-Platform FPGA Family** : la nueva plataforma de Xilinx. Aunque son dispositivos con alta velocidad y capacidad de procesamiento, son excesivamente caros frente a una Virtex IV.

    - **Virtex-5 LX:** Altas prestaciones lógicas
    - **Virtex-5 LXT:** Altas prestaciones lógicas y conexión serie de baja potencia.
    - **Virtex-5 SXT:** optimizada para procesado de datos, aplicaciones con utilización de memoria intensiva y conexión serie de baja potencia.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

En el presente proyecto se ha desarrollado un prototipo de sistema de lectura y procesado para múltiples sensores embebidos en una FPGA para el control de un sistema de telescopio. El diseño es una colaboración con el Instituto de Astrofísica de Canarias (IAC) para el control de un telescopio basado en la técnica de la espectroscopia. El sistema se encuentra dentro de un entorno de criogenización, lo que implica trabajar en sistema con parámetros inestables. Se desarrollara un sistema realimentado que permita posicionar ciertas barras que controlan la apertura de una lente.

El sistema bajo estas condiciones necesita de adaptar su funcionalidad dependiendo de las condiciones mecánicas de los actuadores que cambian de características con la temperatura. Para lograr estos requerimientos de precisión y velocidad en la actuación, manteniendo un bajo coste se optó con un diseño hardware-software sobre una plataforma de lógica reconfigurable.

El presente prototipo está compuesto por dos partes claramente diferenciadas: un diseño hardware que contiene un microprocesador y periféricos para el control de los actuadores, y por otro un diseño software donde se describe el comportamiento del control.

Para la consecución del diseño se han seguido los pasos que se enumeran a continuación:

Análisis de las especificaciones del sistema: se analizan las características del sistema mecánico las cuales determinaran el diseño del sistema de control. Del análisis del sistema mecánico se determina:

- El sistema dispone de dos *encoder*, uno relativo y otro absoluto. El *encoder* absoluto viene implementado mediante las cuatro capacidades del sistema. Para el *encoder* relativo será necesario incluir un contador que permita identificar el tramo de barra en el cual están actuado los sensores capacitores.

- Serán necesario al menos 17 bits para describir absolutamente una posición de una barra.

- Cada una de las barras deberá ser controlada de manera independiente debido a las condiciones especiales de fricción y desplazamiento que pueden aparecer en el sistema debido a la inclinación del mismo y las condiciones de criogenización a las que se encuentra sometido el sistema.

Documentación y análisis sobre las posibles opciones de diseño. En base a las especificaciones del sistema se analizan las posibles opciones de mercado que permitan implementar un sistema de control dentro de una FPGA:

- Se descarta la opción de utilizar un sistema hardware de control (PID) debido a la inestabilidad paramétrica existente en entornos de criogenización.

- Se analiza la opción de utilizar un microprocesador o un microcontrolador para implementar el sistema:

  - Se descarta la utilización de microprocesadores "*hard core*" por su escasa flexibilidad y la posible ineficiencia en sistema sencillo debido a su gran peso en área de silicio.

  - Se opta por el co-diseño, la utilización de un microcontrolador y periféricos hardware. Dentro de los microprocesadores posibles se opta por un "*soft core*" de 32 bits, ya que se ajusta mejor a las especificaciones del sistema.

Descripción de un módulo simple de diseño. Se describe un módulo sencillo de control para 8 barras pareadas. Este módulo es la base del sistema. cada uno de estos módulos estará compuesto por:

- Microprocesador MicroBlaze, que contiene el código software con el algoritmo de realimentación para el control de los piezoeléctricos.

- Periférico PWM, que describe las 8 señales *pwm* necesarias para controlar los piezoeléctricos de cada una de las barras.

- Periférico de simulación, que contiene 8 sub-módulos que simulan el movimiento de las barras y devuelve información sobre la posición de las barras.

- Periférico GPIO que capta los datos del módulo de simulación.

Implementación de un código software de control: Se ha descrito un código en C que se correrá en el microprocesador embebido. Este código permite controlar la comunicación mediante el puerto serie, captura de los datos de módulo GPIO, actualiza los datos de parametrización del *pwm* y mantiene la cuenta del *encoder* relativo de las barras. Además, en el prototipo del sistema que incluye el periférico de simulación, se actualizaran los datos del módulo de simulación.

Descripción de programa de simulación del sistema: Se ha descrito un programa en java que permite cargar las posiciones requeridas en el sistema y simular el movimiento de las barras. Este programa que inicialmente se diseño como método de control de las simulaciones permite visualizar el movimiento de las barras, cargar las posiciones de las barras y resetear al sistema. El código del programa podría ser reciclado para utilizarse en un programa definitivo para el control del sistema.

Comprobación de las características del sistema: Se ha comprobado el correcto funcionamiento de cada módulo independientemente mediante el programa de simulación ModelSim y del conjunto en general mediante el sistema de simulación implementado.

Análisis de las características de ocupación del sistema y posibles dispositivos FPGA para la implementación: Se han obtenido los datos de ocupación del sistema. la principal restricción del sistema en la necesidad de memoria para el procesado de los datos.

Como resultado se ha obtenido un sistema control completamente parametrizable y basado en módulos simples de control, controlados por microcontroladores que permiten desplazar las barras de manera óptima y a gran velocidad, y periféricos para el control de las señales *pwm*.

La utilización de una FPGA frente al anterior modelo de movimiento de las barras presenta una mejora no solo en el tiempo de ejecución sino también en los aspectos económico. A pesar del coste de mercado de las FPGAs necesarias para montar el sistema de control, el coste del enfriamiento del sistema (del orden de miles de euros por cada enfriamiento del sistema) es en proporción mucho más económico.

6.2 Trabajo futuro

El proyecto anteriormente descrito en este documento se trata de un prototipo de diseño en base a las especificaciones provistas por el IAC. Como se describe en apartados anteriores, la función de transferencia existente entre le piezoeléctrico y el desplazamiento de las barras está aún por determinar. Por tanto, dentro de las posibles mejorar cabría indicar:

Determinación de la función de transferencia del sistema en base a los datos obtenidos experimentalmente dentro de la cámara de vacio a temperaturas criogénicas. Con esta

nueva función de transferencia se adaptaran los parámetros de la señal *pwm* que mueve los piezoeléctricos.

Incluir una mayor funcionalidad al microprocesador. En un principio, el diseño del sistema ha sido pensado para contener la menor memoria posible. De esta manera se evita la utilización de memorias externas, que encarecerían el diseño y lo harían más complejo en su implementación.

- Funciones de control de distancia entre las barras pareadas. Estas funciones añadirían inteligencia al sistema, permitiendo evitar colisiones.
- Funciones de control de obstrucción, que permiten detectar cuando una barra se ha quedado parada. Como primera medida, esta función debería provocar que le sistema de barras se reiniciara y activase una señal de alarma para el usuario.

Desde el punto de vista del diseño del hardware se debería estudiar la posibilidad de compartir la memoria de programa entre dos ó más microprocesadores de modo de ahorrar memoria y evitar tener que utilizar memoria externa al aumentar la funcionalidad software.

El montaje final con los actuadores y el funcionamiento en estado de criogenización generará la necesidad de ajustes que, se supone a priori, solo influirán en el software de control y lo podrá incluso ajustar un técnico sin necesidad de entender en profundidad los detalles del diseño completo.

7 Referencias

1. **Xilinx**, www.xilinx.com, información corporativa de Xilinx.
2. **Altera**, www.altera.com, información corporativa de Altera.
3. **Lattice semiconductor**, www.latticesemi.com, información corporativa de Lattice Semiconductor.
4. **Actel**, www.actel.com, información corporativa de Actel.
5. **Atmel**, www.atmel.com, información corporativa de Atmel.
6. **Quicklogic**, www.quicklogic.com, información corporativa de Quicklogic.
7. **Achronix**, www.achronix.com, información corporativa de Achronix.
8. **Mathstar**, www.mathstar.com, información corporativa de Mathstar.
9. **Java Package javax.comm,** http://java.sun.com/products/javacomm/reference/api/index.html, información sobre el API de comunicaciones.
10. Xilinx,"**Designing Custom OPB Slave Peripherals for MicroBlaze**", 1-800-255-7778, www.xilinx.com, febrero 2008.
11. Xilinx," **OPB IPIF**", Product Specification DS414, http://www.xilinx.com/products/ipcenter/OPB_IPIF_Architecture.htm, 2 de Diciembre del 2005.
12. Xilinx,"**EDK Concepts, Tools, and Techniques: A Hands-on Guide to Effective Embedded System Design**", EDK Concepts, Tools, and Techniques P/N XTP013 (v. 9.1i), www.xilinx.com , Septiembre 2007 .
13. Xilinx,"**EDK 8.2 MicroBlaze Tutorial in Spartan 3**" 1-800-255-7778 WT001 (v4.0), www.xilinx.com, Agosto 30, 2006.
14. Xilinx,"**MicroBlaze Processor Reference Guide Embedded Development Kit EDK 9.1i**", MicroBlaze Processor Reference Guide UG081, http://support.xilinx.com/support/techsup/tutorials/index.htm,Septiembre del 2007.
15. Xilinx," **Virtex-4 Family Overview**", Preliminary Product Specification DS112 (v1.6), www.xilinx.com ,Octubre 10, 2006
16. Xilinx," **Virtex-4 User Guide**",Virtex-4 User Guide UG070 ,www.xilinx.com, Octubre 6, 2006
17. Xilinx," **Spartan-3 FPGA Family: Complete Data Sheep** ",Product Specification DS099 , http://www.xilinx.com/support/documentation/spartan-3.htm, 30 de Noviembre del 2007.

