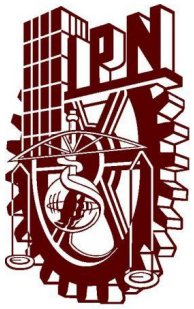


INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Laboratorio de Microtecnología y Sistemas Embebidos

METODOLOGÍA PARA EL DESARROLLO DE BIBLIOTECAS DE FUNCIONES HARDWARE

TESIS

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS EN INGENIERÍA DE CÓMPUTO CON OPCIÓN EN SISTEMAS DIGITALES

P R E S E N T A:

Lic. Alejandro Gómez Conde

Directores de tesis:
Dr. Marco Antonio Ramírez Salinas
Dr. José Luis Oropeza Rodríguez

Mexico, D.F. Enero 2013

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 12:00 horas del día 07 del mes de diciembre de 2012 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Metodología para el desarrollo de bibliotecas de funciones hardware"

Presentada por el alumno:

GÓMEZ	**CONDE**	**ALEJANDRO**
Apellido paterno	Apellido materno	Nombre(s)

Con registro:

B	1	0	1	8	7	6

aspirante de: **MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO CON OPCIÓN EN SISTEMAS DIGITALES**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA
Directores de Tesis

Dr. Marco Antonio Ramírez Salinas

Dr. José Luis Oropeza Rodríguez

Dr. Amadeo José Argüelles Cruz

Dr. Rolando Menchaca Méndez

M. en C. Osvaldo Espinosa Sosa

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Luis Alfonso Villa Vargas

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION

DIRECCIÓN

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de **México, D. F.** el día **12** del mes de **Diciembre** del año **2012**, el que suscribe **Alejandro Gómez Conde** alumno del Programa **Maestría en Ciencias en Ingeniería de Cómputo con Opción en Sistemas Digitales** con número de registro **B101876**, adscrito al **Centro de Investigación en Computación**, manifiesta que es autor intelectual del presente trabajo de Tesis bajo la dirección de **Dr. Marco Antonio Ramírez Salinas** y **Dr. José Luis Oropeza Rodríguez** y cede los derechos del trabajo intitulado **Metodología para el desarrollo de bibliotecas de funciones hardware**, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección **alegzc@yahoo.com.mx**. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Alejandro Gómez Conde

Nombre y firma

Resumen

La creciente demanda de dispositivos móviles ha fomentado el desarrollo y aplicación de sistemas embebidos que incluyen uno o más FPGAs (Field Programmable Gate Arrays). En este nuevo contexto de desarrollo la industria está experimentando el paradigma de ejecución Hardware/Software para dispositivos móviles. Para ello ha creado dispositivos que incluyen uno o varios procesadores, tarjetas de desarrollo que incluyen uno o varios dispositivos configurables y dispositivos que incluyen procesadores y FPGAs en el mismo empaque.

La comunidad de investigadores ha trabajado en el paradigma de ejecución Hardware/Software desde hace tres lustros aproximadamente. Producto de las diversas líneas de investigación asociadas al proceso de interacción Hardware/Software para el desarrollo e implementación de sistemas embebidos, se han propuesto metodologías para describir los procedimientos relacionados a la interacción Hardware/Software.

Las propuestas generadas por la comunidad de investigadores atacan problemas relacionados a la implementación de estos sistemas y están centradas en la modificación de los servicios que proporciona el núcleo del sistema operativo. Algunas de estas propuestas requieren la modificación de subsistemas críticos del núcleo como el planificador de tareas o el subsistema de ejecución de procesos; otras propuestas agregan campos de datos a las estructuras descriptivas de sistema para diferenciar entre tareas hardware y software. Estas propuestas requieren la modificación del código fuente del núcleo y son dependientes de la versión del núcleo en el que se implementan y al tipo de arquitectura sobre la cual se ejecuta el sistema operativo.

Otro conjunto de propuestas de la comunidad de investigadores está centrada en trasladar elementos funcionales del software al hardware y *viceversa*. Se han desarrollado planificadores de tareas hardware que colaboran con el planificador de tareas software a fin de presentar una única interfaz de aplicación a otros subsistemas dentro del núcleo. Además de trasladar elementos funcionales del software al hardware y *viceversa* estas propuestas agregan servicios para configurar una o más regiones dentro del FPGA. Sin embargo, las metodologías propuestas están fuertemente ligadas a detalles específicos de la arquitectura hardware y no toman en cuenta la característica modular utilizada en los sistemas operativos, limitando con ello la portabilidad entre las diversas arquitecturas en desarrollo. Aunado a ello, los sistemas embebidos con hardware reconfigurable permiten a las tareas cumplir con

los requerimientos en tiempo de ejecución y al mismo tiempo mantener los requerimientos de energía promedio. Pero programar efectivamente este tipo de arquitecturas reconfigurables, es un proceso complejo y propenso a errores, ya que exige a los desarrolladores a asumir el papel de programador, diseñador hardware y dominar los lenguajes de descripción de hardware para hacer una implementación eficiente de los módulos funcionales.

Por ello, esta tesis genera una metodología para el desarrollo de bibliotecas de funciones hardware; la cual permite a los desarrolladores de SoPC usar bibliotecas de funciones hardware tal y como lo harían con las actuales bibliotecas de funciones software, ello facilita el uso y difusión de SoPC.

La metodología propuesta ejemplifica, a través de siete módulos IPCore, los procedimientos necesarios para generar una biblioteca experimental de funciones hardware, la capa de servicios software, los modelos de interacción Hardware/Software y complementa esta propuesta con el modelo de interceptación de llamadas a funciones de bibliotecas dinámicas.

Abstract

The growing demand for mobile devices has promoted the development and implementation of embedded systems that include one or more FPGAs (Field Programmable Gate Arrays). In this new development context the industry is experimenting with a Hardware/Software co-execution paradigm for mobile devices. Therefore industry has created circuits that include one or more processors, developed boards that include one or more configurable chips and devices that include processors and FPGAs in the same package.

The research community has been working on the Hardware/Software co-execution paradigm since mid 90's. Methodologies, derived from the various lines of research related to the Hardware/Software interaction specifically for the development and implementation of embedded systems, have emerged to describe the interaction between hardware and software.

Proposals generated by the research community to attack problems regarding the implementation of embedded systems and are focused on modifications to services provided by the operating system kernel. Some of those proposals require the modification of critical subsystems within the kernel like the scheduler or process management subsystem. Other proposals add data structures to system's description structures to differentiate between hardware and software tasks. They require modifications to the kernel source code; they are dependent to the kernel version in which they are implemented and the type of architecture on which operating system is running.

The research community have proposed the migration of functional elements from software to hardware and *vice versa*. Hardware tasks schedulers have been developed that work together with a software scheduler to generate a single application program interface for other subsystems within the kernel. Besides that, these proposals added software services to configure one or more regions within the FPGA. However, the proposed methodologies are closely tied to specific details of the hardware architecture and do not take into account the philosophy used in Linux to create modular elements that can be dynamically loaded, thus limiting the portability among new architectures. In addition to this, embedded systems with reconfigurable hardware allow applications to meet execution time constraints while maintaining average power requirements. But effectively programming such reconfigurable architectures, however, is an extremely cumbersome and error-prone process, as it requires programmers to assume the role of hardware designers while mastering hardware description languages.

Therefore, this thesis creates a methodology for developing hardware function libraries; through it, embedded systems developers use hardware function libraries as they do with existing software libraries, allowing the use and dissemination of SoPC.

The proposed methodology exemplifies, through seven IPCore modules, the procedures needed to generate an experimental hardware library, software services, Hardware/Software interaction models and finally this work complements this proposal with the function interception model for dynamic libraries.

Dedicatoria

A mi familia:

Teresa Conde Hernández,
Juan Carlos y Mary Carmen Gómez Conde.

Fuente inagotable de motivación.

Agradecimientos

Agradezco a mi madre, hermano y hermana por su amor y apoyo incondicional.

Agradezco a la Dra. María Teresa Cuamatzi y familia por el incondicional apoyo recibido.

Agradezco a todos mis amigos por su apoyo, consejos y recomendaciones, pero sobre todo por compartir conmigo el muy preciado tiempo de sus vidas.

Agradezco de manera especial a mis directores de tesis por el esfuerzo y dedicación para consolidar este proyecto de investigación y a todos los profesores que colaboraron directa o indirectamente en mi formación académica.

Agradezco al Centro de Investigaciones en Computación, la oportunidad otorgada para la realización de mis estudios de maestría y las facilidades que permitieron hacer realidad este sueño de mi vida.

Agradezco al Instituto Politécnico Nacional por concederme el honor de pertenecer a esta comunidad y facilitar el desarrollo de mis actividades académicas, deportivas y culturales, y de manera muy especial a los departamentos de servicios médicos y rehabilitación.

Agradezco al Consejo Nacional de Ciencia y Tecnología por el apoyo recibido durante mis estudios de maestría.

Acrónimos, abreviaturas y siglas

Este apartado presenta los acrónimos, abreviaturas y siglas usadas en el texto. Se incluye una descripción breve a fin de que el lector pueda conocer el contexto en el que fue enunciada.

ABI	Application Binary Interface.
API	Application Programming Interface.
APU	Auxiliary Pocessor Unit.
ASIC	Application Specific Integrated Circuit, es un circuito integrado diseñado y optimizado para implementar alguna funcionalidad en particular.
BEE 2	Berkeley Emulation Engine, versión 2; es una tarjeta de desarrollo con cinco FPGAs Virtex 2 Pro 70.
BOF	BORPH Object File, archivo ejecutable en la plataforma BEE 2 con el sistema operativo BORPH.
BORPH	Berkeley Operating system for ReProgrammable Hardware.
BPS	Baudios Por Segundo.
BRAM	Block RAM, son elementos de memoria RAM agrupados en bloques que se implementan dentro del FPGA.
BSB	Base System Builder.
BSP	Board Support Packages.
CF	Compact Flash.

CIP	Create or Import Peripheral.
CPU	Central Processing Unit.
DCR	Device Configuration Register.
DDS	Direct Digital Synthesizer.
DLL	Dynamic Linker Loader.
EDLK	Embedded Linux Development Kit.
ELF	Executable and Linkable Format.
FIFO	First In First Out.
FPGA	Field Programmable Gate Array.
HELF	Hardware Executable and Linkable Format.
ICAP	Internal Configuration Access Port, Dispositivo hardware dentro del FPGA que permite acceder al puerto de configuración.
ISA	Instruction Set Architecture.
JTAG	Joint Test Action Group (IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture).
L/S	Unidad de carga y almacenamiento (*Load/Store Unit*).
LED	Light-Emitting Diode
MMU	Memory Management Unit.
OPB	On-chip Peripheral Bus.
PID	Process IDentification, número de identificación del proceso.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computing.
SoPC	System on a Programmable Chip.
TLB	Translation Lookaside Buffer.

Glosario

Este apartado presenta los términos usados en esta tesis y describe el significado que representan en el contexto del desarrollo de este trabajo.

Archive: En el contexto del sistema operativo Linux un *archive* es tipo especial de archivo que almacena una colección de otros archivos en una estructura ordenada que permite recuperar los archivos originales.

Big Endian: Se refiere al orden de almacenamiento de datos en memoria.

Bootstrap: En el contexto de los SoPCs, el cargador de inicialización es un fragmento de código que se incrusta en el archivo de configuración del FPGA. Se ha denominado cargador de arranque de primera etapa (*First Stage Bootloader*).

Colocación: En el contexto de los FPGAs, se deriva de la traducción del término *placement*; el término hace referencia a un conjunto de procedimientos que determinan la distribución física de los elementos lógicos de un diseño en el área del dispositivo.

Enrutado de un FPGA: Consiste en la especificación y selección de los interruptores de interconexión para los alambres que implementan líneas de comunicación dentro del FPGA.

Gateware: Diseños hardware para FPGAs que pueden ser compartidos entre los investigadores. El concepto evolucionó para crear lo que hoy se conoce como *Open Source Hardware*.

Hardware/Software: En el contexto de ejecución de tareas, este concepto se refiere a los elementos hardware y software necesarios para ejecutar una tarea dada. De manera particular para los sistemas embebidos este concepto representa la interacción entre módulos hardware implementados en el FPGA y procedimientos software que hacen uso de esos recursos hardware.

IPCore: Es un módulo hardware que implementa determinada funcionalidad a través de elementos lógicos y de almacenamiento.

Núcleo de Linux: También llamado kernel de Linux es un conjunto de servicios software que implementa además, una capa de abstracción entre el hardware y los programas de usuario proveyendo una interfaz de acceso entre la funcionalidad provista por el hardware y los requerimientos de los programas.

Plataforma Hardware: Es el conjunto de elementos lógicos configurables (IP Cores) y físicos (circuitos integrados) sobre los cuales se ejecuta un sistema operativo.

Soft Core: IP Core que implementa la funcionalidad de un procesador a través de elementos reconfigurables de un dispositivo programable.

SysACE: Sistema de configuración de dispositivos programables desarrollado por Xilinx mediante el cual se accede a un dispositivo de almacenamiento del tipo Compact Flash.

Traza de ejecución: Una traza de ejecución se integra por las condiciones resultantes del proceso de ejecución de una instrucción donde se incluyen los resultados de las banderas de estado y el contenido de los registros del procesador.