18. Xilinx," **Virtex Pro II and Virtex Pro II X Plataform Family: Complete Data Sheep** ",Product Specification DS083 , http://www.xilinx.com/support/documentation/virtex-ii_pro.htm, 5 de Noviembre del 2007.
19. Xilinx," **Virtex-4 FPGA Family: Complete Data Sheep**", Product Specification DS112 , http://www.xilinx.com/support/documentation/virtex-4.htm, 28 de Septiembre del 2007.
20. Xilinx," **Xilinx University Program Virtex-II Pro Development System: Hardware Reference Manual** ",XUP Virtex-II Pro Development System UG069 (v1.0), www.xilinx.com , Marzo 8, 2005
21. Xilinx," **PicoBlaze™ 8-bit Microcontroller Reference Design for FPGAs and CPLDs**", www.xilinx.com , Agosto 2004.
22. Stephen Brown and Jonathan Rose, "**Architecture of FPGAs and CPLDs: A Tutorial**", Department of Electrical and Computer Engineering University of Toronto
23. Jason G. Tong, Ian D. L. Anderson and Mohammed A. S. Khalid," **Soft-Core Processors for Embedded Systems**", 1-4244-0765 University of Windsor - Department of Electrical and Computer Engineering Research Centre for Integrated Microsystems Windsor, Ontario, Canada,2006
24. Jerry Case, Nupur Gupta, Jayant Mittal and David Ridgeway,"**Design Methodologies for Core-Based FPGA Designs**", en www.xilinx.com/,April 9, 1997
25. F. Krahenbiihl (chairman), B. Bernstein, M. Danikas, J. Densley, K. Kadotani,,M. Kahle, M. Kosaki, H. Mitsui, M. Nagao, J.Smit, and T. Tanaka,"**Properties of Electrical Insulating Materials at Cryogenic Temperatures: A Literature Review**", 08S3-75 IEEE Electrical Insulation Magazine N0.4 VOl. 10, Julio,Agosto 1994, Switzerland
26. Ahmad Hammoud, Scott Gerber, Richard L. Patterson, Malik Elbuluk, "**Electronic Components and Circuits for Extreme Temperature Environments**", Electronic Circuits and System 0-7803-8163 IEEE, Septiembre del 2003
27. Campbell, William A. Jr., and John J. Scialdone," **Outgassing Data for Selecting Spacecraft Materials**", NASA Reference Publication 335385, http://outgassing.nasa.gov , Octubre del 2007
28. Walter J. Sarjeant, *Fellow IEEE,* Jennifer Zirnheld, *Member IEEE,* and Frederick W. MacDougall, "**Capacitors**", IEEE Transactions on plasma science, Octubre de 1998.
29. Xilinx, "**PicoBlaze™ 8-bit Microcontroller Reference Design for FPGAs and CPLDs**", Product Literature PN 0010949, http://www.xilinx.com/publications/prod_mktg/index.htm, 2006.
30. Eduardo Boemo, Ivan Gonzalez, Estanislao Aguayo, "Tutorial XlinxMicroBlaze-uCLinux", Escuela PolitecnicaSuperior,Universidad Autonoma de Madrid, http://www.ii.uam.es, 2007
31. Xilinx, "**LibXil Estándar C Libraries**", EDK 9.1i, http://www.xilinx.com/support/documentation/index.htm, Diciembre del 2006.
32. Xilinx ,"**OPB timer / counter**", Logic Core DS475, http://www.xilinx.com/support/documentation/index.htm,Diciembre 2005

8 Glosario

ASIC	**Aplication Specific Integrated Circuit**
CCC	**Closed Cycle Coolers**
CPLD	**Complex Programmable Logic Device**
CLB	**Configurable Logic Block**
DRAM	**Dinamic Random Access Memory**
EDK	**Embedded Design Kit**
EPROM	**Erasable Programable Read Only Memory**
EEPROM	**Electrical Erasable Programable Read Only Memory**
FPL	**Field Programmable Device**
FPGA	**Field Programmable Gate Arrays**
FSL	**Fast Simple Link**
GPIO	**General Purpose Input/Output**
IP	**Intelectual Property**
IPIF	**Intelectual Property Interface**
IOB	**Input Output Block**
LCA	**Logic Cell Array**
LMB	**Local Memory Bus**
MHS	**Microprocessor Hardware Specification**
PLA	**Programmable Logic Array**
PWM	**Pulse Wave Modulation**
OPB	**On-chip Peripheral Bus**
SoC	**System on Chip**

SRAM	**Static Random Access Memory**
XCL	**Xilinx Cache Link**

9 Anexos

A Microprocesador MicroBlaze

A.1Introducción

MicroBlaze es un procesador “soft core ” con un número reducido de instrucciones (RISC) optimizado para ser implementado en FPGAs [14].El procesador Microblaze de 32 bits cumple unos requisitos de ocupación y prestaciones muy rígidos, debida a la limitación de recursos de la FPGA, por este motivo, su funcionalidad puede resultar limitada en comparación con otros micros. Soluciones arquitectónicas como la supeerescalabilidad o la supersegmentacion no resultan adecuadas en un entorno lógico reprogramable. A esto abra que añadir que la frecuencia final se ve limitada por los altos retardos de interconexión de la FPGA. Permite un alto grado de configuración. La figura 7 muestra un esquema general de su estructura.

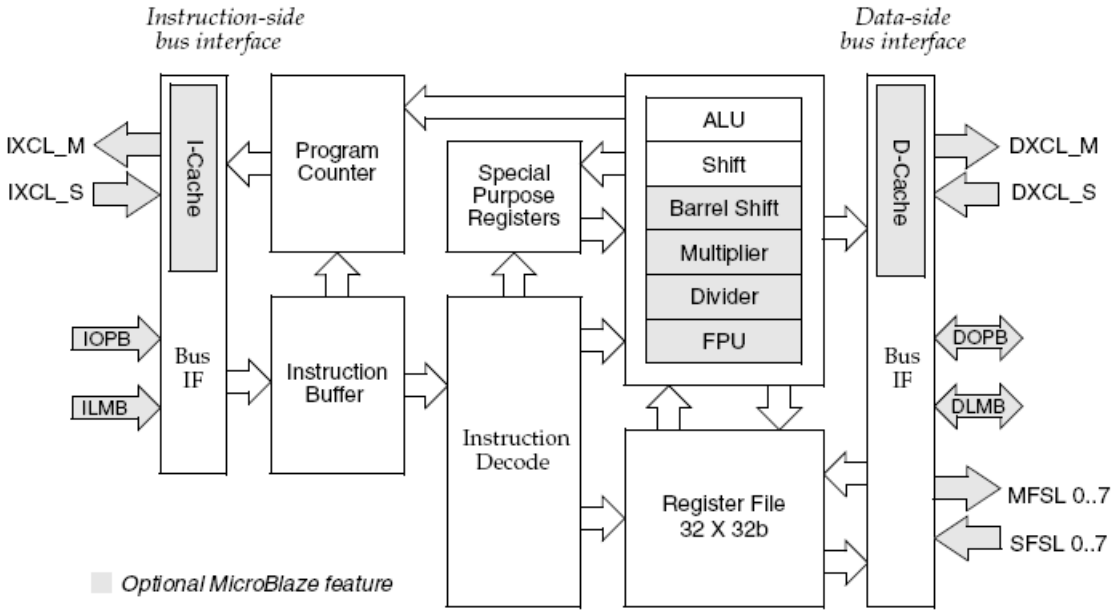


Figura 41: diagrama de la estructura general de un microprocesador Microblaze [14]

Sus principales características son:

- Datos de 32 bits
- Instrucciones de 32 bits por palabra(con 3 operandos y dos modos de direccionamiento)
- 32 registros de propósito general de un tamaño de 32 bits.
- Un pipeline de 3 estados a 5 estados (para la versión v6.00 de MicroBlaze).
- Soporta interrupciones y excepciones hardware.

A.2Repertorio de instrucciones

El sencillo diseño la que se ve forzado este micro, hace que se ajusta a una arquitectura de tipo RISC (*reduced instrution set computer*). Un limitado número de instrucciones permite reducir la complejidad de la unidad decodificación. En Microblaze el juego de instrucciones se compone de un total de 87 instrucciones (siempre considerando como diferentes instrucciones las que operan con valores inmediatos de aquellas que realizan la misma operación mediante registros). Las instrucciones se organizan en las siguientes categorías: aritméticas, lógicas, saltos, descarga/carga e instrucciones especiales (todas aquellas que no se pueden encuadrar en las otras categorías).

A.3Pipeline

Una arquitectura RISC aumenta fácilmente su rendimiento por medio de la segmentación (*Pipelining*) [9]. En el caso de MicroBlaze, el número de etapas de pipeline puede ser 3 o 5, ejecutándose una sola instrucción por ciclo de reloj. Frente a esta mejora en el diseño, será necesario incorporar mecanismos para evitar problemas relacionados con los saltos de programa y las instrucciones que necesiten un mayor número de ciclos para ejecutarse.

- **3 estados de segmentación:**

  Cuando el sistema está configurado para optimizar el coste computacional hardware. Se divide cada instrucción en tres etapas: fetch (traer la instrucción de memoria), “decode” (decodificar la instrucción) y “execute” (ejecutar la instrucción)

	Ciclo 1	**Ciclo 2**	**Ciclo 3**	**Ciclo 4**	**Ciclo 5**	**Ciclo 6**	**Ciclo 7**
Instrucción 1	Fetch	Decode	Execute				
Instrucción 2		Fetch	Decode	Execute	Execute	Execute	
Instrucción 3			Fetch	Decode	Stall	Stall	Execute

Tabla 15: ejemplo ejecución con 3 estados de segmentación

- ***5 estados de segmentación:***

Cuando el sistema esta segmentado en cinco estados, las diferentes etapas de cada instrucción serán: IF (fetch: traer el dato de memoria),OF (decode: decodificar la instrucción),EX(execute: ejecutar la instrucción),MEM(Access memory: hace a memoria),WB(whiteback)

	Ciclo 1	Ciclo 2	Ciclo 3	Ciclo 4	Ciclo 5	Ciclo 6	Ciclo 7	Ciclo 8	Ciclo 9
Instrucción 1	IF	OF	EX	MEM	WB				
Instrucción 2		IF	OF	EX	MEM	MEM	MEM	WB	
Instrucción 3			IF	OF	EX	Stall	Stall	MEM	WB

Tabla 16: ejemplo ejecución con 5 estados de segmentación

- **Saltos**

  Cuando en el programa se ejecuta un salto, la pipeline contiene instrucciones que no se corresponden con el flujo de ejecución del programa. Por tanto, tras producirse un salto efectivo será necesario vaciar el pipeline y volver a cargar las nuevas instrucciones calculadas con la dirección que contendrá nuestra dirección de salto. Un salto en Microblaze necesita tres ciclos de reloj para ejecutarse, dos de ellos se utilizan para cargar las nuevas instrucciones en el pipeline. En el repertorio de instrucciones se incluyen un cierto número de instrucciones de salto que permiten la ejecución de la instrucción que les precede, con el objetivo de reducir la penalización de vaciado. Esta técnica es conocida como *Delay Slots*.

A.4Registros internos y cache

Las FPGAs actuales disponen de memoria distribuida con un tiempo de acceso corto, cuando son utilizadas por la lógica cercana. Este tipo de memoria es utilizada por MicroBlaze para materializar sus 32 registros internos, el contador de programa y el registro de estado. Este último sólo contiene el bit de acarreo, habilitación de cachés, indicador de estado de parada (*break*), error en el FSL y bit de excepción producida por división por cero. Además de estos registros internos, MicroBlaze utiliza un *buffer* de 16 instrucciones mapeado en los registros de desplazamiento SRL de los *slices*. Este recurso es fundamental para un buen rendimiento del procesador. Por ejemplo, en caso de que no disponer de multiplicador y/o divisor hardware, se aprovecha el tiempo que tarda en ejecutarse esta instrucción (32 ciclos) para tener preparadas las siguientes instrucciones.

La utilización de cachés es una práctica habitual en arquitecturas modernas; el diseño de MicroBlaze no es una excepción. Pero en este caso, la utilización y configuración de los tamaños de caché pueden ser fijados por el usuario. Para ello, existe el siguiente conjunto de opciones a la hora de configurar el procesador: habilitación de caché de instrucciones y/o

datos, tamaño, rango de direcciones y tamaño de palabra, aunque esta última opción solo es válida para la caché de datos. Opciones no configurables por el usuario son tamaño de los bloques de caché, la política de sustitución de bloques o el grado de asociatividad de la caché.

Todas estas opciones se especifican antes de sintetizar el diseño. Las cachés son de tipo asociativo, por lo cual es necesario el calcular el número de bits de la dirección (*tag bits*) que especifican bloques contenidos en caché, mediante la ecuación.

Número *tag* bits = log2(rango de memoria cacheable) - log2(tamaño de la caché)

Las cachés utilizan los bloques de RAM fijos (BRAM) que contiene la FPGA. Sin embargo, códigos compactos también pueden mapearse en BRAM. Por lo tanto, la utilización de cachés será necesaria en aquellos casos donde el código o los datos utilizado por MicroBlaze residan fuera de la FPGA.

A.5Buses del sistema

MicroBlaze sigue el modelo de arquitectura Harvard, donde datos e instrucciones son almacenados en memorias diferentes. Gracias a la capacidad de reconfiguración de las FPGAs, es posible configurar el sistema con diferentes opciones sobre los *buses*, pudiéndose reducir de este modo el tamaño final del sistema.

MicroBlaze utiliza el estándar *CoreConnect* creado por IBM, para conectar diferentes elementos en un circuito integrado. Un aspecto interesante es que *CoreConnect* permite reducir la carga capacitiva del *bus*, repartiéndola entre varios *buses.* Así, se consiguen mayores rendimientos, dado que los retardos de pistas globales son muy importantes en FPGAs. El estándar define varios tipos de *buses*, cada uno con unas características de velocidad y conectividad. A continuación se describen dos de los que se utilizan en Microblaze [30]:

LMB: (*Local Memory Bus*): *Bus* síncrono de alta velocidad, utilizado para conectar periféricos y los bloques de memoria interna de la FPGA. Solo admite un maestro en su implementación para MicroBlaze y puede ser utilizado tanto para instrucciones como para datos. Este *bus* es compatible con el PLB (*Processor Local Bus*) incluido en el estándar *CoreConnect,* pero a diferencia de éste, el LMB no admite varios maestros ni tamaños de palabra diferentes a 32 bits.

OPB: (*On-chip Peripheral Bus*): *Bus* síncrono utilizado para conectar periféricos con tiempos de acceso variables. Soporta varios maestros y la conectividad de muchos periféricos es sencilla gracias a su identificación por multiplexación distribuida. Soporta transferencias de tamaño de palabra dinámico.

Además de estos buses, existen otras alternativas para comunicarse con el procesador MicroBlaze. A continuación se describen varios protocolos de comunicación que se utilizan actualmente para comunicarse con el microprocesador:

FSL: (Fast Simple Link) protocolo que permite la comunicación con el procesador a través de sus propios registros internos. . El periférico actuaría a modo de co-procesador y la conexión se realiza de manera sencilla a través de unos registros de desplazamiento de 32 bits de ancho. La comunicación con estas FIFOs se realiza con dos instrucciones específicas del repertorio, que realizan las funciones de *push* y *pop* de estas memorias de desplazamiento. MicroBlaze soporta hasta 16 dispositivos conectados con este protocolo.

XCL: (Xilinx CacheLink) protocolo de comunicación para memorias externas. Este interfaz está diseñado para conectar directamente con un controlador de memoria que contiene buffer FSL. Este método se caracteriza por tener la menor latencia y mínimo número de instanciaciones.

	LMB	OPB	FSL
Ancho de banda máximo	500 MB/s	167 MB/s	800 MB/s
Número de periféricos	1	Limitado por el espacio de la FPGA	16

Tabla 17: Comparación de buses de MicroBlaze[30]

A.6Interrupciones y excepciones

El procesador Microblaze permite la utilización tanto de interrupciones como de excepciones. Para ello será necesario activar la línea de interrupciones que hará que el procesador ejecute la rutina de manejo de interrupciones. El manejo de excepciones tiene un manejo similar, cuando se produce una excepción, el flujo normal de instrucciones se paraliza y se ejecuta una rutina de manejo de excepciones. Puede ser necesario que el sistema necesite manejar más de una interrupción. Para ello será necesario utilizar un periférico específico, que permitirá multiplexar y reconocer las diferentes interrupciones. Este periférico se comunicara con el procesador mediante el bus OPB.

B Entorno de desarrollo EDK

En el caso de Microblaze, el diseño en VHDL o Verilog no está disponible para el usuario. En caso de querer integrarlo dentro de la FPGA será necesario hacerlo mediante una herramienta específica proporcionada por el fabricante. Esta herramienta se llama EDK (*Embedded Design Kit*)

Esta herramienta está formada por una serie de aplicaciones que permiten la configuración hardware y la inclusión de periféricos hardware en el diseño de procesador [12]. Sin embargo, esta herramienta no está completa sin el apoyo de la herramienta ISE, que permite realizar las tareas de emplazamiento y rutado.

A continuación se describen y analizan algunas de los archivos que utiliza la herramienta de entrono para configurar le sistema.

B.1Archivos de configuración hardware

Este archivo con extensión .mhs (*microprocessor hardware specification*) contiene las especificaciones de los puertos del sistema. Cada uno de estos puerto, además de ser identificado con un nombre, debe incluir el tipo de puerto (entrada o salida) y el número de bits que lo compone. En este fichero también se especifican las opciones de configuración del procesador, tales como los *buses* que componen el sistema. Otros parámetros a especificar son la utilización de recursos *hardware* de la FPGA, como por ejemplo divisores hardware (en caso de querer utilizar recursos de la FPGA para esta tarea), habilitación de cachés y tamaño de las mismas, y por último, habilitación y opciones del modo de solución de errores (*debug*).

Este archivo contiene las instanciaciones de los periféricos del sistema, así como la configuración de los mismos. Entre estas opciones figura el rango de memoria en el cual están mapeados estos periféricos. Estas pueden asignarse en un archivo de opciones hardware .mpd (*microprocessor peripheral description*) y descargar de contenido al archivo de configuración de hardware.

B.2Implementación del hardware.

Aplicando la herramienta de la herramienta EDK XPS Platform Generator (Platgen) sobre los ficheros MHS y MPD. Utilizando el fichero de salida del “Platgen” y los ficheros HDL del sistema en los cuales se ha definido los periféricos, se obtienen un fichero “netlist”.

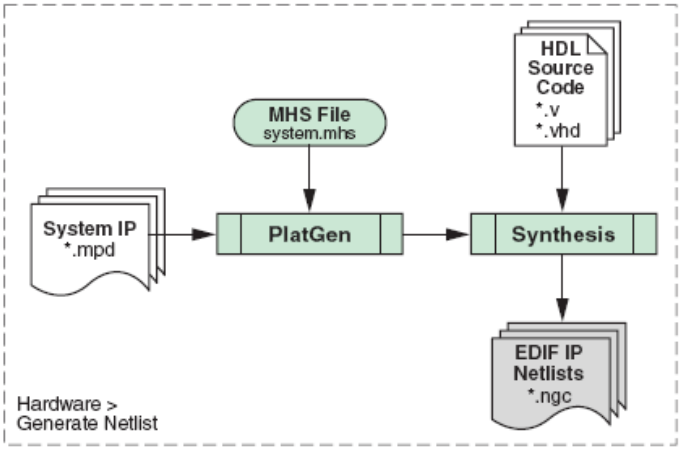


Figura 42: elementos y estados de generación del fichero netlist [12].

Una vez obtenido el fichero netlist será necesario generar el fichero bitstream (Bit File), el cual será descargado la propia placa. Para ello se sintetiza el fichero utilizando el fichero User Contrains File(UCF) definido por el diseñador mediante la herramienta EDK.

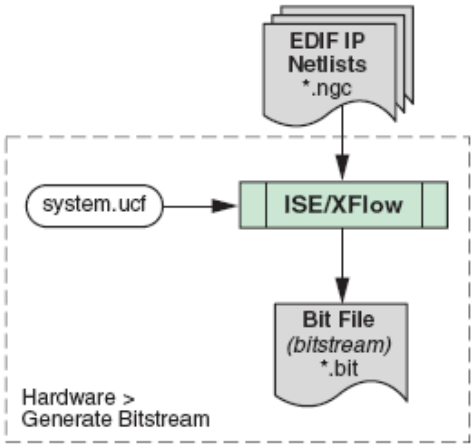


Figura 43: elementos y estados para la generación del fichero "bitstream"[12].

B.3Archivos de descripción software

Este archivo con extensión .mss (*microprocessor software specification*) contiene las opciones de compilación del software del sistema. Incluye la especificación del modo de compilación del código, asignación de librerías a periféricos, especificación del método de solución de errores, y otras opciones del compilador. Este archivo depende explícitamente del archivo de descripción de hardware, ya que debe de especificarse el controlador para cada uno de los periféricos instanciados. De igual forma, si el dispositivo donde se va a cargar el diseño utiliza recursos hardware específicos, por ejemplo un multiplicador, es en este archivo donde se debe indicar al compilador que utilice este recurso hardware.

Los controladores de los periféricos se encuentran en otros archivos específicos. Los .mdd (*microprocessor driver definition*) contienen el código necesario para el control de cada periférico. En la instanciación del controlador se debe especificar la versión del mismo, en caso de que existan varias y el nivel de abstracción con el que se va a utilizar el periférico en el código.

B.4Implementación de software

Una vez programa el procesador en código c, la herramienta EDK compila los ficheros mediante las rutinas de compilación, obteniendo un fichero ELF (*Executable and Linkabel Format*). Este fichero contiene la información de ejecución para el hardware en unos y ceros y será el fichero que se cargara en el procesador embebido en la FPGA.

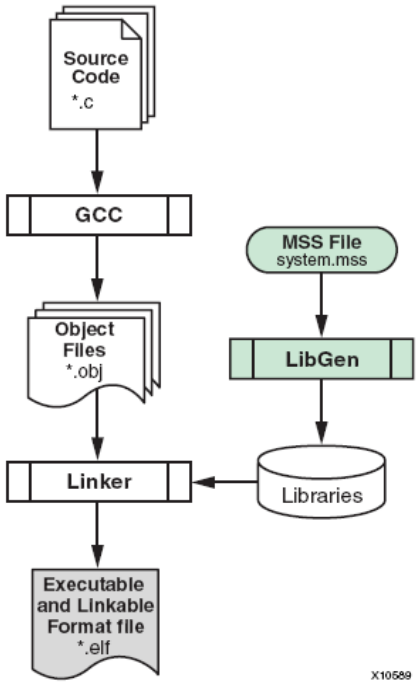


Figura 44: elementos y estados para la generación del fichero .elf [12]

B.5Código

En los archivos anteriormente mencionados, la sintaxis es muy simple ya que tan solo permiten configurar e instanciar periféricos y sus controladores. Con estos archivos consigo configurar la estructura del procesador y los periféricos agregados.

Para completar el diseño del procesador será necesario incluirle un código con las diversas instrucciones a realizar. Los archivos de código pueden escribirse tanto en C, C++ o ensamblador. No existe ninguna restricción a la hora de escribir código para un sistema MicroBlaze. Al ser un procesador de 32 bits soporta todos los tipos de datos. En el caso que el código del sistema esté diseñado para ser almacenado en diferentes zonas del mapa de memoria, por ejemplo código repartido entre la memoria interna y externa de la FPGA, es necesario escribir un *Linker Script* conteniendo las instrucciones necesarias para que se compile cada parte del código en el rango de direcciones correspondiente.