Índice general

Índice de figuras

Índice de tablas

Índice de Listados

Capítulo 1

Introducción

Un sistema embebido es un sistema en el cual hardware y software se unen para realizar una tarea específica[5]. De manera particular, un sistema embebido basado en FPGA (Field Programmable Gate Array) incluye un núcleo de procesamiento, que puede ser prefabricado (*hard processor*) o sintetizable (*soft processor*), elementos de memoria, un sistema operativo, programas de usuario y bibliotecas de funciones. Históricamente, el diseño de estos sistemas había sido regido por dos conceptos de diseño: el tamaño o la capacidad de cómputo; estos conceptos parecían ser mutuamente excluyentes pero el desarrollo tecnológico actual ha permitido estrechar la brecha entre ellos.

Tiempo después la industria comenzó a desarrollar dispositivos altamente integrados que permiten generar sistemas embebidos más compactos. La comunidad científica propuso modelos de ejecución Hardware/Software que logran incrementar la eficiencia o aumentar la capacidad de cómputo en los dispositivos; sin embargo las metodologías empleadas están estrechamente ligados a los detalles de la implementación y por ello el proceso para integrar dicha funcionalidad a otros sistemas resulta ser largo y en ocasiones inviable debido a las especificaciones hardware presentes en otras plataformas.

Otras propuestas centran su atención en mejorar el tiempo de ejecución a través de la incorporación de elementos funcionales hardware que colaboran en la realización de tareas del sistema operativo y fueron específicamente desarrolladas para administrar elementos funcionales implementados en hardware; entre estos elementos podemos mencionar al planificador de tareas, al subsistema de entrada y salida de datos y al subsistema de ejecución de procesos. Aunado a ello, la creciente demanda de dispositivos exige al desarrollador de sistemas embebidos asumir el papel de programador, dise-

ñador hardware y dominar los lenguajes de descripción de hardware para hacer una implementación eficiente de los módulos funcionales al tiempo de explorar y seleccionar de entre los modelos de interacción Hardware/Software aquél que mejor se adapte al desarrollo del sistema embebido. Debido a ello, esta tesis presenta una metodología para el desarrollo de bibliotecas de funciones hardware donde un módulo funcional hardware colabora en el proceso de ejecución de un programa de usuario; esta metodología también presenta una solución para simplificar el proceso de interacción Hardware/-Software mediante el acceso a funciones de bibliotecas. Este concepto ha sido ampliamente utilizado en el desarrollo software puesto que generalmente un programa de usuario es segmentado en diversos elementos funcionales los cuales pueden ser agrupados en bibliotecas de funciones para su posterior uso.

1.1. Antecedentes

El resultado de combinar los recursos hardware y software a fin de resolver un problema específico se remonta al inicio de las computadoras. Las propuestas de la comunidad científica para trasladar elementos funcionales del entorno software al hardware y *viceversa* derivó en la necesidad de generar nuevas metodologías que permitiesen especificar la interacción Hardware/-Software tal como se destaca en los siguientes trabajos.

Hayden Kwok-Hay So *et al.*[42] presentaron una metodología para el co–diseño Hardware/Software basado en BORPH (*Berkeley Operating system for ReProgrammable Hardware.*). Esta metodología incluye el diseño, desarrollo e implementación de un sistema operativo para computadoras reconfigurables, un modelo de ejecución unificado para procesos hardware y software, la creación de un nuevo tipo de archivo ejecutable que contienen datos específicos y necesarios para la ejecución del procesos hardware, un protocolo de comunicación entre el SO y el proceso hardware implementado en un FPGA, y servicios de reconfiguración para la programación y configuración de los FPGAs. La implementación de BORPH se realizó en una plataforma hardware con cinco FPGAs donde uno de ello se configuró como unidad maestra y los cuatro restantes se pusieron a disposición del sistema operativo para implementar un proceso hardware en cada FPGA.

Qingxu Deng, *et al.*[13] propusieron un modelo de arquitectura unificada

para la implementación de servicios de reconfiguración parcial de un FPGA en tiempo de ejecución. Esta propuesta implementó un planificador de procesos para administrar la ejecución de los módulos hardware en el área reconfigurable del FPGA. También implementaron el enrutado de la interfaz de entrada y salida de los módulos funcionales, y la colocación de bloques funcionales en el área reconfigurable del FPGA. Las tareas hardware y software son tratadas como procesos dentro del sistema operativo; una tarea hardware hace uso de una capa de comunicación software en el sistema operativo, la cual se encarga de enviar y recibir datos entre el módulo funcional y la tarea en ejecución. En el proceso de ejecución de una tarea hardware se hace uso de una versión modificada de la llamada a sistema *exec* para que el sistema operativo reconozca y ejecute un tipo especial de archivo denominado HELF (*Hardware Executable and Linkable Format*). Esta llamada a sistema extrae la información de configuración del FPGA y el esquema de comunicación a implementarse entre la capa software de comunicación y el modulo hardware.

Vaibhawa Mishra *et al.*[32] presentaron la implementación de un sistema SoPC (*System on a Programmable Chip*) con reconfiguración dinámica que incluye un sistema operativo basado en el kernel de Linux 2.6.34. Este proyecto se implementa en la tarjeta de desarrollo ML507 de Xilinx la cual incluye el procesador PowerPC 440, un FPGA de la familia Virtex 5, elementos de almacenamiento y periféricos de entrada/salida. La propuesta de esta implementación ejemplifica un modelo de ejecución donde el módulo hardware es accesible desde el programa de usuario con la opción de realizar la reconfiguración parcial del módulo funcional.

Alonso[24] desarrolló un conjunto de reglas para el sistema de desarrollo de Poky que permiten generar un sistema operativo para la tarjeta de desarrollo ML507 de Xilinx. Estas reglas hacen uso de la funcionalidad que provee Poky para conectarse a repositorios remotos y recuperar los paquetes necesarios para la generación del sistema. Este trabajo también propuso una plataforma hardware con soporte para gráficos la cual es sintetizable en la tarjeta ML507.

1.2. Planteamiento del problema

La comunidad científica ha propuesto diversas metodologías para la interacción Hardware/Software. Todos las mejoras propuestas impactan de manera directa o indirecta en la disminución del tiempo de ejecución de la aplicación de usuario.

De la investigación antes mencionada podemos resaltar las siguientes aspectos que son los más afines al desarrollo de este trabajo de tesis:

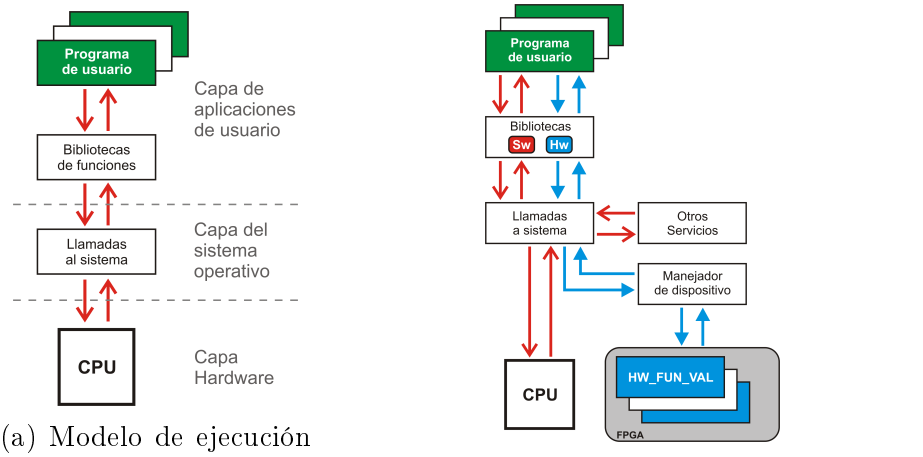
La propuesta de Hayden Kwok-Hay So *et al.*[42] presentó una revolucionaria forma de implementar procesos hardware; la metodología propuesta funciona en una plataforma hardware con cinco FPGAs pero es necesario desarrollar tanto la aplicación de usuario como el módulo hardware para concatenarlos dentro de un nuevo tipo de archivo ejecutable llamado *BOF* y modificar los subsistemas del núcleo de Linux para que pueda interpretar adecuadamente este nuevo tipo de archivo ejecutable.

Qingxu Deng *et al.*[13] presentaron un enfoque para crear un modelo abstracto de una tarea hardware que puede ser lanzada desde el espacio de usuario, requiere modificar una función critica del núcleo del SO y la integración de los servicios de planificación de procesos hardware al subsistema de planificación del SO.

El desarrollo de aplicaciones para sistemas embebidos puede incluir toda la funcionalidad requerida dentro del código ejecutable de la aplicación o bien la implementación de la solución es segmentada en múltiples elementos funcionales los cuales pueden agruparse en un conjunto que se denomina biblioteca de funciones. Así la aplicación de usuario reduce su tamaño pero aumenta el número de archivos. Sin embargo el aumento en el número de archivos puede beneficiar a otras aplicaciones que hacen uso de los mismos elementos funcionales. De este concepto surgió lo que ahora se conoce como objetos compartidos (Traducción de *shared objects*); la ventaja de esta metodología es que permite compartir elementos funcionales entre distintas aplicaciones, por ejemplo: los programa de usuario en Linux generalmente requieren de la funcionalidad provista por la biblioteca estándar de C la cual se encuentra en un archivo tipo *.so* para el sistema de desarrollo usado, esta biblioteca se llama `libc-2.13.so`. Las bibliotecas software proveen la funcionalidad requerida por muchos programas, disminuyen el tamaño de la implementación y colaboran en la generación de aplicaciones modulares, siguiendo la filosofía del núcleo de Linux.

La fig. 1.1a muestra el esquema de implementación para una aplicación que fue segmentada en múltiples elementos funcionales y usa llamadas a bibliotecas de funciones. En este esquema, un programa de usuario implementa llamadas a funciones de biblioteca las cuales a su vez hacen uso de forma directa o indirecta de las llamadas a sistema; de manera tradicional la funcionalidad asociada a la llamada al sistema se ejecuta en la CPU. Para un sistema embebido este modelo de ejecución ha sido empleado desde hace varios años; sin embargo los desarrollos actuales permiten explorar nuevos paradigmas en la interacción Hardware/Software.

Esta tesis propone una metodología para asistir a una tarea software a través de módulos funcionales hardware. La fig. 1.1b presenta el modelo de ejecución derivado de la metodología propuesta. Este esquema de interacción permite incrustar las llamadas a funciones dentro del código fuente. Estas funciones permiten en envío y recepción de datos entre el programa de usuario y el sistema operativo. Todas las funciones de biblioteca en algún momento hacen uso, de forma directa o indirecta, de una o más llamadas a sistema que proporciona el núcleo de Linux. A través de estas llamas a sistema se puede interactuar con un manejador de dispositivo el cual a su vez interactúa con el elemento funcional hardware solicitado por la aplicación de usuario.



(a) Modelo de ejecución tradicional.

(b) Modelo de ejecución Hardware/Software.

Figura 1.1: Modelos de ejecución.

1.3. Justificación

Los sistemas embebidos están presentes en muchas aplicaciones hoy día, sin embargo, los desarrolladores de este tipo de sistemas rara vez son expertos en desarrollo software y desarrollo hardware por tanto no siempre pueden desarrollar módulos hardware optimizados, codificar las aplicaciones que harán uso de los módulos funcionales y explorar los modelos de interacción Hardware/Software para seleccionar el mejor método de intercomunicación en función de los detalles de la arquitectura, por ello esta tesis genera una metodología para el desarrollo de bibliotecas de funciones hardware que permitirá asistir a una tarea software mediante el uso de elementos funcionales en hardware.

Actualmente México ha iniciado el proceso de independencia tecnológica a través de diversos proyectos. Alligator_SP[31] ha generado los módulos funcionales de un procesador superescalar con ejecución fuera de orden. Este proyecto generó diversas líneas de investigación. Uno de estos trabajos desarrolló un *buffer* de re-ordenamiento de instrucciones[36]. Además de ser un procesador de alto rendimiento Alligator_SP ha generado nuevas ideas que mejoran el rendimiento de los procesadores superescalares. Alligator_OS[24], es el nombre del proyecto que generó un sistema operativo embebido a partir del uso de herramientas de compilación, configuración e integración presentes en Poky. Esta propuesta generó un sistema operativo para un procesador PowerPC 440 y el subsistema de desarrollo utilizado hace uso de los repositorios que constantemente están siendo actualizados para la generación de las herramientas de compilación y de manera general para la generación del sistema operativo.

El proyecto desarrollado en esta tesis contribuirá a la estabilización del sistema de desarrollo en Poky Linux a través de la creación de repositorios locales y la modificación de los archivos que rigen los procesos de compilación generación del sistema operativo a fin de que los investigadores puedan trabajar en un entorno de desarrollo controlado.

1.4. Objetivo

Generar una metodología para el desarrollo de bibliotecas de funciones hardware.

1.4.1. Objetivos específicos:

- Integrar el sistema operativo Poky Linux para el desarrollo de funciones hardware a la tarjeta ML507 de Xilinx.
- Proponer un modelo de transferencia de datos entre el módulo funcional y la aplicación de usuario.
- Implementar un modelo funcional que permita a los desarrolladores software hacer uso de la funcionalidad hardware cuando no se tiene acceso al código fuente de la aplicación.

1.4.2. Alcances del trabajo

Este trabajo desarrollará un sistema embebido con soporte para la adición de funciones hardware que permitan asistir una tarea software usando la metodología propuesta sin integrar los servicios de reconfiguración dinámica en tiempo de ejecución. También se desarrollará la capa software que proporciona los servicios de acceso a la función hardware y la generación de una biblioteca de funciones hardware.