B.6Configuración de la FPGA

Para descargar la información de diseño sobre la placa, se debe conectar el ordenador a la placa mediante un cable JTAG. La información software se cargara en la BRAMs (*Internal Block RAM*). Esto no permitirá poder utilizarla también para debuggear el sistema en la propia placa.

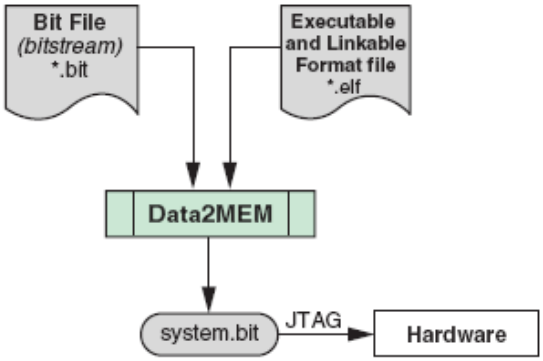


Figura 45: Elementos y estados para la generación del "bitstream" para la FPGA embebida [12].

A continuación se muestra el esquema de flujo total del sistema:

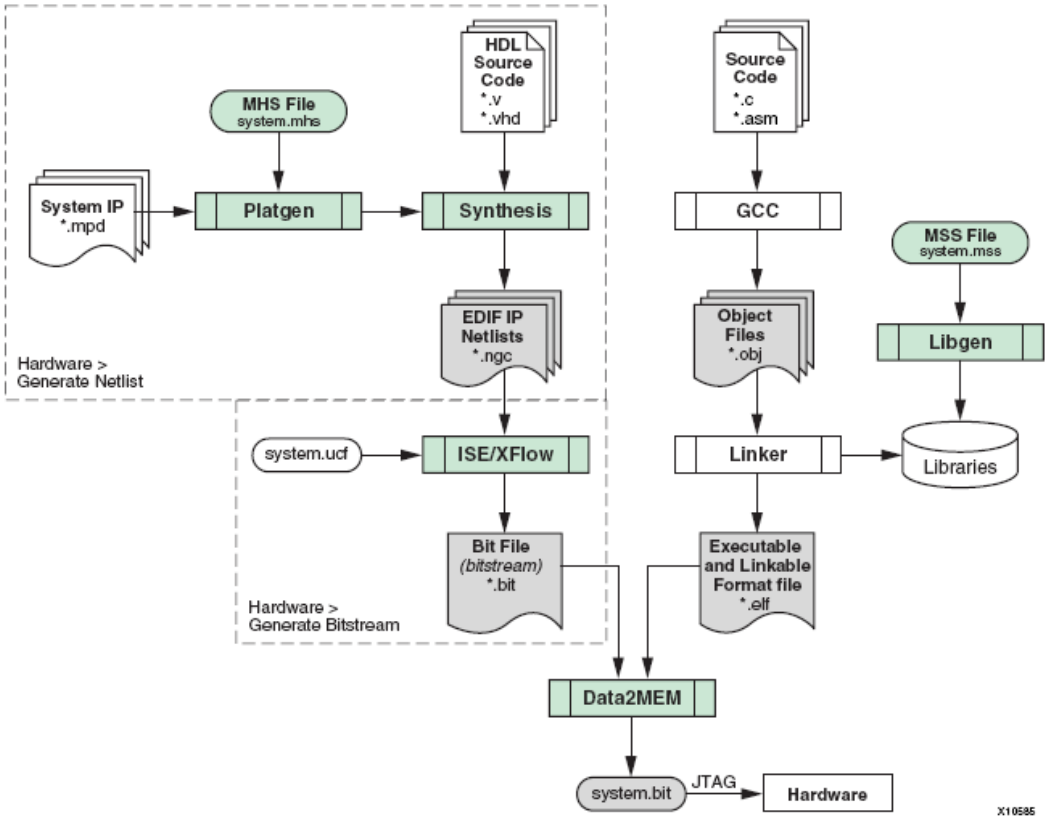


Figura 46: Elementos y estados totales para implementar y descargar un diseño [12].

C Diseño de periféricos

La herramienta EDK permite crear periféricos para instanciar dentro de nuestro procesador [12]. Para ello como ya se indicó con anterioridad es necesaria la herramienta ISE.

C.1Descripción lógica del periférico

El primer paso para el diseño de un periférico será describir el funcionamiento del mismo mediante un lenguaje hardware (VHDL,Verilog…). La conexión con el periférico se realizara mediante un bus (OPB o PLB). La elección del bus debe estar en consonancia con las especificaciones del sistema y del periférico, ajustándose así a sus necesidades tanto de consumo como de velocidad. También existe la posibilidad de conectarlo directamente a los registros internos de MicroBlaze con el protocolo FSL.

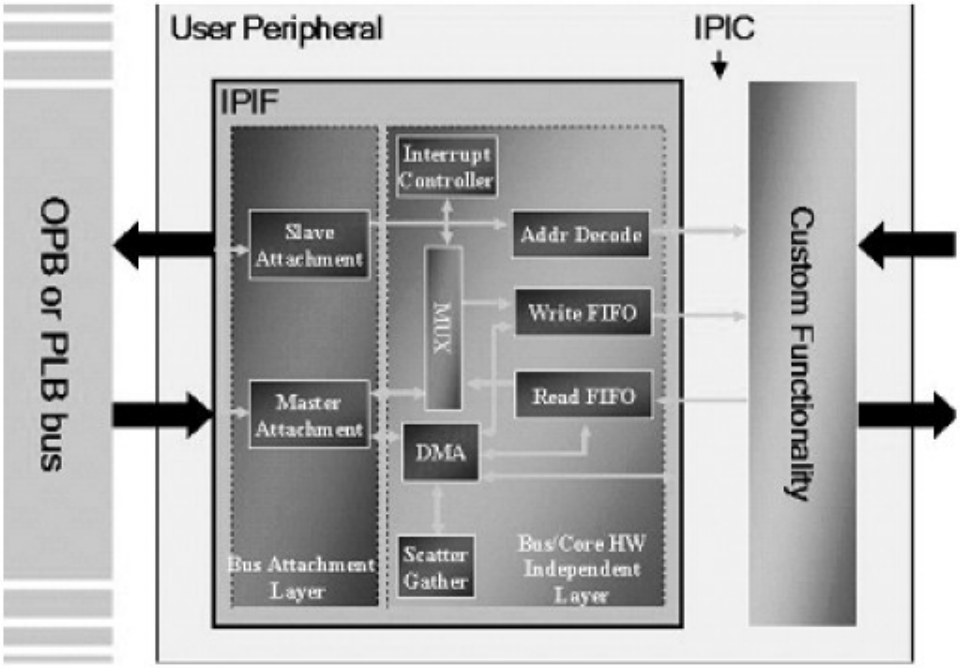


Figura 47: Descripción de la estructura del IPIF en la conexión con el microprocesador [11]

Para las conexiones de los periféricos Xilinx ofrece un *soft-macro* que maneja el protocolo necesario para la comunicación con el procesador. Estos bloques, denominados IPIF, ahorran al diseñador el estudio detallado de los diferentes protocolos.

C.2Compilación de los periféricos

Para incluir el periférico diseñado en las librerías del procesador, la herramienta EDK utiliza una aplicación llamada *Import Peripheral Wizard,* que se encarga de realizar una compilación del diseño. Las librerías que se han utilizado en el diseño deben ser

especificadas a esta herramienta. Esta operación se puede realizar seleccionando directamente las librerías en el menú de esta aplicación o bien mediante un archivo con extensión .pao (*peripheral analyze order*). Una vez compilado el diseño satisfactoriamente, se asignan las señales de los *buses* con las del periférico y posteriormente se especifican las diferentes opciones de configuración del periférico. Para almacenar los valores por defecto de estas opciones se requiere la creación de un archivo .mpd (*microprocessor peripheral definition*). Una vez compilado, EDK incluye al periférico dentro de la lista de posibles bloques a utilizar por MicroBlaze y solo habrá que instanciarlo para incluirlo en el diseño.

D Bus OPB

En un sistema embebido aparece la necesidad de conectarlo a un periférico, para ello será necesario diseñar un bus de datos y de direcciones que permita conectar el microprocesador con el periférico [10]. Con el objetivo de simplificar los diseños y facilitar la conexión de diferentes macros, Xilinx ha implementado varios buses de conexión basados en protocolos estándar de comunicación.

Las características básicas del bus OPB son:

Sistema completamente síncrono con el flanco de subida de la señal de reloj.

Bus de datos de 32 bits, bus de direcciones de 32 bits

Solo necesita un ciclo de reloj para la transferencia de datos entre el "master" OPB y el esclavo OPB.

Permite habilitaciones de bytes en el master.

Permite supresiones fuera de tiempo por parte del esclavo.

Soporta paradas determinada por el esclavo (retry)

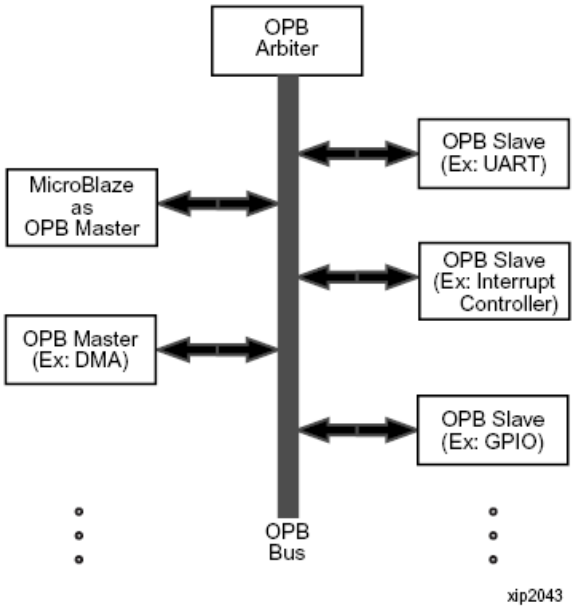


Figura 48: Descripción física de las conexiones a un bus OPB [10]

En la figura 48 se puede observar un diagrama de bloques que representan la estructura física de un bus OPB. Si se utiliza el entorno de desarrollo EDK de Xilinx, mediante el fichero MHS (Microprocessor Hardware Specification) el "PlatformGenerator" es capaz de determinar el número de elementos conectados al bus (tanto los master como los esclavos) y de instanciarlos, permitiendo al usuario abstraerse de los detalles de programación del estándar OPB. Como se puede observar el al figura 48, el bus OPB soporta más de un esclavo. El módulo OPB_v20 implementa el sistema de gateo de

puertas OR que permitirán seleccionar las señales que pasaran al bus OPB. Para controlar las conexiones de los mismos será necesario un árbitro. En el caso de disponer de un único esclavo, el sistema podría prescindir del árbitro, ya que los esclavos disponen de una cierta inteligencia.

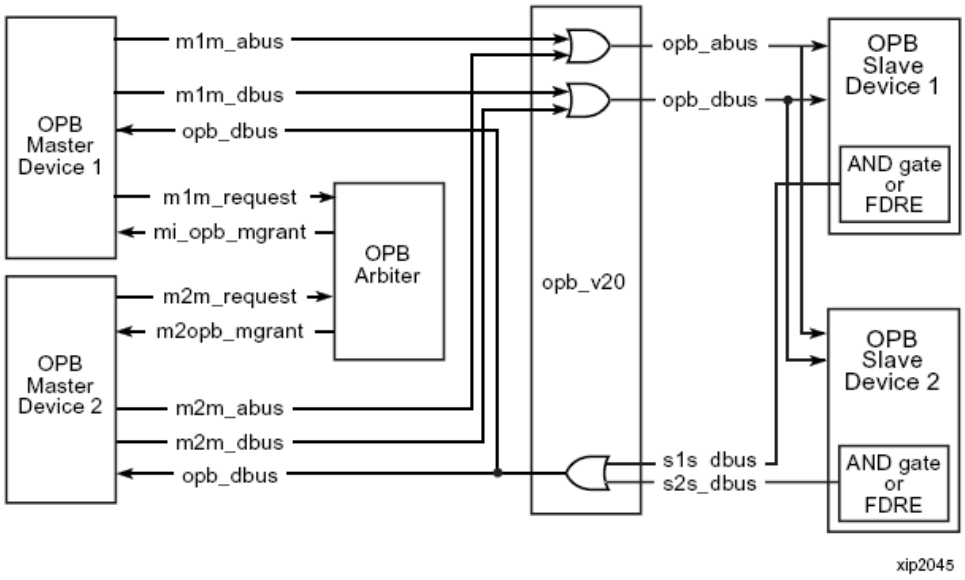


Figura 49: Ejemplo conexión de master y esclavos a un bus OPB [11]

Sin embargo, el diseño del esclavo con una conexión al bus OPB puede resultar costoso y repetitivo. Por este motivo, Xilinx incorpora un elemento en el esclavo del OPB que nos permitirá acceder y controlar el esclavo desde el nivel superior (el “firmware”). Para ello, la herramienta EDK de Xilinx permite crear un periférico incluyendo el módulo IPIF de conexión al OPB.

El módulo OPB IPIF fue concebido como un interfaz estándar que permite la conexión de periféricos al bus OPB para integrarlos dentro un sistema con un procesador MicroBlaze. Este módulo cumple dos funciones básicas en el diseño: facilita la conexión al bus OPB y ofrece un servicio ágil de conexión para diferentes tipos de IP. Además, este módulo es parametrizable, pudiendo determinar la funcionalidad y capacidad del mismo [11].

E Actuadores piezoeléctricos

Se conoce por “efecto piezoeléctrico” al fenómeno físico que produce que al aplicar una tensión entre dos caras de algunos materiales cristalinos se produzca un campo eléctrico en el cristal. Este efecto es también inverso, por lo tanto, los materiales piezoeléctricos se pueden utilizar para convertir energía eléctrica en energía mecánica y viceversa.

VENTAJAS DE LOS SISTEMAS DE POSICIONAMIENTO PIEZOELECTRICO:

- **RESOLUCION:** los actuadores piezoeléctricos permiten obtener cambios de posición fijos, de un orden menor del nanómetro.
- **GENERACION DE GRANDES FUERZAS:** algunos actuadores piezoeléctricos son capacites de producir varios miles de de newton de fuerza, llegando así a permitir desplazar toneladas.
- **EXPANSION RAPIDA:** tienen un tiempo de respuesta del orden de microsegundos.
- **NO PRODUCCION DE EFECTOS MAGNETICOS:** a pesar de producir campos magnéticos, los actuadores piezoeléctricos no producen campos magnéticos si se ven afectados por ellos. Por tanto, son sistema s muy utilices para aplicaciones en las cuales se deba de evitar la contaminación de campos magnéticos.
- **BAJA POTENCIA DE CONSUMO:** el actuador piezoeléctrico es un transductor activo, por lo tanto, tan solo absorbe la energía eléctrica que produce el movimiento, teniendo así un consumo muy reducido frente a otro tipo de actuadores.
- **LIBRE DE DESGASTES Y ROTURAS:** su desplazamientos e basa en la dinámica del estado solidó, y al no disponer de engranajes ni ejes, le sistema no muestra ni desgastes ni rotura.
- **COMPATIBLE CON CUARTOS LIMPIOS O EN VACIO:** son elementos que no necesitan lubricación y tampoco presentan desgaste o abrasión, por lo tanto se pueden utilizar en cualquier entorno.
- **OPERACIÓN A TEMPERATURAS CRIOGENICAS:** el efecto piezoeléctrico se basa en campos eléctricos, con lo cual, funcionan hasta llegar a los 0 grados Kevin, pero con las especificaciones reducidas.

OPERACIÓN DE LOS SISTEMAS PIEZOELECTRICOS:

Existen dos modos de operación de los actuadores piezoeléctricos, lazo abierto y lazo cerrado.

En los actuadores en lazo abierto el desplazamiento obtenido es proporcional al voltaje manejado. Sin embargo, este sistema proporciona un posicionamiento menos preciso que

el que se obtiene trabajando en modo lazo abierto y además, presenta problemas de histéresis y comportamiento “creep”. Para el control de la posición en este modo de trabajo será necesario incluir sensores externos que determinen la posición del sistema posicionado.

Los actuadores piezoeléctrico en modo lazo cerrado son ideales para posicionamientos precisos, ya que proporcional una alta linealidad, estabilidad en la posición, repetitividad y precisión. Este tipo de actuadores están equipados con sistemas de control de posicionamiento que permiten un control subnanométrica y un ancho de banda de hasta 10 KHz. El servo controlador determinara la potencia que debe enviar al piezoeléctrico hasta ajustarse a la posición adecuada.

F Estructura de ficheros del diseño hardware

Este anexo contiene la información relacionada con la estructura del diseño hardware que se obtienen con la utilización de la herramienta EDK de Xilinx. A continuación se muestra la estructura de ficheros de la cual se compone le proyecto [12]. La carpeta principal (en este caso con el nombre “instanciar_8pwm”) contiene una serie de ficheros de información general sobre el proyecto implementado:

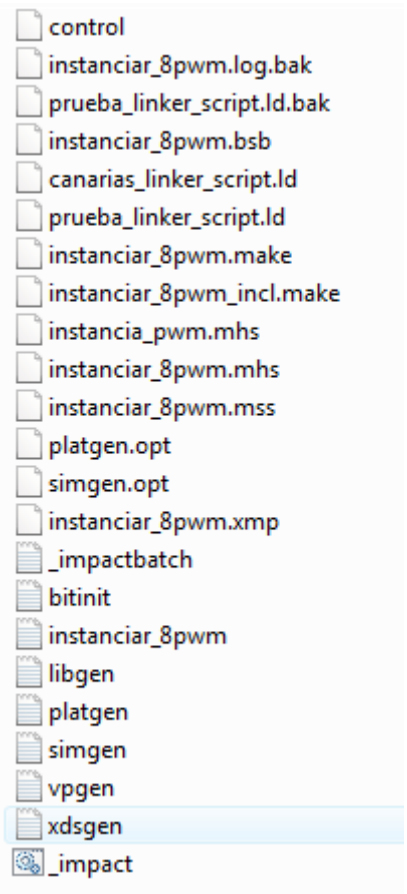


Figura 50: Ficheros contenidos en la carpeta del proyecto hardware

Estos ficheros serán generados automáticamente por la herramienta EDK. Los ficheros más significativos son:

instanciar_8pwm.xmp: este es el “top level” del proyecto EDK.

instanciar_8pwm.mhs: especificaciones hardware del proyecto.

instanciar_8pwm.mss: especificaciones software del proyecto.

Esta carpeta también contendrá subcarpetas en su interior una serie de carpeta dentro de la misma.

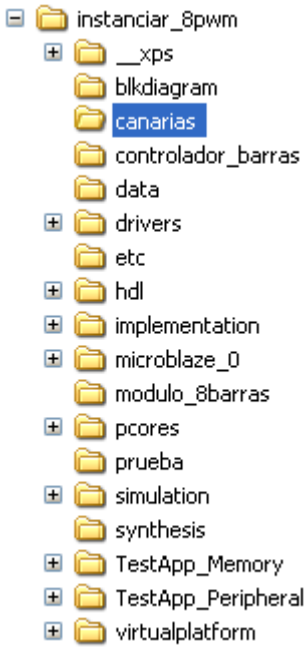


Figura 51: Directorio de carpetas contenidos en la carpeta del proyecto hardware

Las carpetas más significativas en el diseño:

Data: contiene el fichero UCF(*User Constrains File*)

Implentation: esta carpeta contiene los ficheros y careptas relativa a la información obtenida de la implementación. Los ficheros más destacados:

- **Instanciar_8pwm.par:** Fichero *Place & Route Report*.
- **Instanciar_8pwm.mrp:** Fichero *Map Report*.

TestApp_Memory: esta carpeta contiene los ficheros C y los ficheros cabecera del microprocesador.

Pcores: Esta carpeta contiene los periféricos disponibles en el proyecto y definidos mediante la herramienta ISE. En este caso el proyecto incluye tres periféricos:

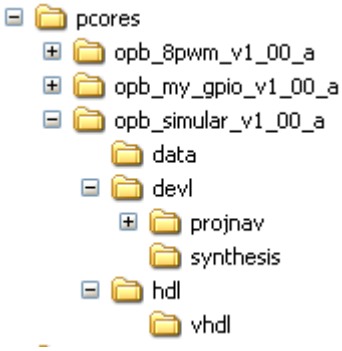
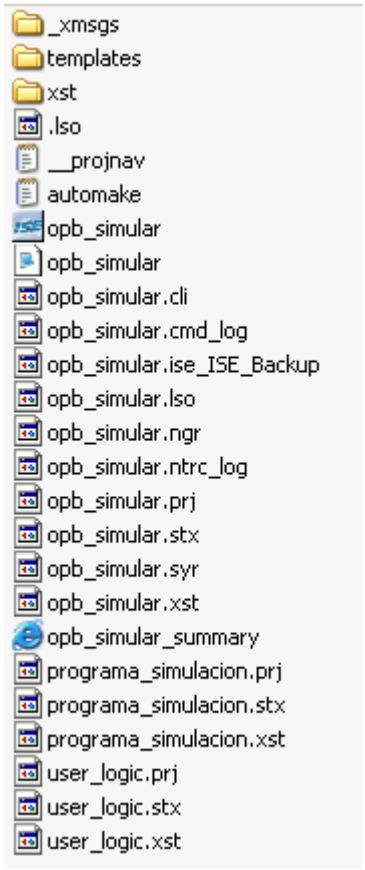


Figura 52: Directorio de carpetas contenidos en la carpeta pcores

Cada uno de los periféricos contendrá a su vez tres carpetas:

- La carpeta “data” contendrá el fichero MPD (*Microprocessor Peripheral Description*) que describe los pines del periférico y el fichero PAO (*Peripheral Analysis Order*) que contiene todos los periféricos que deben ser compilados para sintetizar el periférico y el orden en el que deben compilarse.
- La carpeta “devl” contiene dos carpetas. La carpeta “projnav” contiene el proyecto ISE en el cual se ha diseñado el periférico. Este proyecto contendrá el “opb_simular_v_1_00.ise” y todos los ficheros VHDL que describen el OPIF del bus OPB. La carpeta “synthesis” contiene información para la sintonización del proyecto ISE en la herramienta EDK.