1.4.3. Contribuciones

Esta tesis presenta una metodología para el desarrollo de funciones hardware y para ello es necesario configurar y estabilizar el proceso de generación del sistema operativo Poky Linux para que a través de un conjunto de repositorios locales se provea acceso sin restricciones al código fuente a fin de mantener un control sobre el núcleo del sistema operativo, el cargador de arranque y de otras utilidades necesarias en el proceso de generación del sistema operativo.

La implementación de este proyecto insta a los desarrolladores software a crear modelos de ejecución basados en esta metodología para facilitar el uso de sistemas embebidos que incluyen uno o más FPGAs. Esta metodología permite además agregar portabilidad a las aplicaciones desarrolladas a través de la captura de llamadas a funciones. Esto permite a una aplicación mantener independencia al no incluir rutinas específicas para la interacción con elementos hardware como parte de su código sino que sólo deberá incluir llamadas a funciones las cuales pueden ser implementadas en software o en hardware de acuerdo a las necesidades de cada sistema en particular.

1.5. Trabajo surgido de la tesis

Artículo: A Hardware/Software Co–Execution Model Using Hardware Libraries for a SoPC Running Linux. Alejandro Gómez-Conde, José de Jesús Mata-Villanueva, Marco A. Ramírez-Salinas and Luis A. Villa-Vargas. 12^{th} International Congress on Computer Science CORE2012.

1.6. Organización de la tesis

Capítulo 1 inicia con la introducción a los sistemas embebidos, comenta los antecedentes del área, argumenta la importancia de esta investigación y presenta los objetivos de la tesis.

Capítulo 2 Presenta un análisis del estado del arte, de la problemática a resolver y las propuestas que ha generado la comunidad de investigadores para explorar los procesos de interacción Hardware/Software.

Capítulo 3 Presenta los fundamentos teóricos relativos al diseño de sistemas embebidos con ejecución Hardware/Software.

Capítulo 4 Presenta la metodología propuesta para la ejecución de funciones hardware.

Capítulo 5 Detalla los experimentos de validación y los resultados obtenidos en cada prueba derivada de la metodología propuesta.

Finalmente, se presentan las conclusiones y los trabajos a futuro que se pueden desarrollar a partir de este trabajo.

Capítulo 2

Estado del arte

Este capítulo describe las propuestas generadas por la comunidad de investigadores donde se presentan soluciones a diversos problemas existentes durante el desarrollo e implementación de sistemas embebidos que incluyen FPGAs. De manera particular se analizarán a detalle tres propuestas de la comunidad de investigadores que implementan modelos de interacción que están particularmente diseñados para dar solucionar problemas de interacción entre Hardware/Software en sistemas que incluyen FPGAs.

2.1. Diseño de un Sistema Embebido

Los sistemas embebidos son el resultado de integrar hardware y software para la solución de problemas específicos. Hasta hace algunos años el proceso de diseño de un sistema embebido se dividía en dos grandes secciones claramente delimitadas: hardware y software. Aún es frecuente observar la integración de una plataforma hardware con un CPU, elementos de almacenamiento, dispositivos de entrada y salida, etc., y sobre esta plataforma hardware desarrollar un conjunto de capas software que resuelven un problema específico.

En la actualidad, el proceso de diseño de un sistema embebido ya no está claramente separado, los requerimientos del problema a resolver son cada vez más complejos y su implementación requiere de la asistencia de herramientas de desarrollo diseñadas específicamente para tales fines. Los desarrollos tecnológicos de los últimos años permiten explorar metodologías aplicables a los procesos de ejecución Hardware/Software donde una aplicación de usua-

rio implementa elementos funcionales en módulos hardware y procedimientos software los cuales en conjunto colaboran para la solución de un problema particular. Un caso específico derivado de los desarrollos tecnológicos actuales es la integración de procesador y FPGA dentro del mismo circuito integrado.

Los dispositivos reconfigurables como el FPGA, son usados para implementar la funcionalidad de al menos una sección del algoritmo a través de módulos hardware. Debido a la segmentación del programa en ejecución, se vuelve indispensable explorar los modelos de interacción hardware/software; por ello esta sección analiza las propuestas de la comunidad de investigadores relativas a los procesos de interacción Hardware/Software.

Los sistemas embebidos desarrollados con anterioridad, ejecutan tareas que se implementan enteramente en hardware (ej. ASIC) o enteramente en software (ej. una o más aplicaciones que se ejecutan en un procesador). Ambas propuestas ejecutan una tarea dada para resolver un problema particular, y por tanto generan el mismo resultado pero difieren en el tiempo de ejecución y/o el costo de fabricación. Esta es una razón por la cuál la industria ha desarrollado dispositivos que integran hardware reconfigurable a unidades de procesamiento (CPU) para explorar el paradigma de ejecución Hardware/-Software para dispositivos móviles.

Existen tres grandes áreas de investigación para los sistemas embebidos; ellas son:

- Co-diseño Hardware/Software
- Co-síntesis Hardware/Software
- Particionamiento Hardware/Software

Estas áreas permiten clasificar las aportaciones resultantes de los procesos de investigación y clasificarlas conforme al marco de referencia que cada una de ellas considera. La metodología propuesta en esta tesis involucra estas tres áreas de desarrollo tal y como se muestra en la figura 2.1 puesto que la aplicación fue segmentada y cada elemento de la partición implementa algún procedimiento software o funcionalidad hardware. De manera general el proceso de particionamiento de una aplicación software genera diversos elementos funcionales que comúnmente se agrupan en bibliotecas de funciones software para ser enlazadas estática o dinámicamente a la aplicación de usuario. La figura 2.1 también muestra la relación en los temas de investigación relativos al diseño de sistemas embebidos.

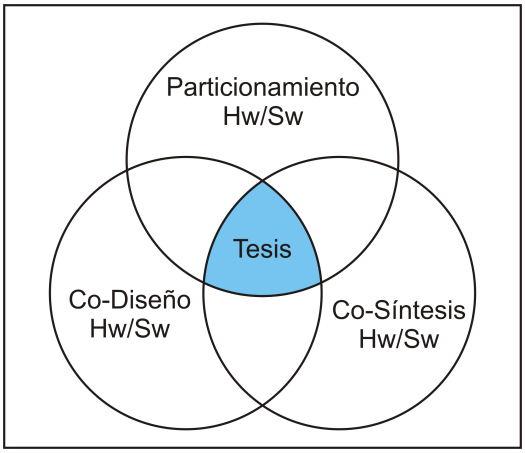


Figura 2.1: Áreas de investigación para sistemas embebidos.

En el caso particular de los sistemas embebidos que implementan algún esquema de interacción Hardware/Software el particionamiento de una aplicación ha motivado a la comunidad de investigadores a desarrollar metodologías que permitan optimizar esta interacción; ejemplos de estas propuestas se muestran en [4], [6], [11], [15], [19], [20], [26], [30], [34] y [38].

El el contexto del diseño de sistemas embebidos, el co-diseño Hardware/-Software se utiliza para determinar la forma con se integrarán los módulos hardware con el software ya sea a nivel de sistema operativo o de aplicación de usuario. En este marco de referencia, la comunidad de investigadores ha propuesto técnicas de diseño asistido que permiten utilizar las especificaciones derivadas del proceso de particionamiento para establecer restricciones de diseño y políticas de interacción entre los elementos que conforman la implantación del sistema; ejemplos de estas propuestas se pueden encontrar en [5], [25], [34], [40], [43] y [45]; J. Teich presenta un reporte técnico acerca del proceso de evolución del co-diseño Hardware/Software en [44].

La industria han propuesto y desarrollado entornos de trabajo que retoman las ideas generadas por los desarrolladores e investigadores y que además integran diversas herramientas para la síntesis de módulos hardware a fin de simplificar los procesos de integración entre la arquitectura del sistema y los bloques hardware funcionales; ejemplos de ello se pueden ver en [49], [2], [3] y [7].

Dentro del marco de referencia asociado al proceso de co-síntesis para sis-

temas embebidos, se determina la forma de implementar *a priori*[1] o *in situ*[2] los elementos funcionales requeridos por el sistema embebido. Para resolver los problemas derivados de este proceso, la comunidad de investigadores ha propuesto [7], [12], [13], [16], [18], [22], [32], [33], [35], [37] y [40].

R. Gupta y G. De Micheli[18] mostraron que es posible conseguir la síntesis de sistemas heterogéneos que utilizan restricciones de tiempo para delegar tareas entre el hardware y el software de tal manera que la aplicación final cumpla los requerimientos y restricciones del problema a resolver.

Ellos presentan un modelo de síntesis enfocado a los sistemas embebidos mixtos, el cual se presenta en la figura 2.2. El proceso inicia con la especificación del problema a resolver; de él se genera un modelo que representa la secuencia de ejecución para resolver el problema planteado. Posterior a ello, se realiza un particionamiento del grafo que representa al problema siguiendo un conjunto de reglas propuestas. El resultado de este particionamiento genera un grafo bi-seccionado donde un subconjunto se implementará en hardware y el otro en software.

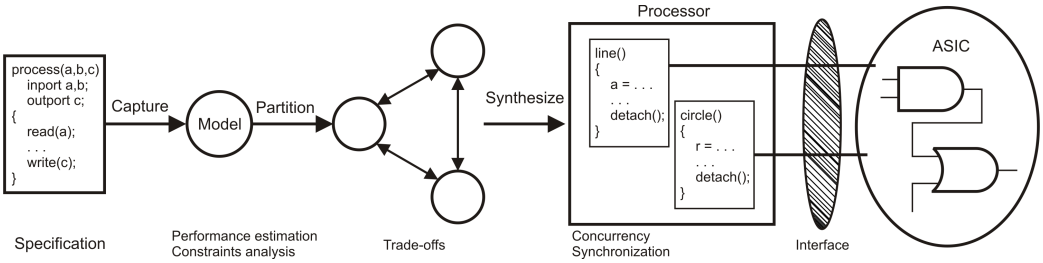


Figura 2.2: Modelo de desarrollo propuesto por Guta y De Micheli.

2.2. BORPH

Hayden Kwok-Hay So[41] desarrolló un proyecto de investigación durante su doctorado en *UC Berkeley* que generó una metodología para el co–diseño

[1]Este término hace referencia al proceso de síntesis realizado antes de configurar los elementos internos del FPGA.

[2]Este término hace referencia a los procesos necesarios para modificar en tiempo de ejecución el archivo de configuración asociado a un bloque hardware dentro del FPGA.

Hardware/Software basado en BORPH (*Berkeley Operating system for Re-Programmable Hardware*). El proyecto hace una extensión de la interfaz de aplicación de programa (API) del sistema operativo Linux para la ejecución de procesos hardware. De manera general, un proceso es una instancia en ejecución de un programa; en el contexto de desarrollo de BORPH, un proceso hardware es una instancia en ejecución de un programa *gateware*[3][23].

En este contexto, un proceso hardware tiene asociado los mismos privilegios que un proceso software, es decir, cada proceso de usuario se ejecuta en su propio espacio de direcciones aislado del espacio de direcciones de otros procesos activos. Más aún, el proceso también tiene acceso completo al procesador en el que se está ejecutando y por ello es frecuente decir que tiene su propia máquina virtual en la que se ejecuta, bajo el control del sistema operativo, pero independiente de otros procesos. Un esquema general del modelo de implementación de BORPH se muestra en la fig. 2.3; en ella se muestra la diferencia existente entre los elementos que conforman la plataforma hardware y el área del hardware que virtualmente se asocia al sistema operativo a través de lo que ellos llaman *Hardware Reconfigurable Region*.

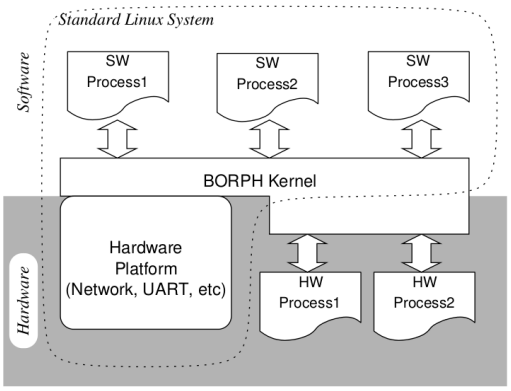


Figura 2.3: Esquema general de BORPH.

También implementa un archivo ejecutable tipo *BOF* (*BORPH Object File*) tal y como se muestra en la figura 2.4. Además, a nivel del núcleo del sistema operativo, un proceso hardware incluye una entrada en el directorio `/proc`, un número `PID`, un área de memoria virtual reservada por el sistema operativo, tiene asociada una entrada en la cola de procesos del planificador de tareas de Linux.

[3]Diseños hardware para FPGAs que pueden ser compartidos entre los investigadores.

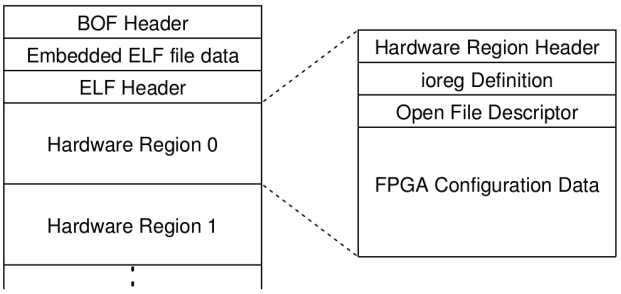


Figura 2.4: Estructura del BORPH Object File.

BORPH se implementa sobre la plataforma de desarrollo *Berkeley Emulation Engine* en su versión 2 (BEE 2) la cual integra cinco FPGAs.

La fig. 2.5 muestra el esquema general de implementación con los detalles para la unidad de control y para los FPGAs asignados a la ejecución de procesos hardware y la fig. 2.6 muestra la implementación física de la tarjeta de desarrollo BEE 2.

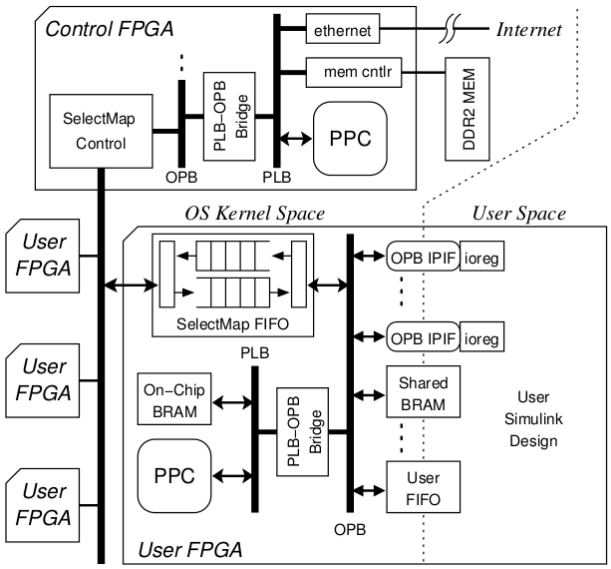


Figura 2.5: Diagrama a bloques del proyecto BORPH implementado en la tarjeta BEE2.

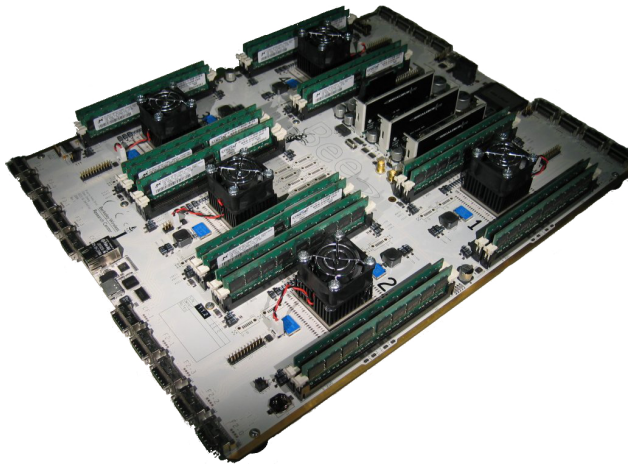


Figura 2.6: Implementación de la tarjeta BEE2.

2.3. Modelo de arquitectura unificada

El modelo de arquitectura unificada propuesto por Q. Deng, *et al.*[13] desarrolló un modelo para la implementación de servicios de reconfiguración parcial de un FPGA en tiempo de ejecución. Esta propuesta implementa la arquitectura mostrada en la fig. 2.7 donde se muestra el procesador PowerPC, dispositivos de almacenamiento, periféricos, un módulo IP Core que integra un bloque *BRAM* y el IP Core *ICAP*. A través de este último IP-Core el sistema operativo es capaz de acceder al subsistema de configuración interno del FPGA para configurar una región del FPGA. La fig. 2.8 muestra una abstracción que representa los principales elementos del sistema. En ella se observan los bloques que representan dispositivos reconfigurables en hardware, el módulo de configuración ICAP y la CPU.

La propuesta de Q. Deng, *et al.* también implementó un planificador de procesos hardware para administrar la ejecución de módulos hardware en el área reconfigurable del FPGA. Este planificador de procesos coopera en las actividades de planificación del sistema operativo a fin de disminuir las tareas asignadas para ejecución en la CPU. Analiza las propuestas de la comunidad de investigadores relativas a la complejidad del algoritmo de planificación.

La planificación de tareas requiere administrar los espacios de reconfiguración hardware así como la modificación en tiempo de ejecución de los puertos de intercomunicación entre el módulo hardware y el bus del sistema por ello, la propuesta de Q. Deng, *et al.* implementó un servicio de planifica-

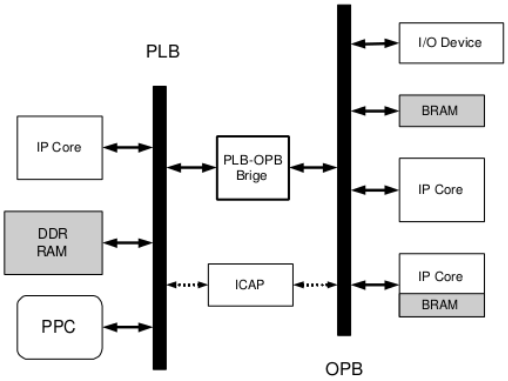


Figura 2.7: Arquitectura del sistema propuesto por Deng *et al.*

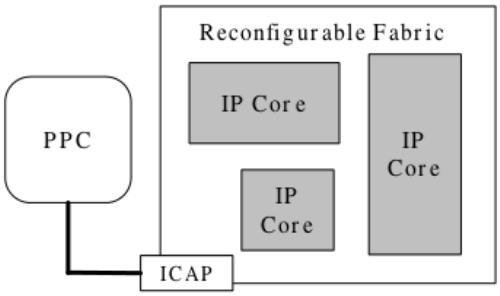


Figura 2.8: Diagrama a bloques de funcionamiento del sistema.

ción hardware; notar que el funcionamiento del planificador hardware difiere del concepto asociado al planificador de procesos software del sistema operativo Linux. La implementación del planificador de tareas hardware requiere hacer modificaciones y extensiones a la funcionalidad provista por el núcleo del sistema operativo con la finalidad de dar soporte al subsistema propuesto y presentar un API consistente con la API presente en un sistema operativo Linux.

Esta propuesta, también implementó un servicio de enrutado para la interfaz de entrada y salida de los módulos funcionales, y la colocación de bloques hardware en el área reconfigurable del FPGA. Ello implicó la modificación de los archivos de configuración parcial asociados a cada módulo funcional para adaptarlos a la interconexión del bus OPB a fin de que a través de este módulo de interconexión se envíen datos desde y hacia el módulo

hardware funcional.

Este modelo propone tratar las tareas hardware y software como procesos, pero estrictamente hablando dentro del contexto del sistema operativo, todas las aplicaciones son procesos software y en algunos casos particulares, los procesos software cuentan con una instancia hardware que los asiste durante el proceso de ejecución. La fig. 2.9 presenta un esquema descriptivo de la relación entre los elementos implementados en hardware y software. La comunicación con la *instancia* hardware se realiza a través de un programa dedicado que implementa el protocolo de comunicación entre la capa software y el módulo IPCore en hardware.

El módulo hardware contiene un elemento funcional que permite a la aplicación software ejecutar parte de su algoritmo en hardware y reducir con ello el tiempo total de ejecución.

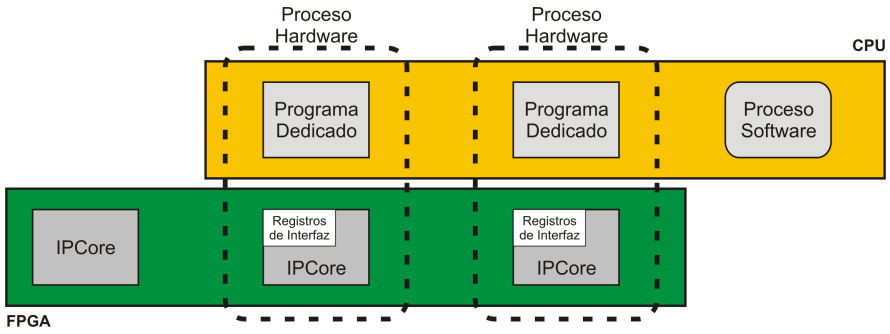


Figura 2.9: Esquema de un proceso hardware y software.

En el proceso de ejecución de una tarea hardware se hace uso de una versión modificada de la llamada a sistema *exec* para que el sistema operativo reconozca y ejecute un tipo especial de archivo denominado HELF (*Hardware Executable and Linkable Format*). Esta llamada a sistema extrae la información de configuración del FPGA y el esquema de comunicación a implementarse entre la capa software de comunicación y el modulo hardware. La fig. 2.10 muestra la estructura de este tipo de archivo ejecutable. En ella se observa la organización interna del *archive*[4]

[4]En el contexto del sistema operativo Linux un *archive* es un archivo que tiene una organización específica para almacenar datos y código ejecutable.

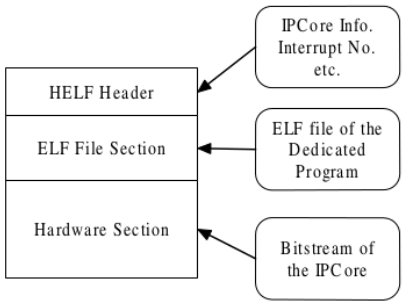


Figura 2.10: HELF Object file.

2.4. Implementación de un SoPC

La implementación de un SoPC (*System on a Programmable Chip*) para la tarjeta de desarrollo ML507 se presentó en [32] por Vaibhawa Mishra *et al.* esta propuesta implementa la arquitectura mostrada en la fig. 2.11; en ella se muestran el procesador PowerPC 440, el módulo IP Core para la configuración de la tarjeta *SysACE*, el módulo para el acceso al puerto ICAP, y un periférico para transferencia de datos entre el sistema desarrollado y la computadora. Este propuesta genera un sistema operativo basado en el kernel de Linux 2.6.34 a través del entorno de desarrollo ELDK[9]. Esta propuesta ejemplifica un modelo de interacción Hardware/Software donde la aplicación de usuario incluye la funcionalidad asociada al módulo hardware a través de un manejador de dispositivos tipo caracter.

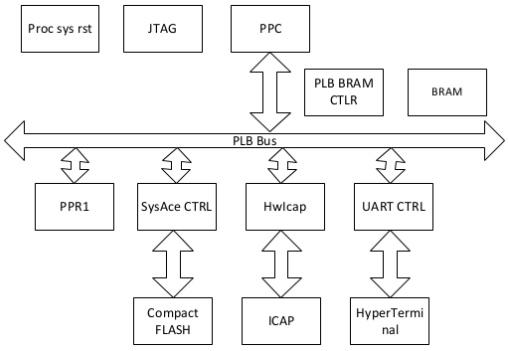


Figura 2.11: Arquitectura implementada por Mishra *et al.*

La tabla 2.1 presenta una síntesis de las diferentes características de implementación y diseño en las propuestas analizadas anteriormente.

Tabla 2.1: Características de las principales propuestas analizadas.

Propiedad	Kwok-Hay[41]	Deng[13]	Mishra[32]
Implementación	5 FPGAs	1 FPGA	1 FPGA
Procesos hardware	Sí	Parcial	No
Planificador Hardware	Parcial	Sí	No
Comunicación inter–proceso	Sí	Parcial	No
Tipo de bus	PLB y OPB	OPB y PLB	PLB
Reconfiguración	Soportada	Sí	Sí
Tipo de archivo ejecutable	Sí, BOF	Sí, HELF	No
Versión del núcleo	2.4.30	2.6	2.6.36
Llamada a sistema	Sí, hsc	Sí, exec	No
Diseño modular	Parcial	No	No

2.5. Resumen

Este capítulo inicia con una breve descripción de las etapas de desarrollo de un SoPC donde se incluyen las tres áreas de investigación para sistemas embebidos y las aportaciones generadas por la comunidad de investigadores. Dichas propuestas aprovechan la sinergia de hardware y software con el objetivo de optimizar los procesos de ejecución y cumplir con las restricciones de diseño. De este conjunto de propuestas, se examina de manera específica y detallada la realizada por H. Kwok-Hay So para la implementación de procesos hardware y la propuesta de Q.Deng sobre un modelo de arquitectura unificada.

Kwok-Hay hace una extensión de la interfaz de aplicación de programa (API) del Sistema Operativo para dar soporte a la ejecución de procesos hardware a través del tipo de archivo ejecutable llamado BOF (*BORPH Object File*). Los procesos hardware implementados en esta propuesta, incluyen diversas características asociadas a los procesos software, entre las que podemos

destacar la capacidad de cambiar su estado de ejecución por el planificador de procesos, herencia de descriptores de archivo y permisos de acceso.

Q.Deng propuso un modelo de arquitectura unificada a través del cual una tarea se ejecuta por elementos hardware y software dentro de un mecanismo común. Sin embargo, una tarea hardware tiene asociado un proceso software del cuál depende. La interacción entre hardware y software se da a través de un manejador de dispositivo. La aplicación de usuario se encapsula en un nuevo tipo de archivo ejecutable llamado HELF (*Hardware Executable and Linkable Format*) que además del código de usuario, incluye parámetros para la configuración del protocolo de comunicación así como un segmento de código de configuración del FPGA que implementa la funcionalidad requerida por la aplicación.