- La carpeta HDL contiene los ficheros VHDL del periférico. El fichero “opb_simular” describe el “top” del diseño, a partir del cual se instancia el resto de los módulos del periférico. El fichero “user_logic” es un fichero VHDL generado automáticamente por la herramienta EDK al incluir el IPIF de conexión del OPB. El fichero “programa_simulacion” describe cada módulo individual de simulación :

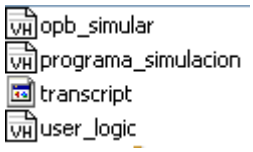


Figura 53: ficheros contenidos en la carpeta HDL de los periféricos contenidos en pcores.

G Comunicación puerto serie en Java

Para la comunicación por el puerto serie de la FPGA será necesario incluir en nuestro programa el paquete del API de comunicaciones de java (package javax.comm [9]) compatible con JDK1.2. Este paquete contiene:

Interfaces: CommDriver, CommPortOwnershipListener,ParallelPortEventListener, SerialPortEventListener

Clases: CommPort, CommPortIdentifier, ParallelPort, ParallelPortEvent , SerialPort, SerialPortEvent

Excepciones: NoSuchPortException, PortInUseException, UnsupportedCommOperationException

Mediante la utilización del API de comunicaciones se puede describir una clase "SerialPortJava" que contenga una cadena de bytes de entrada al puerto serie, otra que contendrá la cadena de salida, un puerto serie (con el que se realiza la comunicación).

```
public class SimpleSerialJava implements SimpleSerial {
    DataInputStream     m_DIS = null;
    DataOutputStream    m_DOS = null;
    SerialPort          m_SerialPort = null;
    boolean             m_BeenWarned = false;
```

Constructores de un nuevo puerto serie en el programa. El constructor por defecto describirá un puerto sin paridad y a 9600 baudios:

```
SimpleSerialJava(int comPort) {
  _initPort(comPort, 9600, 8, ONESTOPBIT, NOPARITY);
}
SimpleSerialJava(int comPort, int baud, int dataBits, int stopBits, int
parity) {
  _initPort(comPort, baud, dataBits, stopBits, parity);
}
```

La clase también incluye una serie de funciones para la transmisión de datos mediante el puerto serie. La función "_initPort" permite inicializar las características del puerto serie. Es utilizada por uno de los constructores de esta clase:

```
public void _initPort(int comPort, int baud, int dataBits, int stopBits,
int parity) {

        CommPortIdentifier  cpi = null;
        CommPort            cp = null;

          System.out.println("Initing JAVA port.  Com = " + comPort + ",
baud = " + baud);
```

```
        try {
            cpi = CommPortIdentifier.getPortIdentifier("COM" + comPort);
        }
        catch (NoSuchPortException e) {
            System.out.println("#### ERROR:  no such port: (" + comPort +
")");
            return;
        }

        try {
            cp = cpi.open("SimpleSerial", 1000);
        }
        catch (PortInUseException e) {
            System.out.println("Port in use");
            return;
        }

        if (cp instanceof SerialPort) {
            m_SerialPort = (SerialPort)cp;

            try {
                byte stopBitsLookup[] = {1, 3, 2};

                        m_SerialPort.setSerialPortParams(baud,  dataBits,
stopBitsLookup[stopBits], parity);
                 m_SerialPort.setFlowControlMode(SerialPort.FLOWCONTROL_NO
NE);
            }
            catch (UnsupportedCommOperationException e) {
                    System.out.println("####  ERROR:   Unsupported  comm
operation exception");
                return;
            }

            try {
                                                        m_DIS    =    new
DataInputStream(m_SerialPort.getInputStream());
                                                        m_DOS    =    new
DataOutputStream(m_SerialPort.getOutputStream());
            }
            catch (IOException e) {
                    System.out.println("###  ERROR:   Could't  open  data
stream");
                m_DIS = null;
                m_DOS = null;
            }
        }
    }
```

La función “close” permite cerrar el puerto serie definido:

```
    public void close() {
        if (m_SerialPort != null) {
            m_SerialPort.close();
        }
        m_SerialPort = null;
    }
```

A continuación se detallan las funciones relativas al envió de datos mediante le puerto serie definido:

La función “writeByte” permite enviar un byte por el puerto serie de la clase.

```
    public boolean writeByte(byte byteVal) {
        try {
            m_DOS.writeByte(byteVal);
            return true;
        }
        catch (IOException e) {
            System.out.println("### IO ERROR WRITING BYTE");
            System.out.println("### error is:  " + e);
            return false;
        }
    }
```

La función “readByte” para leer un byte del puerto serie:

```
    public int readByte() {
        try {
            return m_DIS.readByte();
        }
        catch (IOException e) {
            System.out.println("### IO ERROR READING BYTE");
            System.out.println("### error is:  " + e);
            return 0;
        }
    }
```

Las funciones “getOutputStream” y “getInputStream” para devolver la cadena leídas y enviadas en la clase. La función “isValid” permite determinar si la cadena a enviar no está vacía y la función “avaible” determina si las cadenas a enviar son viables:

```
    public OutputStream getOutputStream() {
        return m_DOS;
    }

    public InputStream getInputStream() {
        return m_DIS;
    }

    public boolean isValid() {
        return (m_DOS != null && m_DIS != null);
    }

    public int available() {
        try {
            return m_DIS.available();
        }
        catch (IOException e) {
                    System.out.println("###  ERROR:   Got  IOException  in
avaialable");
            m_DIS = null;
            return -1;
```

```
        }
    }
```

La función “readBytes” permite leer un conjunto de byte.

```
public byte[] readBytes() {

        String aux;

        try {
            int available = m_DIS.available();
            if (available > 0) {
                byte        data[] = new byte[available];

                m_DIS.read(data);
                aux=new String (data);
                System.out.println(aux);
                return data;
            }
            return new byte[0];
        }
        catch (IOException e) {
            System.out.println("### IO ERROR reading multiple bytes");
            System.out.println("### Error is:  " + e);

            return new byte[0];
        }
    }
```

La función “readString” permite leer un string.

```
  public String readString() {
        int     ii;

        byte data[] = readBytes();

        if (!m_BeenWarned) {
            for (ii = 0; ii < data.length; ii++) {
                if (!m_BeenWarned && data[ii] < 0) {
                    m_BeenWarned = true;
                          System.out.println("--> #### WARNING:  You are
reading string data with values less than zero.");
                     System.out.println("--> #### This can be dangerous as
Char->Byte remapping can change negative values!");
                       System.out.println("--> #### It's MUCH safer to use
readBytes[] instead");
                        System.out.println("--> #### You will only receive
this warning ONCE");
                    System.out.println("--> ####");
                }
            }
        }

        return new String(data);
    }
```

H Descripción de los módulos hardware

H.1Módulo simple pwm

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


ENTITY pwm_n IS

  generic
  (
    DATA_WIDTH                    : integer             := 24;
    LENGTH_WIDTH                  : integer             := 32
  );
PORT(
   clk : IN std_logic;
   rst : IN std_logic;
   capData : IN std_logic;
   ce  : IN std_logic;
   disp : IN std_logic;
   disp_ack : out std_logic;
   pwm_dat : in std_logic_vector(DATA_WIDTH-1 downto 0);
   pwm_lenght : in std_logic_vector(LENGTH_WIDTH-1 downto 0);
   pwm_out : out std_logic);
END pwm_n;


ARCHITECTURE behavorial OF pwm_n IS

signal count_dat: std_logic_vector (DATA_WIDTH-1 downto 0) := x"00000000";
signal ce_aux: std_logic:= '0';
signal func: std_logic;

-- inicializacion del sistema a inactivo
signal pwm_lenght_int: std_logic_vector (LENGTH_WIDTH-1 downto 0) := x"00000000";
signal pwm_dat_int: std_logic_vector (DATA_WIDTH-1 downto 0) := x"000000";

BEGIN


pwm:process (rst, clk)
begin

   -- reset asincrono del sistema
   if(rst='1') then
      func <= '0';
      count_dat <= (others => '0');
      disp_ack <= '0';
      pwm_out <= '0';
      ce_aux <= '0';

   elsif clk'event and clk ='1' then

      -- carga de datos de entrada
      if capData = '1' then
         pwm_dat_int <= pwm_dat;
         pwm_lenght_int <= pwm_lenght;
      end if;

      -- inicializacion de los datos en funcion del modo de trabajo
```

```
     if disp = '1' and ce='0' then
        func <= '1';
        disp_ack <= '1';
        count_dat <= (others => '0');
        ce_aux<='0';

     elsif disp = '1' and  ce = '1' then
        disp_ack <= '1';
        count_dat <= (others => '0');
        func <= '0';
        ce_aux<='1';

     -- contador de control de la longitud del pwm
     elsif (ce_aux = '1' and disp='0') or func = '1' then
           disp_ack <= '0';
           count_dat <= count_dat + '1';
           --tiempo en activo del pulso
           if count_dat = pwm_lenght_int then
              count_dat <= (others => '0');
           end if;

           if count_dat < pwm_dat_int then
              pwm_out <= '1';
           else
              pwm_out <= '0';
              func <= '0';
           end if;

     end if;

  end if;

end process pwm;