Capítulo 3

Marco teórico

Este capítulo contiene los fundamentos teóricos más importantes relativos al proceso de diseño e implementación de un SoPC (*System on a Programmable Chip*). Inicialmente se presentan definiciones, características, aplicaciones y diferencias entre sistemas embebidos y SoPC. También se examinan los conceptos de diseño de la arquitectura de sistema para un SoPC, tras lo cual se estudian las propuestas de la comunidad de investigadores para la generación de arquitecturas modulares y se examinan las repercusiones derivadas de los requerimientos de un sistema operativo tipo Linux en el proceso de diseño de un SoPC. También se analizan los conceptos de sistemas operativos relativos a la integración de módulos kernel y las propuestas de la comunidad de investigadores para la interacción de elementos Hardware/Software.

3.1. Sistemas Embebidos y SoPC

El uso de sistemas embebidos es una realidad en nuestros días. Los encontramos en automóviles, celulares, refrigeradores, teléfonos inteligentes, televisores, asistentes personales, reproductores de vídeo, UMPCs, relojes, consolas de videojuegos, etc.

La creciente necesidad por aumentar la capacidad de cómputo y disminuir el consumo energético ha llevado a los investigadores a explorara nuevos paradigmas en la ejecución de programas en un sistema embebido.

La industria privada está empujando el desarrollo de estos sistemas; Intel, AMD, Xilinx, Altera, Lattice y Texas Instruments son algunos de los mayores fabricantes de circuitos integrados que ha apostado por el desarrollo de

sistemas embebidos.

La mayoría de ellos ha puesto a disposición de sus clientes conjuntos de herramientas software/hardware a fin de colaborar en el desarrollo, integración e implantación de estos sistemas. La gran apuesta está en los sistemas móviles, por ello es posible encontrar en el mercado una gran variedad de componentes para la integración de estos sistemas.

El desarrollo de software para sistemas embebidos se ha distribuido a lo largo y ancho del planeta. Los polos de desarrollo tecnológico se pueden observar en Estados Unidos, China, India, Alemania, Francia, Japón, Inglaterra, Holanda y Brasil. A diferencia del diseño hardware, el diseño software no está restringido por las fronteras políticas o geográficas de los países ello ha derivado en la conformación de una comunidad global de desarrolladores de software.

Google, Intel y Nokia, son algunos ejemplos dentro de la iniciativa privada que han explotado los beneficios del uso de software libre al desarrollar sistemas operativos embebidos basados en Linux. Android, WebOS, MobLin y MeeGo son ejemplos concretos de los beneficios que se logran al usar software libre.

El uso de un FPGA como elemento principal de un sistema embebido permite al desarrollador un mayor grado de flexibilidad en la implementación abriendo un nuevo campo de aplicaciones. Esta nueva área es tan importante y prometedora que por primera vez, Intel ha decidido integrar un FPGA a uno de sus procesadores.[10] Stellarton es la familia de procesadores que se presentó en el CES 2010 en el que se integran un CPU Intel Atom y un FPGA Arria GX 2 de Altera dentro del mismo empaquetado. El prototipo de esta nueva tecnología se muestra en la figura 3.1.

Xilinx por su parte ha fomentado el desarrollo de su EPP (Extensible Processing Platform por sus siglas en inglés) la cual se basa en la familia de FPGAs Zync. Esta familia de FPGAs incluye un Procesador ARM junto con el FPGA. Este enfoque ya había sido abordado por Xilinx cuando en años anteriores diseño y fabricó los FPGAs Virtex donde se integran hasta dos procesadores PowerPC al FPGA durante el proceso de fabricación.

Si bien estas tecnologías son muy prometedoras, la realidad es que ellas no están disponibles en México. Sin embargo es posible generar una infraestructura para la comprobación de los nuevos paradigmas para el desarrollo de sistemas embebidos. Ello permitirá a los investigadores experimentar y verificar las propuestas derivadas de los proyectos de investigación en desarrollo. Mediante el uso de la tarjeta de desarrollo ML507, que incluye un FP-

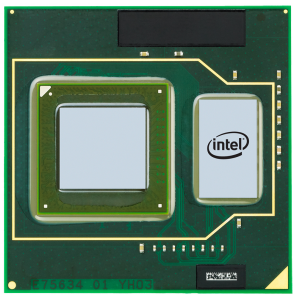


Figura 3.1: Prototipo del Procesador Intel Atom y el FPGA Arria GX 2 de Altera.

GA XC5VFX70TFFG1136, se realizará la implantación de una plataforma hardware y un sistema operativo Linux. En este sistema, se implementarán funciones hardware y capas de servicios software para ejemplificar el proceso de desarrollo de funciones de bibliotecas hardware para sistema embebidos usando la metodología propuesta en el siguiente capítulo.

La presente investigación establece una base para el uso de sistemas embebidos en la solución de problemas complejos donde se requiera cumplir con requerimientos específicos. Este trabajo también presenta una alternativa para acelerar los procesos de ejecución software al capturar llamadas a funciones de bibliotecas dinámicas las cuales se pueden ejecutar en software o hardware. Este procedimiento podrá aplicarse a sistemas existentes donde las aplicaciones que se deseen acelerar, hagan uso de bibliotecas dinámicas durante su ejecución. No se pretende re-inventar la rueda cada vez que sea necesario implantar una solución a un problema específico. Por el contrario, se usarán como base los conocimientos existentes, a fin de adaptar o desarrollar nuevos conocimientos para resolver las necesidades y requerimientos actuales, por ello la metodología propuesta genera una capa de servicios intermedios. El kernel del sistema operativo Linux permite el acceso a su código fuente sin restricciones. Al igual que el kernel de Linux, otros programas son desarrollados bajo la filosofía del software libre. Esta filosofía permite el acceso al código fuente, promueve la inclusión de más desarrolladores a fin de conocer la constitución interna de dicho software, coadyuvando en la expansión del conocimiento pero también fomentando el auto–aprendizaje a través del estudio del código fuente y de la forma en como se implementaron.

El trabajo desarrollado por otros grupos de investigación o por otros colaboradores del software libre enriquece el universo de aplicaciones, innovaciones e implantaciones de soluciones para diversas plataformas. Esto se ha visto reflejado en el creciente número de plataformas que operan con Linux como sistema operativo, también se ve reflejado en la colaboración global para el desarrollo de software. Los fabricantes de hardware han coadyuvado al desarrollo de software libre no sólo permitiendo el acceso a información técnica relativa a los modos de operación de sus dispositivos sino también como patrocinadores de diversos proyectos.

El acceso a los detalles técnicos del software libre, permite el análisis del código fuente de las aplicaciones coadyuvando al desarrollo de conocimientos especializados en el área. Ello permite también que una vez conocida la forma como funciona un determinado software sea posible adaptarlo a nuestro hardware o modificar hardware y software a fin de implantar la solución a un problema específico. Con ello se incursiona en el desarrollo e implantación de sistemas embebidos a fin de sentar las bases para su aplicación futura en proyectos comerciales y para el desarrollo de conocimientos dentro de las universidades.

3.2. Arquitectura hardware de un SoPC

Los sistemas SoPC tienen recursos hardware limitados y por ello la funcionalidad provista por la plataforma hardware difiere de la funcionalidad presente en una computadora de propósito general.

Un sistema de propósito general integra al menos un núcleo de procesamiento, elementos de almacenamiento, dispositivos de entrada/salida de datos, contadores, temporizadores, controladores de interrupciones, unidad de manejo de memoria y dispositivos de comunicaciones. El amplio ecosistema de aplicaciones y la gran cantidad de información generada alrededor de estas plataformas hacen de estos sistemas una opción para la implantación de sistemas embebidos. Sin embargo, este tipo de plataformas no siempre puede cumplir con las restricciones de tiempo de ejecución o tamaño de implementación necesarias en la mayoría de aplicaciones específicas. Más aún, tras haber sido integrado, el sistema embebido permanece estático en el sentido en que su arquitectura no puede modificarse.

Es por ello que los SoPC son utilizados en la implementación de sistemas donde la adaptabilidad es una condición imperativa. Esta cualidad hace más

atractiva la propuesta generada por un SoPC para la implementación de sistemas de propósito específico. Aunado a ello, el sistema puede beneficiarse de los esfuerzos realizados por la comunidad de desarrollo de software libre y la industria, quienes en conjunto han desarrollado herramientas que permiten integrar en un menor tiempo un SoPC en comparación con el tiempo que tomaba integrarlo hace un par de años. Estas herramientas permiten también, adaptar la arquitectura del sistema a los distintos requerimientos del SoPC; es por ello que esta sección explora los detalles de una arquitectura SoPC.

El grupo de desarrollo de sistemas embebidos en IBM propuso en [21] un modelo de arquitectura para un SoPC basado en el procesador PowerPC el cual se muestra en la fig. 3.2; en ella se observa la unidad central de procesamiento, la unidad de punto flotante conectada a través de un bus específico, los buses de interconexión PLB y OPB, además de algunos periféricos.

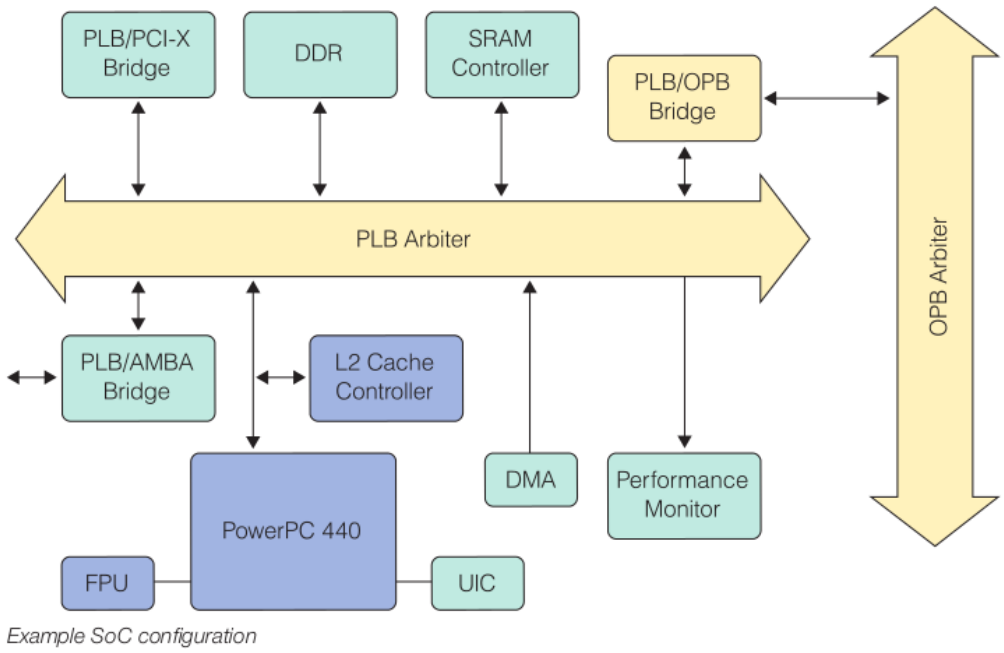


Figura 3.2: Propuesta de IBM para un SoPC.

La arquitectura de estos sistemas dependen en gran medida de la arquitectura del procesador. Debido a que la tarjeta de desarrollo ML507 de Xilinx incluye un procesador PowerPC 440, se analizará la arquitectura de un SoPC basado en este tipo de procesador. De manera general, la plataforma hard-

ware de un SoPC está integrada por:

- Procesador
- Elementos de almacenamiento
- Dispositivos de entrada/salida
- Señales de sincronización.

3.2.1. Arquitectura del procesador PowerPC

El PowerPC 440 es un procesador RISC de 32 bits *Big Endian*, superescalar, con emisión, ejecución y terminación fuera de orden, con un *pipeline* de siete etapas, soporte para la ejecución de dos instrucciones por ciclo de reloj[1], utiliza una caché para datos y otra para instrucciones e incluye una unidad de manejo de memoria.

Este procesador pertenece a la familia de procesadores PowerPC de IBM y está optimizado para su aplicación en SoPC, incluye una interfaz JTAG, un puerto FIFO para captura de trazas de ejecución[2], soporta la diferenciación de producto a través de la inclusión de una unidad de procesamiento auxiliar, es compatible con el set de instrucciones del IBM PowerPC para facilitar la migración de software. La figura 3.3 muestra el diagrama de bloques de la arquitectura del procesador PowerPC y la tabla 3.1 muestra un resumen de las principales características de esta arquitectura.

La arquitectura del procesador PowerPC define los tipos de datos escalares (enteros) como se muestra en la tabla 3.2. Esta arquitectura también define cinco tipos de registros: de propósito general (*General Purpose Register*, GPR), propósito especial (*Special Purpose Register*, SPR), de control (*Device Control Register*), de la máquina de estados (*Machine State Register*) y registros de condición (*Control Register*).

A nivel de micro-arquitectura, el procesador PowerPC especifica dos niveles de ejecución, modo usuario y modo supervisor. Cuando un programa es ejecutado en modo usuario se restringe el acceso a la mayoría de los registros

[1]Se deberá tomar en cuenta esta característica para evitar el re-ordenamiento de instrucciones durante las transferencias de datos entre el procesador y el módulo funcional hardware.

[2]Traza de ejecución también llamado en la literatura inglesa *trace execution*.