END behavorial;
```

H.2Módulo simple de simulación

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity programa_simulacion is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           cap: in STD_LOGIC;
           cap_aux: out STD_LOGIC;
           pwm : in STD_LOGIC;
           posicion_inicial : in  STD_LOGIC_VECTOR (31 downto 0);
           retardo   : in STD_LOGIC_VECTOR (7 downto 0);
            dir : in  STD_LOGIC;
           vuelta: in  STD_LOGIC_VECTOR (7 downto 0);
```

```
           dat : out  STD_LOGIC_VECTOR (31 downto 0));
end programa_simulacion;

architecture Behavioral of programa_simulacion is

   constant precision: integer := 32;

   -- valores de las constantes
   constant   desplazamiento_50:   std_logic_vector   (precision-1   downto   0)   :=
x"000000C8";
   constant   desplazamiento_40:   std_logic_vector   (precision-1   downto   0)   :=
x"0000009B";
   constant   desplazamiento_30:   std_logic_vector   (precision-1   downto   0)   :=
x"0000006E";
   constant   desplazamiento_20:   std_logic_vector   (precision-1   downto   0)   :=
x"00000041";
   constant   desplazamiento_10:   std_logic_vector   (precision-1   downto   0)   :=
x"00000014";

   constant posicion_max: std_logic_vector (31 downto 0) := x"00000898";
   signal posicion: std_logic_vector (31 downto 0) := x"00000000";
   signal posicion_aux: std_logic_vector (31 downto 0) := x"00000000";
   signal desplazamiento: std_logic_vector (31 downto 0) := x"00000000";

   -- para retardar los pulsos y contabilizar las vueltas
   signal retar : std_logic:= '0';
   signal cont_retardo : std_logic_vector (31 downto 0) := x"00000000";
   signal vuelta_cont : std_logic_vector (7 downto 0) := x"00";
   signal cuenta : std_logic_vector (31 downto 0) := x"00000000";
   signal cnt : std_logic_vector (31 downto 0) := x"00000000";
   signal aux : std_logic := '0';
   signal act : std_logic_vector(31 downto 0):= x"00000000";

begin

-- para conocer que desplazamiento está asignado a esta posicion
   mult:process (cuenta)
   begin
      if cuenta=x"00000000" then
         desplazamiento<=x"00000000";
      else
         if (cuenta = x"00000032") then
            desplazamiento<=desplazamiento_50;
         -- al 20 %
         elsif (cuenta=x"00000014") then
            desplazamiento<=desplazamiento_10;
         -- al 30 %
         elsif (cuenta=x"0000001E") then
            desplazamiento<=desplazamiento_30;
         -- al 40 %
         elsif (cuenta=x"00000028") then
            desplazamiento<=desplazamiento_40;
         -- al 10 %
         elsif (cuenta=x"0000000A") then
            desplazamiento<=desplazamiento_20;
         else
            desplazamiento<=x"00000000";
         end if;
      end if;
   end process;

basic:process (clk)
   begin
      -- resetear el sistema
      if rst='1' then
         posicion<=posicion_inicial;
         cont_retardo<=x"00000000";
```

```
         vuelta_cont<=vuelta;
      elsif clk'event and clk ='1' then
         -- cargar los datos iniciales de simulacion
         if cap='1' then
            cap_aux<= '1';
            vuelta_cont<=vuelta;
            posicion<=posicion_inicial;
         end if;

         -- producir el retardo de movimiento
         if retar='1' then
               if cont_retardo=retardo then
                  cont_retardo<=x"00000000";
                  retar<='0';
                  dat<=posicion;
               else
                  cont_retardo <= cont_retardo + '1';
               end if;
         end if;

         --
         if pwm = '1' then
            cuenta<=x"00000000";
            if aux='1' then
               cnt<= cnt+1;
            else
               cnt<= x"00000000";
            end if;
            aux<='1';
            act<=x"00000000";
         else
            cuenta<=cnt+1;
            aux <='0';
            act<=act+1;
            if act=x"00000002" then
            --actualizar la posicion del sistema
               if dir ='1' then
                  if (posicion < desplazamiento and  vuelta_cont >"00")then
                     posicion <= posicion_max + posicion - desplazamiento;
                     vuelta_cont <= vuelta_cont-1;
                  elsif posicion >= desplazamiento then
                     posicion <= posicion - desplazamiento;
                  end if;
                  retar<='1';
               else
                  posicion<= posicion + desplazamiento;
                  if (posicion > posicion_max) then
                     posicion <= posicion - posicion_max;
                     vuelta_cont <= vuelta_cont+1;
                  end if;
                  retar<='1';
               end if;
            end if;
         end if;
      end if;
   end process;

end Behavioral;
```

I Descripción del código software para el microprocesador

I.1 Funciones para control de barras.

```
/*********************************************************************
Fichero :
*  Marina Aparicio Rodriguez
*  28-07-2007
*  Descripcion: definicion de las funciones de control y actuacion sobre el
*               sistema de barras pareadas del sistema
*********************************************************************/

#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"
#include "opb_8pwm.h"
#include "canarias.h"

/*********************************************************************
*  actualizarIzquierda: actualiza la posicion de una barra izquierda
*  entradas:   barra : estructura que contiene la posicion de la barra
*              dir:  direccion de movimiento de la barra
*              direccion_lectura: posicion de lectura de la barra.
*********************************************************************/

int actualizarIzquierda(Posicion_barra * barra,int dir,Xuint32 direccion_lectura)
{
      Xuint16 posicion_nueva;

      //ver la salida actual del sistema
      posicion_nueva=(Xuint16)XIo_In32(direccion_lectura);

      //actualizar la vuelta en la que se encuentra
      if(posicion_nueva<=barra->precision && dir==1)
      {
         barra->precision=posicion_nueva;
      }
      else if(posicion_nueva < barra->precision && dir==0)
      {
         barra->vuelta++;
         barra->precision=posicion_nueva;
      }
      else if(posicion_nueva > barra->precision && dir==1)
      {
         //no permite estar en vueltas negativas
         if (barra->vuelta==0)
            barra->precision=0x0000000;
         else
         {
            barra->vuelta--;
            barra->precision=posicion_nueva;
         }
      }
      else if(posicion_nueva>=barra->precision && dir==0)
      {
         barra->precision=posicion_nueva;
      }

}

/*********************************************************************
*  actualizarDerecha: actualiza la posicion de una barra derecha
```

```
*  entradas:   barra : estructura que contiene la posicion de la barra
*              dir:  direccion de movimiento de la barra
*              direccion_lectura: posicion de lectura de la barra.
**********************************************************************/

int actualizarDerecha(Posicion_barra * barra,int dir,Xuint32 direccion_lectura)
{
      Xuint16 posicion_nueva;

      //ver la out actual del sistema
      posicion_nueva=(Xuint16) XIo_In32(direccion_lectura);

      //actualizar la vuelta en la que se encuentra
      if(posicion_nueva<=barra->precision && dir==1)
         barra->precision=posicion_nueva;
      else if(posicion_nueva < barra->precision && dir==0)
      {
         barra->vuelta++;
         barra->precision=posicion_nueva;
      }
      else if(posicion_nueva > barra->precision && dir==1)
      {
         //no permite estar en vueltas negativas
         if (barra->vuelta==0)
            barra->precision=0x0000000;
         else
         {
            barra->vuelta--;
            barra->precision=posicion_nueva;
         }

      }
      else if(posicion_nueva >= barra->precision && dir==0)
         barra->precision=posicion_nueva;

}

/**********************************************************************
*  controlDerecha: actualiza los parametros de movimiento para una barra derecha
*  entradas:   drch : estructura que contiene la posicion de la barra derecha
*              drch_ideal  :  estructura  que  contiene  la  posicion  al  que  debemos
llegar
*              dir:  direccion de movimiento de la barra
*              flanco: tiempo en pulsos de reloj que debe estar el pwm activado.
**********************************************************************/

int control(Posicion_barra * barra,Posicion_barra * barra_ideal,Xuint32 * flanco,int
* dir)
{
   //margenes de trabajo de control
   int sup,inf;

   sup= barra_ideal->precision + PRECISION;

   if (barra_ideal->precision ==0)
      inf=0;
   else
      inf= barra_ideal->precision - PRECISION;

      if(barra->vuelta==barra_ideal->vuelta)
      {
         //cuidado esta condicion en el cero
         if (barra->precision <= sup && barra->precision >= inf)
         {
            *flanco=DATA_PARADA;
            *dir=0;
            return 1;
```

```
        }
        else if (barra->precision < barra_ideal->precision)
        {
           *flanco=DATA_PRECISION;
           *dir=0;
        }
        else
        {
           *flanco=DATA_PRECISION;
           *dir=1;
        }
     }
     else if(barra->vuelta < barra_ideal->vuelta)
     {
        *flanco=DATA_BURDO;
        *dir=0;
     }
     else
     {
        *flanco=DATA_BURDO;
        *dir=1;
     }

     return 0;
}


/*********************************************************************
* Reset: esta funcion mueve todas las barras que entran como parametro de entrada a
los extremos.
* entrada:    barra-> tabla que contiene la estructura Posicion_barra se 4 pares
*             de barras las impares son las barras izquierdas y las pares las
derechas.
*********************************************************************/

int reset(Posicion_barra * barra)
{
   Xuint32 habilitacion,aux,dir_aux;
   Xuint32 cont_time;
   int opc = 1,i;
   int time=0;
   int cont_inac=0;
   int aux_1;

   // incluir un margen de variacion
   int sup,inf;

   /*************************************************/
   //             Inicializar timer
   /*************************************************/

   //activar generate mode
   //desabilutada señal de salida
   //desabilitar trigger
   //cuenta hacia arriba
   XIo_Out32(timer_TCSR0, 0x00000080);
   XIo_Out32(timer_TLR0, 0x00000000);


   // variables del sistema
   Xuint32 posicion_anterior[8],posicion_actual[8];

   habilitacion=0x0000FF00 + DISP;


   //fijar las constantes del sistema
   aux=0xFF000032;
```

```
dir_aux=DIR;

sup= CALIBRACION + PRECISION;
inf= CALIBRACION - PRECISION;

//inicializar las variables de pwm
XIo_Out32(baseaddr_pwm_total, 0x00000060);
XIo_Out32(baseaddr_pwm_dat, aux);
XIo_Out32(baseaddr_pwm_dir, dir_aux);

// inicializar las variables para la simulacion
XIo_Out32(baseaddr_retardo, 0x00000000);
XIo_Out32(baseaddr_vuelta, 0x80000006);
XIo_Out32(baseaddr_vuelta+4, 0x80000008);
XIo_Out32(baseaddr_vuelta+8, 0x80000006);
XIo_Out32(baseaddr_vuelta+12, 0x80000008);
XIo_Out32(baseaddr_vuelta+16, 0x80000008);
XIo_Out32(baseaddr_vuelta+20, 0x80000008);
XIo_Out32(baseaddr_vuelta+24, 0x80000008);
XIo_Out32(baseaddr_vuelta+28, 0x80000008);

// activar la señal el pwm
XIo_Out32(baseaddr_pwm_abilitacion, habilitacion);

for(i=0;opc!=0;i++)
{
   for (aux_1=0;aux_1<NUM;aux_1++)
      posicion_actual[aux_1]=XIo_In32(baseaddr_my_gpio+aux_1*4);


   // poner que permita un cierto margen

   if (posicion_actual[0]==posicion_anterior[0] &&
      posicion_actual[1]==posicion_anterior[1] &&
      posicion_actual[2]==posicion_anterior[2] &&
      posicion_actual[3]==posicion_anterior[3] &&
      posicion_actual[4]==posicion_anterior[4] &&
      posicion_actual[5]==posicion_anterior[5] &&
      posicion_actual[6]==posicion_anterior[6] &&
      posicion_actual[7]==posicion_anterior[7] && i!=0)
   {
      cont_inac++;
   }
   else
   {
      for (aux_1=0;aux_1<NUM;aux_1++)
         posicion_anterior[aux_1]=posicion_actual[aux_1];

      cont_inac=0;
   }

   if (cont_inac > 10)
   {
      opc=0;
   }

}

cont_time= XIo_In32(timer_TCR0);
time=((int)cont_time+2);

//xil_printf("tiempo tardado en resetear-> %d\r\n",time);
//xil_printf("i-> %d\r\n",i);

//Mostrar el reset
//codigo de reset
//xil_printf("**********REALIZANDO UN RESET*********\r\n");
```

```
   xil_printf("cd:reset\n");


   // actualizar los dat en la posicion y vuelta
   for (aux_1=0;aux_1<NUM;aux_1++)
      barra[aux_1].vuelta=0;
   //asi ya te dice la posicion en la que se estabiliza

   for (aux_1=0;aux_1<NUM;aux_1++)
      barra[aux_1].precision=posicion_actual[aux_1];


   //parar el sistema



   XIo_Out32(baseaddr_pwm_total, 0x00000060);
   XIo_Out32(baseaddr_pwm_dat, aux);
   XIo_Out32(baseaddr_pwm_dir, dir_aux);
   habilitacion=0x00000000 + DISP;
   XIo_Out32(baseaddr_pwm_abilitacion, habilitacion);

   dir_aux=0x00000000;
   XIo_Out32(baseaddr_pwm_dir, dir_aux);
}
```