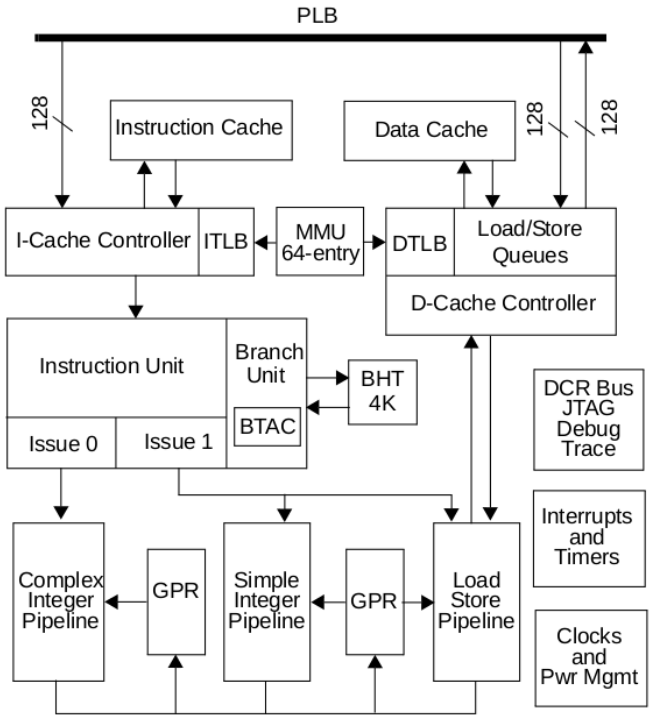


Figura 3.3: Arquitectura del procesador PowerPC.

del sistema a fin que las aplicaciones estén protegidas ante cambios en los valores de los registros como resultado de la ejecución de otros programas. Sólo aquellos programas ejecutados en modo supervisor pueden acceder a todos los registros presentes en el procesador. La tabla 3.3 muestra los registros disponibles a los programas ejecutados en modo usuario (también llamado modo protegido).

3.2.2. Unidad de manejo de memoria

El procesador PowerPC incluye en su arquitectura la unidad para gestión de memoria, MMU por sus siglas en inglés (*Memory Management Unit*). En el caso particular del FPGA integrado en la tarjeta de desarrollo ML507, se tiene el procesador PowerPC 440 y una MMU para gestionar los accesos a memoria, pero debe ser configurada previo a su uso.

Esta gestión se realiza a través de la traducción de direcciones virtuales en direcciones físicas, protección de páginas de memoria y control de la *caché*.

Tabla 3.1: Características del procesador PowerPC.

Elemento	Descripción
Registros	32 Registros de propósito general.
Multiplicador	Multiplicador hardware de 32 bits para enteros.
Divisor	Divisor hardware de 32 bits para enteros.
Saltos	Predictor dinámico de saltos.
Paridad	Detección y recuperación de errores de paridad.
Caché	32 KB de caché para datos y 32 KB para instrucciones.
TLB	64 entradas completamente asociativa.
Bus	Interfaz de bus PLB de 128 bits con un bus de direcciones de 36 bits.
APU	Unidad de procesamiento auxiliar para acoplar coprocesadores.
ISA	Conjunto de instrucciones compatible con IBM PowerPC y soporte para instrucciones de usuario para carga o almacenamiento de 128 bits.
Temporizador	Soporte para temporizador de 64 bits y para temporizador de intervalo fijo.
Contadores	Contadores decrementales de 32 bits.
Depuración	Soporte para depuración mediante JTAG.

Cuando el procesador intenta acceder a una dirección de memoria lógica, la MMU realiza una búsqueda en una memoria local especial llamada TLB por sus siglas en inglés (*Translation Lookaside Buffer*), donde se mantienen las entradas a las páginas más recientemente usadas.

Otros tipos de procesadores como *MicroBlaze* pueden deshabilitar la unidad de gestión de memoria. Esto beneficia el tamaño de implementación pero limita la funcionalidad del sistema.

Un micro sistema operativo adaptado a una plataforma hardware sin MMU tiene un único espacio de direcciones donde se mapean periféricos, memoria del sistema, dispositivos de entrada y salida de datos. En este espacio único, se pueden utilizar directamente los apuntadores para acceder a los periféricos puesto que ellos son mapeados al espacio único de memoria.

Tabla 3.2: Tipos de datos del procesador PowerPC.

Tipo de dato	Tamaño en bytes
Byte	1
Halfword	2
Word	4
Doubleword	8
Quadword	16
String	1 a 128

Tabla 3.3: Registros del procesador PowerPC.

Registro	Tipo	Descripción
GPR0 al GPR31	GPR	General-Purpose Registers
CR	CR	Condition Register
XER	SPR	Integer Exception Register
LR	SPR	Link Register
CTR	SPR	Count Register
SPRG4 al SPRG7	SPR	Special Purpose Registers General
USPRG0	SPR	User Special Purpose Register 0
TBU	SPR	Time Base (Upper 32-bits)
TBL	SPR	Time Base (Lower 32-bits)

Si bien este aspecto simplifica el modelo de operación, las aplicaciones que requieren de un esquema de protección deben emular las funciones realizadas por la MMU para traducir información y gestionar el acceso a las páginas de memoria.

El proyecto μCLinux[1] bajo la dirección de John Williams logró adaptar un sistema operativo basado en el kernel 2.4 a plataformas hardware sin MMU. Esto permite que los sistemas operativos cuya plataforma hardware no incluye una MMU tengan un menor tamaño de implementación sin embargo, la funcionalidad provista por las bibliotecas de funciones del sistema es limitada.

La arquitectura generada por el procesador PowerPC del FPGA Virtex

5 FX70 genera dos espacios de direcciones distintos, uno para los periféricos (dispositivos de entrada/salida de datos) y otro para la memoria general del sistema (Memoria RAM). Para acceder a los puertos de entrada/salida los desarrolladores incluyeron un conjunto de instrucciones que permiten sincronizar la carga y almacenamiento de datos a dispositivos periféricos los cuales se muestran en la tabla 3.4.

La política de almacenamiento de datos que implementa el PowerPC determina cuándo la modificación a un dato deberá ser propagada hacia el espacio de memoria compartida (memoria RAM o dispositivo periférico). La fig. 3.3 muestra los bloques de carga/almacenamiento, la interacción entre la TLB de datos, la caché de datos y los periféricos del sistema. Debido a que la arquitectura implementa una *caché* de datos, las modificaciones a estos no se propagan de manera inmediata hacia los niveles subsecuentes de memoria es por ello que es necesario incluir alguna de las instrucciones listadas en la tabla 3.4.

El procesador copia una línea de datos modificados hacia la memoria compartida (memoria RAM o dispositivo periférico) únicamente si la memoria *caché* es marcada para su remplazo explícitamente ya sea debido a la necesidad de cargar nuevos datos o por una indicación explícita. En una caché tipo *write-through* como la implementada en esta arquitectura, puede hacer escrituras directamente al área de memoria compartida. El remplazo (o vaciado) de la *caché* no requiere escribir al área de memoria compartida.

La tabla 3.4 muestra las instrucciones más relevantes para el proceso de sincronización de carga y almacenamiento de datos para el espacio de direcciones de los puertos de entrada.

3.2.3. Interfaces del procesador

El procesador PowerPC 440 soporta la arquitectura de sistema *IBM CoreConnect* la cual simplifica la adición de dispositivos a través del bus de procesador PLB, la interfaz DCR (*Device Configuration Register*) y la interfaz APU (*Auxiliary Processor Unit*), incluye además un subsistema de administración de energía, administración de las señales de sincronización (reloj) y un puerto para interrupciones.

Hay tres interfaces del PLB independientes de 128-bits, cada una de estas interfaces incluye un bus de direcciones de 36 bits y un bus de datos de 128 bits tal y como se muestran en la figura 3.3; de las tres interfaces, una carga información de la caché de instrucciones mientras que las otras dos soportan

Tabla 3.4: Instrucciones para carga y almacenamiento de datos.

Instrucción	Descripción
`sync`	Sincronización
`msync`	Garantiza la ejecución de `dcbst` y prosigue
`isync`	Sincronización y recarga de datos
`mbar`	Barrera de memoria (*Memory barrier*)
`eieio`	*Enforce In-order Execution of Input/Output*
`stw`	Escribe una nueva instrucción a la caché de datos
`dcbst`	Escribe de la caché de datos hacia la memoria
`icbi`	Invalida la copia de instrucciones viejas en caché

lecturas y escrituras a la caché de datos.

La interfaz DCR proporciona al procesador un mecanismo para configurar otras instancias o periféricos de la arquitectura del sistema sin tener añadir penalizaciones por el tiempo de comunicación entre el procesador y los elementos conectados al bus PLB, sin embargo, requiere de una interfaz especial en el periférico a fin de que esta comunicación sea posible.

El bus DCR es accedido a través de las instrucciones de máquina `mfdcr` y `mtdcr`. El uso de este bus evita la fragmentación del espacio de direcciones al procesar de manera independiente la carga y almacenamiento de información en el bus DCR. La implementación permite la interconexión de dispositivos que operan a diferente frecuencia respecto a la frecuencia de operación del procesador.

La interfaz APU (*Auxiliary Processor Unit*) provee flexibilidad al procesador para añadir coprocesadores altamente acoplados. Permite la definición de instrucciones de usuario las cuales son enviadas a la unidad APU para su ejecución. Este bus provee la funcionalidad necesaria para ser utilizado como interfaz entre la unidad de punto flotante y el procesador. Esta adaptabilidad permite al sistema incorporar módulos especializados para ejecutar operaciones de punto flotante, procesamiento digital de señales, codificadores y decodificadores de audio/video, módulos de criptografía sin embargo el módulo funcional es altamente dependiente de los detalles de implementación por lo que su uso se restringe a las arquitecturas de sistema con soporte para *IBM CoreConnect*.

3.2.4. Arquitectura modular para SoPC

Los sistemas embebidos requieren alta capacidad de procesamiento y a menudo emplean una variedad de funcionalidades diferentes; debido a ello han alcanzado un nivel de complejidad que ha hecho imposible diseñarlos desde cero.

La industria y la comunidad de investigadores ha desarrollado metodologías que permiten agrupar elementos funcionales en módulos *IPCore*, también han propuesto modelos de interconexión para facilitar la integración de los módulos funcionales para formar la arquitectura de un SoPC.

Xilinx ha desarrollado un conjunto de herramientas que asisten al diseñador de sistemas en los procesos de configuración, verificación de dependencias de interconexión, mapeo, traducción, colocación, síntesis y compilación. El entorno de desarrollo *Embedded Development Kit* permite hacer estas tareas mediante su interfaz gráfica, sin embargo, también se provee una interfaz en línea de comandos a través de la cual se pueden automatizar las tareas o procedimientos realizados a través de la interfaz gráfica.

Sin embargo, la creciente necesidad de producir sistemas SoPC con tiempos de desarrollo cada vez menores, ha instado a fabricantes e investigadores a desarrollar herramientas que permitan integrar módulos funcionales y acoplarlos en una plataforma hardware sintetizable dentro del SoPC.

Diversos ejemplos han sido probados en la industria, entre ellos, el más popular es el diseño modular de un sistema dónde cada elemento de éste se puede integrar a la arquitectura debido al concepto de diseño modular con el que fueron creados. Esto reduce considerablemente el tiempo de desarrollo e integración pero aumenta la complejidad de cada módulo y por consecuencia la complejidad general del sistema en desarrollo.

La fig. 3.4 muestra una representación abstracta de la arquitectura de un SoPC donde cada rectángulo representa un elemento funcional encapsulado en un módulo IPCore.

3.2.5. Arquitectura Hardware y Sistema Operativo

La arquitectura de un SoPC depende en gran medida de los requerimientos del problema a resolver sin embargo un SoPC que requiere un sistema operativo deberá añadir algunos requisitos más para permitir el correcto funcionamiento de sistema en su conjunto.

Los sistemas operativos tienen requerimientos muy particulares y espe-

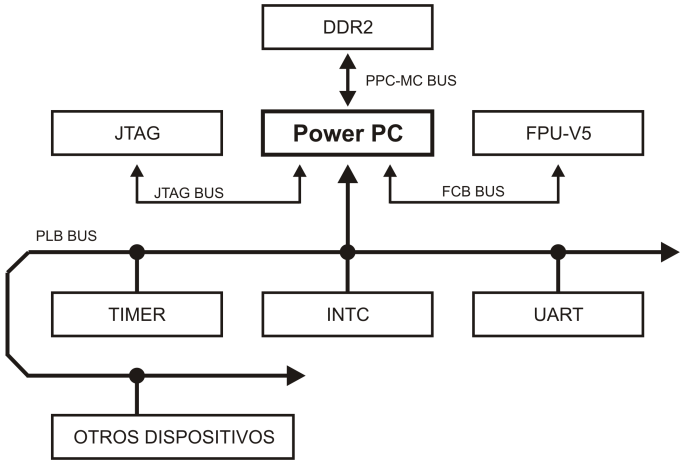


Figura 3.4: Abstracción de la arquitectura de una plataforma hardware.

cíficos en cada caso. Linux es el sistema embebido seleccionado para este proyecto debido a la modularidad de sus funciones y servicios, el reducido tamaño de implementación que puede alcanzar, los servicios de administración de memoria que incorpora, y por la capa de abstracción hardware que implementa.

Linux requiere de una señal de sincronización que periódicamente genere una interrupción al sistema. El propósito de esta interrupción periódica es generar una base de tiempo a fin de que el sistema pueda cuantificar el tiempo que ha transcurrido entre un evento y otro. Sirve también para realizar algunas tareas muy importantes para el sistema operativo como lo es la planificación de tareas. Es a través de esta interrupción periódica que el sistema administra el tiempo en ejecución, cambia el estado de los procesos, crea procesos, sincroniza la carga y almacenamiento de datos y realiza otras actividades administrativas sobre los recursos del sistema en general.

El valor de la variable `HZ = 100` determina el periodo de incremento del temporizador del kernel. Esto significa que el temporizador genera interrupciones al sistema a razón de 100 interrupciones por segundo, dicho de otra forma, el temporizador genera una interrupción cada 10 ms.

El periodo de tiempo con el cual el temporizador del kernel se incrementaba pasó a ser la unidad predeterminada de planificación del sistema o *quantum*, más tarde este valor paso a ser nombrado *jiffy* de manera singular o *jiffies* de manera plural. Esto implica que toda plataforma hardware que utilice Linux como sistema operativo debe incluir un temporizador que

genere interrupciones periódicas.

Linux implementa un modelo jerárquico de acceso a la memoria del sistema. Existe un espacio de trabajo llamado *espacio de memoria del kernel* el cual se utiliza para albergar la funcionalidad crítica del sistema. Esto significa que el área de memoria del sistema tiene acceso a todos los recursos del sistema ya sean privilegiados o no. Es en esta área de memoria donde reside el código del núcleo, y donde también reside el código que implementa los diversos servicios que proporciona el núcleo a las aplicaciones de usuario y a otros procesos del sistema. Debido a la importancia de la administración de la memoria para el sistema, los desarrolladores del Linux han implementado un subsistema de control de acceso a memoria que involucra desde simples macros de verificación de tipo de acceso hasta elaboradas funciones de verificación en base a estructura arbóreas donde los recursos del sistema se concentran. Además de este esquema de protección software, se incluye la funcionalidad provista por la MMU a través de la cual se configura el privilegio de acceso a las páginas de memoria física del sistema y se realiza la traducción correspondiente entre dirección virtual y dirección física. Es por ello que los sistemas que incluyen una MMU en su arquitectura son más afines a la inclusión de sistemas operativos como parte de la solución al problema planteado. Esto supone una ventaja no sólo por el hecho de administrar el acceso y uso de la memoria del sistema sino también por que el sistema operativo proporciona una interfaz común para la implantación de programas y funciones que dan solución a la problemática planteada.

Existen plataformas que incluyen funciones avanzadas para depuración. Durante la fase de desarrollo de un SoPC, los servicios de depuración resultan ser de gran utilidad a la hora de verificar el comportamiento del sistema y en la búsqueda de errores de programación. Por ello en diversas plataformas hardware se incluyen elementos funcionales que facilitan estos procesos; entre estos elementos podemos mencionar a FIFOS, contadores, interfaz JTAG para depuración como las más importantes.

3.3. Modelos de interacción

Los modelos de interacción Hardware/Software son el resultado de procesos de investigación tanto de industria como de la comunidad de investigadores. Debido a ello, esta sección describe los modelos de interacción más relevantes a este proyecto.

Es frecuente encontrar casos que los que una misma tarea se puede implementar como un procedimiento software o como un módulo en hardware; sin embargo, ambas soluciones difieren en rendimiento y costo de implementación, lo cual debe ser considerado durante la etapa de diseño del SoPC.

Debido a la disponibilidad de dos alternativas para la implementación del mismo algoritmo, ya sea a través de un módulo funcional hardware o de un conjunto de procedimientos software, existen dos posibilidades para solucionar el mismo problema. Estas soluciones de manera individual utilizan, en el primer caso la implementación de módulos hardware, y en el segundo caso la implementación de rutinas software para solucionar el problema dado. Sin embargo, en el entorno de desarrollo de sistemas embebidos los requerimientos de las aplicaciones son muy demandantes por lo que una implementación hardware sería la mejor opción, sin embargo, el costo de fabricación de un ASIC en muchos de los casos rebasa el costo de producción cuando el SoPC está en su fase de desarrollo.

Para abordar este problema, se han desarrollado metodologías que permiten al diseñador implantar soluciones Hardware/Software. Los modelos de ejecución Hardware/Software integran elementos hardware a los procedimientos software los cuales pueden ejecutarse concurrentemente en el mismo sistema. Sin embargo nuevos tipos de problemas surgen al implementar estos modelos.

A nivel de sistema operativo, la interacción con elementos funcionales hardware se puede desarrollar en tres formas distintas. La fig. 3.5 presenta los modelos de interacción Hardware/Software. Las figs. 3.6, 3.7 y 3.8 detalla el modelo de interacción estándar, el modelo de interacción en base a llamadas a funciones de bibliotecas y el modelo de interacción en base a llamadas a sistema respectivamente.

El modelo de la fig. 3.6 describe la interacción a nivel de sistema operativo con un elemento funcional hardware. Este modelo es el más ampliamente utilizado por los dispositivos periféricos en Linux. Para ello un programa de usuario hace uso de las funciones de la biblioteca estándar de C *fopen(), fclose(), fread(), fwrite()* a través de las cuales se abrirá un dispositivo especial (tipo caracter o bloque) para transferir información entre sistema↔periférico.

El modelo de la fig. 3.7 usa llamadas a funciones de bibliotecas dinámicas desde la aplicación de usuario. El código que implementa la función de biblioteca está encargado de gestionar la transferencia de información entre sistema↔función hardware.

El modelo de la fig. 3.8 hace uso de la funcionalidad provista por el sistema

para implantar llamadas a sistema mediante las cuales se realizará la transferencia de datos entre sistema↔función hardware. Este modelo en particular puede no hacer uso del módulo manejador de funciones hardware debido a que la implementación de la llamada a sistema se ejecuta en el espacio de memoria del kernel. La llamada a sistema puede incluir las funciones gestión y transferencia de información entre los elementos funcionales en cuestión.

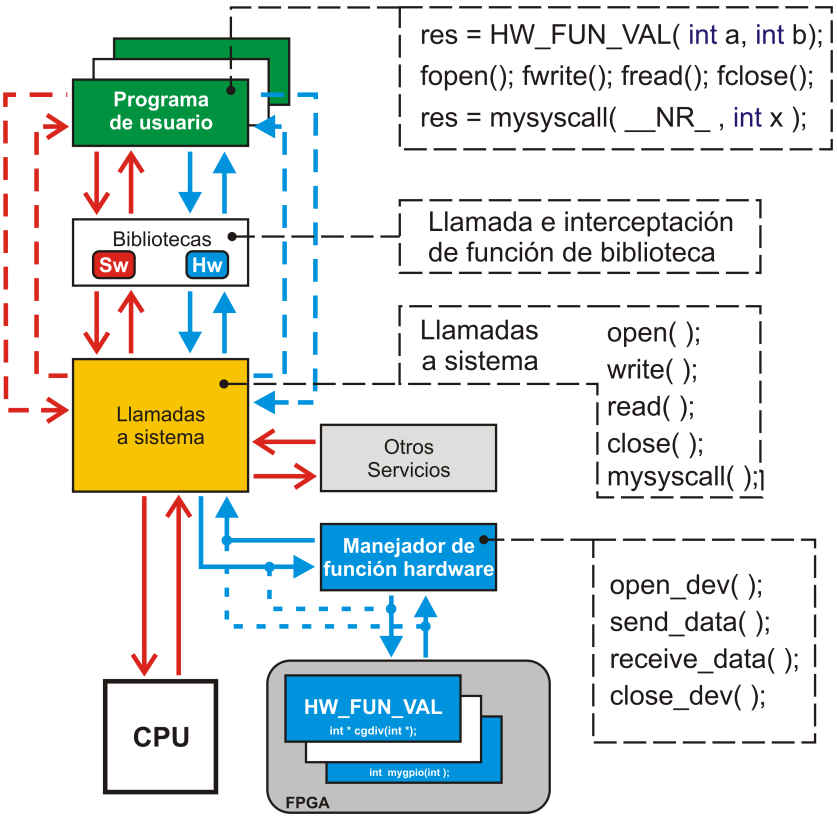


Figura 3.5: Modelos de interacción Hardware/Software.

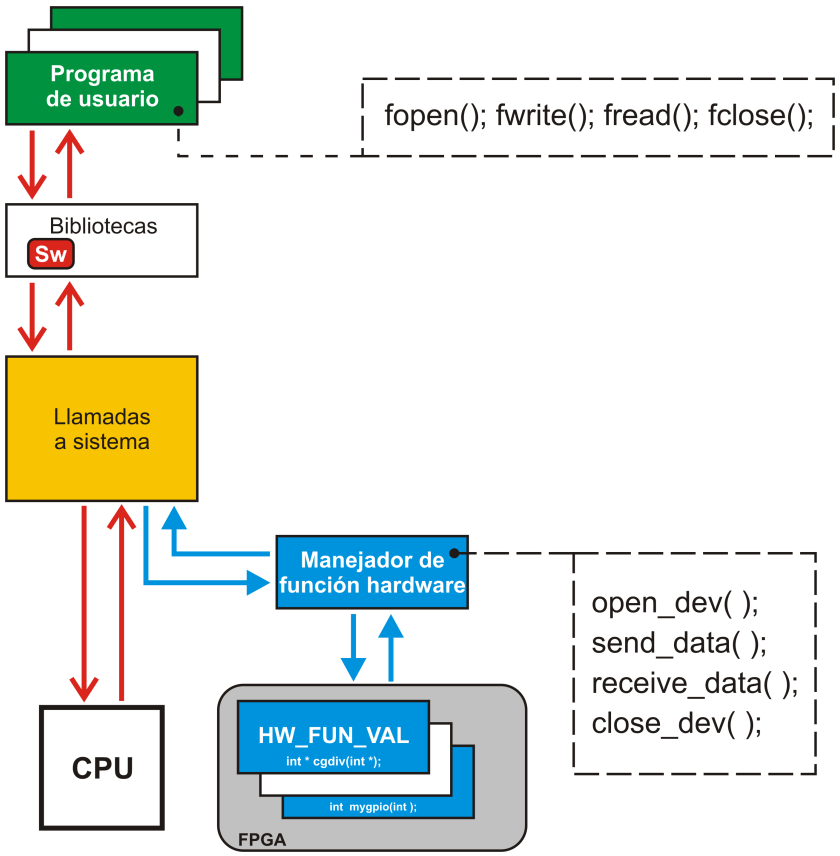


Figura 3.6: Modelo estándar de interacción Hardware/Software.

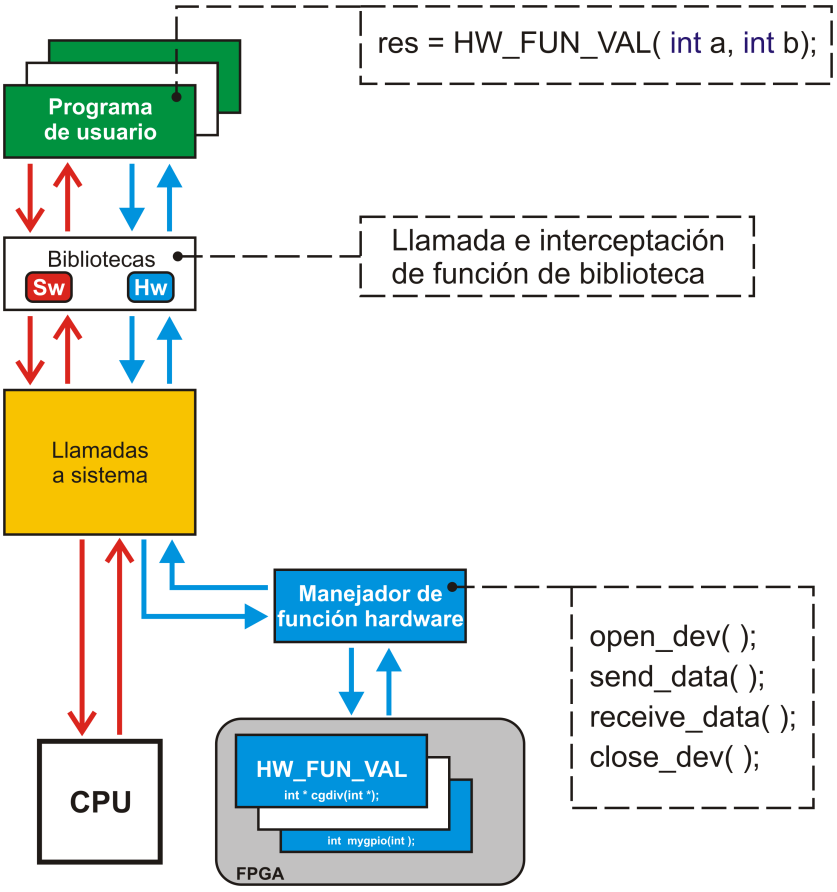


Figura 3.7: Modelo de interacción Hardware/Software en base a llamadas a funciones de biblioteca.