I.2 Función principal del procesador.

```
//====================================================
//            LIBRERIAS
//====================================================

#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"
#include "opb_8pwm.h"
#include "canarias.h"

//====================================================
//          FUNCIONES CONVERSION HEXADECIMAL
//====================================================


Xuint32 letraPrimera(int dat1,int dat2,int dat3, int dat4)
{
   if (dat1==0 && dat2==0 && dat3==0 && dat4==0)
      return 0x00000000;
   else if (dat1==0 && dat2==0 && dat3==0 && dat4==1)
      return 0x10000000;
   else if (dat1==0 && dat2==0 && dat3==1 && dat4==0)
      return 0x20000000;
   else if (dat1==0 && dat2==0 && dat3==1 && dat4==1)
      return 0x30000000;
   else if (dat1==0 && dat2==1 && dat3==0 && dat4==0)
      return 0x40000000;
   else if (dat1==0 && dat2==1 && dat3==0 && dat4==1)
      return 0x50000000;
   else if (dat1==0 && dat2==1 && dat3==1 && dat4==0)
      return 0x60000000;
   else if (dat1==0 && dat2==1 && dat3==1 && dat4==1)
      return 0x70000000;
   else if (dat1==1 && dat2==0 && dat3==0 && dat4==0)
      return 0x80000000;
```

```
   else if (dat1==1 && dat2==0 && dat3==0 && dat4==1)
      return 0x90000000;
   else if (dat1==1 && dat2==0 && dat3==1 && dat4==0)
      return 0xA0000000;
   else if (dat1==1 && dat2==0 && dat3==1 && dat4==1)
      return 0xB0000000;
   else if (dat1==1 && dat2==1 && dat3==0 && dat4==0)
      return 0xC0000000;
   else if (dat1==1 && dat2==1 && dat3==0 && dat4==1)
      return 0xD0000000;
   else if (dat1==1 && dat2==1 && dat3==1 && dat4==0)
      return 0xE0000000;
   else if (dat1==1 && dat2==1 && dat3==1 && dat4==1)
      return 0xF0000000;
}

Xuint32 letraSegunda(int dat1,int dat2,int dat3, int dat4)
{
   if (dat1==0 && dat2==0 && dat3==0 && dat4==0)
      return 0x00000000;
   else if (dat1==0 && dat2==0 && dat3==0 && dat4==1)
      return 0x01000000;
   else if (dat1==0 && dat2==0 && dat3==1 && dat4==0)
      return 0x20000000;
   else if (dat1==0 && dat2==0 && dat3==1 && dat4==1)
      return 0x03000000;
   else if (dat1==0 && dat2==1 && dat3==0 && dat4==0)
      return 0x04000000;
   else if (dat1==0 && dat2==1 && dat3==0 && dat4==1)
      return 0x05000000;
   else if (dat1==0 && dat2==1 && dat3==1 && dat4==0)
      return 0x06000000;
   else if (dat1==0 && dat2==1 && dat3==1 && dat4==1)
      return 0x07000000;
   else if (dat1==1 && dat2==0 && dat3==0 && dat4==0)
      return 0x08000000;
   else if (dat1==1 && dat2==0 && dat3==0 && dat4==1)
      return 0x09000000;
   else if (dat1==1 && dat2==0 && dat3==1 && dat4==0)
      return 0x0A000000;
   else if (dat1==1 && dat2==0 && dat3==1 && dat4==1)
      return 0x0B000000;
   else if (dat1==1 && dat2==1 && dat3==0 && dat4==0)
      return 0x0C000000;
   else if (dat1==1 && dat2==1 && dat3==0 && dat4==1)
      return 0x0D000000;
   else if (dat1==1 && dat2==1 && dat3==1 && dat4==0)
      return 0x0E000000;
   else if (dat1==1 && dat2==1 && dat3==1 && dat4==1)
      return 0x0F000000;
}



Xuint32 resultadoDireccion(int * dir_int)
{
   Xuint32 dato1;
   Xuint32 dato2;
   Xuint32 salida;

   dato1=letraPrimera(dir_int[0],dir_int[1],dir_int[2],dir_int[3]);
   dato2=letraSegunda(dir_int[4],dir_int[5],dir_int[6],dir_int[7]);

   salida=dato1+dato2;

   return salida;
}
```

```
//=====================================================
//            MAIN
//=====================================================

int main (void) {


  /*************** variables ***********************/

  Xuint32 flanco=0x00000000;
  Xuint32 dir_aux=0x00000000,dir_anterior=0xff000000;
  Xuint32 flanco_aux=0x00000000;
  Xuint32 enable_aux=0x00000000;
  Xuint16 precision,posicion_nueva;
  Xuint8 vuelta;
  Xuint32 disp[8];
  Xuint32 cap[8];
  Xuint32 dir[8];
  Xuint32 dir_ant[8];
  Xuint32 flanco_ant[8];
  int aux,i=0,j=0,control_parada=0;
  char vuelta_string[4],precision_string[4],cargar_datos_v[2],cargar_datos_p[3];
  Xuint8 vuelta_dato[8],precision_dato[8];

  // añadido para controlar bien las direcciones

  int dir_int[8];

  /*************** definicion de las barras *********************/

  // considero que:
  //    - las barras con indice par se corresponden con las barras izquierdas
  //    - las barras con indice impar se corresponden con las barras derechas

  Posicion_barra barra_next[8];
  Posicion_barra barra_now[8];

  /*************** definicion tablas de constantes*****************/

  //disparo
  disp[0]=DISP_1;
  disp[1]=DISP_2;
  disp[2]=DISP_3;
  disp[3]=DISP_4;
  disp[4]=DISP_5;
  disp[5]=DISP_6;
  disp[6]=DISP_7;
  disp[7]=DISP_8;

  //captura
  cap[0]=CAP_1;
  cap[1]=CAP_2;
  cap[2]=CAP_3;
  cap[3]=CAP_4;
  cap[4]=CAP_5;
  cap[5]=CAP_6;
  cap[6]=CAP_7;
  cap[7]=CAP_8;


  /************ reset de las barras del sistema *****************/

  reset(barra_now);
```

```
// la idea seria hacer un reset inicial y solo repetirlo en caso //de error con al
funcion de control de error



  /**********BUCLE INFINITO DE CONTROL DEL SISTEMA*************/
  for(;;){

        /** Inicializacion de datos de la futura posicion**/

        //tomar los datos de lectura

        for(j=0;j<8;j++)
        {
           gets(cargar_datos_v);
           precision_dato[j]=cargar_datos_v[1];
           //xil_printf("//%d d0",cargar_datos_v[0]);
           //xil_printf("//%d d1",cargar_datos_v[1]);
           //xil_printf(" precision %d \n",j);
        }

        for(j=0;j<8;j++)
        {
           gets(cargar_datos_v);
           vuelta_dato[j]=cargar_datos_v[1];
           dir_int[j]=0;
           //xil_printf("//%d d0",cargar_datos_v[0]);
           //xil_printf("//%d d1",cargar_datos_v[1]);
           //xil_printf(" vuelta %d \n",j);
        }

        for(j=0;j<8;j++)
        {
           barra_next[j].precision=precision_dato[j];
           barra_next[j].vuelta=vuelta_dato[j];
        }


        xil_printf("barra 0 %d\n",barra_next[0].vuelta);
        xil_printf("barra 1 %d\n",barra_next[1].vuelta);
        xil_printf("barra 2 %d\n",barra_next[2].vuelta);
        xil_printf("barra 3 %d\n",barra_next[3].vuelta);
        xil_printf("barra 4 %d\n",barra_next[4].vuelta);
        xil_printf("barra 5 %d\n",barra_next[5].vuelta);
        xil_printf("barra 6 %d\n",barra_next[6].vuelta);
        xil_printf("barra 7 %d\n",barra_next[7].vuelta);

        xil_printf("barra 0 %x\n",barra_next[0].precision);
        xil_printf("barra 1 %x\n",barra_next[1].precision);
        xil_printf("barra 2 %x\n",barra_next[2].precision);
        xil_printf("barra 3 %x\n",barra_next[3].precision);
        xil_printf("barra 4 %x\n",barra_next[4].precision);
        xil_printf("barra 5 %x\n",barra_next[5].precision);
        xil_printf("barra 6 %x\n",barra_next[6].precision);
        xil_printf("barra 7 %x\n",barra_next[7].precision);


        control_parada=0;


  // fase de movimiento de las barras hacia la posición indicada

  /**fase de movimiento de las barras hacia la posicion indicada*/


  // actualizar los datos que se mantendran constantes del pwm:
  //    - el retardo del sistema
```

```
  //    - tiempo total del ancho de pulso de pwm

        XIo_Out32(baseaddr_retardo, 0x00000000);
        XIo_Out32(baseaddr_pwm_total, 0x00000060);

        xil_printf("*******MOVIMIENTO DE LAS BARRAS*******\r\n");


        while(control_parada < 8){

           // bucle de control de elementos
           control_parada=0;

           for (aux=0;aux<NUM;aux++)
           {
              if(aux%2) //barras izquierdas
           //actualizar la posicion actual de la barra

  actualizarIzquierda(&barra_now[aux],dir_int[aux],baseaddr_my_gpio+4*(aux));
              else  //barras derechas
                 //actualizar la posicion actual de la barra

  actualizarDerecha(&barra_now[aux],dir_int[aux],baseaddr_my_gpio+4*(aux));

              i=control(&barra_now[aux],&barra_next[aux],&flanco,&dir_int[aux]);


              control_parada=control_parada+i;

              //calcular la señal de actualizacion del pwm
              //if (flanco==DATA_BURDO)
              // enable_aux=0x00000000+disp[aux]+ce[aux];
              //else
                 enable_aux=0x00000000+disp[aux];

              //calcular la señal de flanco del pwm
              flanco_aux= flanco + cap[aux];

              // actualizar el flanco de pwm
              if(flanco_ant[aux]!=flanco_aux){
                 XIo_Out32(baseaddr_pwm_dat, flanco_aux);
                 flanco_ant[aux]=flanco_aux;
              }

              dir_aux=resultadoDireccion(dir_int);

              //actualizar la direccion del pwm
              XIo_Out32(baseaddr_pwm_dir, dir_aux);
              //activar el pwm
              XIo_Out32(baseaddr_pwm_abilitacion, enable_aux);

           }

           //codigos de movimiento cd:b1v02p0003

           //pasar a cadenas

           for (j=0;j<8;j++){
              sprintf(vuelta_string, "%02d", barra_now[j].vuelta);
              sprintf(precision_string, "%04d", barra_now[j].precision);
              xil_printf("cd:b%dv%sp%s\n",j,vuelta_string,precision_string);
           }
        }
  }

   return 0;
}
```