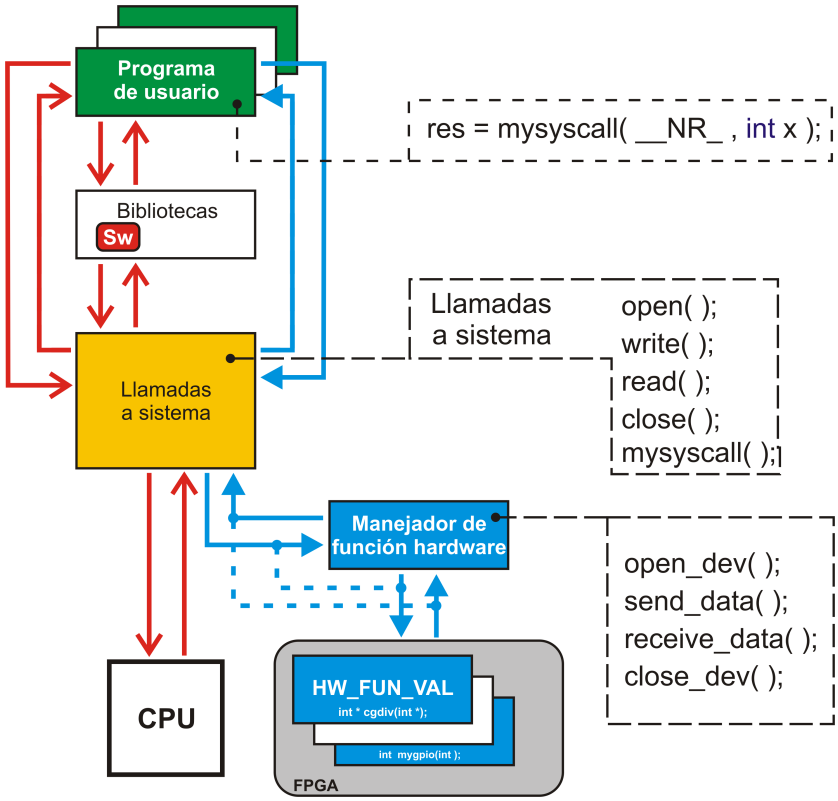


Figura 3.8: Modelo de interacción Hardware/Software en base a llamadas a sistema.

3.4. Sistema Operativo Embebido

Un sistema operativo embebido tiene restricciones en el tamaño de la implementación por lo que el núcleo que se carga inicialmente debe contener el mínimo número de elementos posibles. A través de la funcionalidad que provee el núcleo de Linux es posible aumentar dicha funcionalidad a través de módulos de kernel.

3.4.1. Funciones de un módulo kernel

El núcleo de Linux tiene la capacidad de cargar módulos kernel en tiempo de ejecución para complementar las funciones y servicios que implementa. Durante el proceso de arranque se carga el núcleo del sistema en memoria. Este núcleo debe contener la funcionalidad necesaria para configurar los principales periféricos del sistema y administrar los recursos hardware de la plataforma. Sin embargo, cuando un usuario solicita alguna característica que no está presente en el núcleo residente, se carga dinámicamente en memoria un módulo kernel el cuál contiene código que se ejecutará en el espacio de memoria del kernel.

Hay seis funciones principales para los módulos kernel de Linux y se describen a continuación

Controladores de dispositivos: Un controlador de dispositivo es diseñado para un elemento particular de hardware. El núcleo del sistema utiliza los controladores de dispositivos para comunicarse con esos elementos sin tener que conocer los detalles de operación del hardware.

Manejadores de sistemas de archivos: Estos manejadores interpretan el contenido de un sistema de archivos, el cual típicamente tiene archivos y directorios organizados de una manera específica en él; este elemento de almacenamiento puede ser un elemento local, como un disco duro o virtual, como un servidor remoto; sin embargo, para el sistema operativo pareciera un subsistema local debido a la funcionalidad de traducción y acceso que proveen este tipo de módulos.

Llamadas a sistema: Los programas de usuario utilizan de manera directa o indirecta las llamadas a sistema. Una llamada directa, pasa los argumentos de la función como parte de la llamada mientras que una llamada indirecta hace uso de otros mecanismos los cuales en algún momento se convertirá en una llamada a sistema. De manera particular,

las llamadas a funciones de biblioteca en algún momento requieren de hacer llamadas a sistema de forma directa o indirecta.

Controladores de red: Este tipo de controladores es especial ya que no tiene asociado ningún archivo dentro del sistema de archivos de Linux. Por ello las funciones que provee sólo son accesibles a través de la API que provee el núcleo. Sirven para interpretar los protocolos de comunicación de la red; este tipo de módulos genera y consume cadenas de datos en las distintas capas de las funciones de red del núcleo.

Disciplinas de líneas TTY: Son esencialmente variaciones de los manejadores de dispositivos con opciones específicas para el manejo de dispositivos tipo terminal.

Intérpretes de ejecución: Un intérprete de ejecución carga y ejecuta un archivo ejecu. Linux fue diseñado con la capacidad de ejecutar diferentes tipos de formatos de archivos, cada uno de ellos tiene asociado su propio intérprete de ejecución.

En el caso particular de este proyecto, se utilizarán módulos kernel para proveer la funcionalidad requerida para la interacción entre elementos hardware y software en el sistema.

3.4.2. Servicios de reconfiguración hardware

Walder y Platzner[46] desarrollaron un *Sistema Operativo con Reconfiguración Hardware* el cual utiliza un modelo de desarrollo *top-down*. En él se implementan estructuras y funciones de las capas de servicios del sistema operativo en hardware para generar la funcionalidad habitual pero ahora usando elementos del FPGA. Entre las funcionalidades más importantes destacan: el planificador de tareas, el administrador de recursos lógicos, el administrador de tiempos de ejecución, el subsistema de almacenamiento de *bitstreams*, el subsistema de extracción y cambio de contexto, el subsistema de posicionamiento de tareas hardware, el módulo *puente* entre el sistema operativo y el FPGA, y el módulo de *manejo* del FPGA.

A pesar del nivel de integración de los servicios hardware/software y tomando en cuenta la complejidad en la implementación de la capa de servicios del sistema operativo, esta propuesta depende de la pericia del desarrollador

para hacer una correcta segmentación de los bloques funcionales que deberán ser implementados en hardware.

Sin embargo, establecen un protocolo específico para la interacción entre los módulos hardware y el sistema operativo que necesariamente involucra transportar funcionalidad del sistema operativo al hardware. Los requerimientos particulares de este sistema operativo precisan la descomposición de una aplicación en tareas y objetos que puedan ser asociados directamente a alguna estructura hardware como *temporizadores*, *buffers*, *multiplicadores hardware*, etc.

3.4.3. Separación hardware/software

Burns *et al.*[8] propusieron un Sistema con Reconfiguración Dinámica en Tiempo de Ejecución. A través de él se analizan los diferentes modelos de interacción entre hardware y software; como resultado de este análisis se propone una separación entre la aplicación software y los módulos hardware especializados. Esta separación beneficia a las aplicaciones del Sistema Embebido pues el acceso y uso de los módulos funcionales no se restringe al uso específico de una aplicación particular sino que permanecen externas y disponibles a otras aplicaciones.

Debido al alto costo en tiempo que supone la síntesis de circuitos hardware para un FPGA, ellos sugieren que la mayor parte de la funcionalidad de un módulo hardware sea sintetizada *a priori* ya que no es factible hacerlo en tiempo de ejecución. Ellos sugieren la separación parcial de elementos dinámicos y estáticos, para que estos últimos sean generados durante el proceso de síntesis del módulo y los elementos dinámicos se reconfiguren en tiempo de ejecución. Para ello, utilizan una interfaz de interconexión estática para los puertos del módulo hardware, mientras que la funcionalidad en ellos se mantiene dinámica. Además de ello, los módulos funcionales son sintetizados *a priori* en base a un conjunto de reglas que definen el pre-ruteo y la pre-localización en un área determinada dentro del FPGA. Para poder usar circuitos dinámicamente, es decir, en tiempo de ejecución, los componentes funcionales deben estar pre-ruteados y pre-ubicados en un área asignada y sólo se necesita habilitar los puertos de interconexión entre la parte dinámica y la parte estática.

El objetivo de esta especificación es desarrollar módulos hardware cuya interfaz de interconexión se mantiene controlada a fin de permitir el inter-

cambio de módulos hardware en tiempo de ejecución.

3.4.4. Arquitectura hardware para cómputo reconfigurable

La propuesta de Wigley y Kearney [47] se basa en la creación de un sistema operativo para cómputo reconfigurable donde un módulo hardware independiente al sistema implementa el protocolo de comunicación entre diversos FPGAs donde cada uno puede atender de manera dinámica la ejecución de una tarea para un conjunto de entradas de datos específicas lo que hace al sistema dinámico en el tiempo ya que la disponibilidad de los recursos puede variar en función del tiempo y de la aplicación.

Sin embargo al igual que Burns *et al.*[8] parten del uso de módulos funcionales pre-sintetizados y adaptados para su re-localización. Aún cuando el sistema de administración tanto de áreas reconfigurables como de dispositivos reconfigurables representa un avance significativo para la implementación de elementos funcionales hardware, esta propuesta carece de adaptabilidad para poder ser implementada en otra arquitectura.

Wigley y Kearney trabajan en base a un modelo multi-usuario, como ellos lo llaman, pero se refieren a un modelo en el que múltiples aplicaciones hacen uso de los recursos lógicos del FPGA. En este entorno de operación la disponibilidad de recursos del FPGA cambia dinámicamente con tiempo. Por ello, el diseño tradicional donde un módulo funcional es asignado, ubicado y enrutado *a priori* no es válido en este esquema de operación ya que no se conoce con certeza la disponibilidad de algún recurso particular dentro del FPGA. Para resolver este problema, Wigley y Kearney proponen que el sistema operativo sea el encargado de hacer estas *adaptaciones* en tiempo de ejecución para poder segmentar, ubicar y enrutar las tareas que así lo requieran permitiendo con ello la co-existencia de múltiples aplicaciones en el FPGA.

La propuesta de Wigley y Kearney introduce el concepto de *reconfiguración dinámica* mediante el cual se realiza el particionamiento de una tarea en múltiples bloques funcionales con la finalidad de que los sub-bloques cooperen de manera secuencial o concurrente durante el proceso de ejecución del módulo hardware. Esta idea surge a partir de la existencia de bloques funcionales que no pueden ser implantados dentro del FPGA ya sea por el número de elementos lógicos requeridos o por el área disponible dentro del FPGA.

Para resolver este problema, se propone la segmentación del módulo funcional en bloques más pequeños que sean re-localizables a fin de permitir la síntesis del mayor número de ellos para conformar la funcionalidad del módulo original. Sin embargo, este particionamiento implica el uso de interfaces de comunicación entre los diferentes sub-módulos y tiene en impacto en la intercomunicación entre ellos.

Para ello, representan a toda aplicación software del sistema operativo a través de un grafo de tareas donde cada nodo representa un operador o una función y los arcos dirigidos representan la dependencia de datos entre los nodos. Este concepto se utiliza para segmentar un proceso grande en pequeños componentes tal y como se haría al segmentar una aplicación en páginas de memoria para que su carga se realice por segmentos en lugar de cargar todo el código a ejecutar aún cuando este no sea ocupado.

Las tareas tradicionalmente asociadas a un sistema operativo como el cargador de programas, el planificador, el sistema de memoria virtual, comunicación inter-proceso y entrada/salida de datos deben, necesariamente, ser adaptadas para su correcto funcionamiento.

Los servicios de reconfiguración que proporciona el sistema operativo generado en base a la arquitectura propuesta son: servicios de asignación, particionamiento dinámico, colocación y enrutamiento; aunque estos últimos se realizan sólo para la interconexión de módulos funcionales a la arquitectura del sistema.

Wigley y Kearney identifican las restricciones generadas al agregar la capacidad de reconfiguración dentro del sistema operativo y de manera general dentro del sistema embebido. Las necesidades de ubicación y ruteo de los módulos funcionales son, de manera general, distintas a las habilidades de ubicar recursos y rutear señales en las herramientas de diseño convencionales. Además de ello, se sugiere tomar en cuenta el hecho de que una vez cargado el módulo hardware, éste empieza a funcionar independientemente del estado en el que se encuentre la aplicación software; debido a esto se sugiere cargar los datos de la aplicación en los `BRAM` (*Bloques RAM*) asociados al módulo hardware o agregar un mecanismo que permita su carga en tiempo de ejecución.

3.5. Resumen

Este capítulo presentó diversos conceptos teóricos relativos al proceso de diseño e implementación de un SoPC (*System on a Programmable Chip*). Durante la exposición de estos conceptos se analizaron los modelos de interacción Hardware/Software y los modelos de arquitecturas para sistemas SoPC. También se presentaron los conceptos de diseño de un sistema operativo que incluye servicios de reconfiguración a través de módulos kernel y se analizaron las propuestas para integrar elementos funcionales hardware a la plataforma del SoPC.

Capítulo 4

Metodología propuesta

En base a las metodologías planteadas por la comunidad de investigadores para el desarrollo de elementos funcionales en SoPC, se propone una metodología que utiliza los procedimientos existentes en otras propuestas para la generación de un SoPC, pero además añade procedimientos para la generación de una biblioteca de funciones hardware y hace especial énfasis en la capa de servicios software a nivel de sistema operativo.

La metodología propuesta consta de los siguientes procedimientos:

1. Generar la Plataforma Hardware
2. Generar e integrar los módulos de funciones hardware
3. Parametrizar la arquitectura hardware
4. Generar Sistema Operativo
5. Generar el módulo manejador para las funciones hardware
6. Generar la aplicación de usuario que interactúa con la función hardware

En base al análisis de los modelos de interacción se determinó la necesidad de implementar el *modelo de interacción Hardware/Software estándar* y el *modelo de interacción Hardware/Software basado en llamadas a funciones de bibliotecas dinámicas* los cuales se presentan en la fig. 4.1.

Además, se incluye el concepto de interceptación de llamadas a funciones. Este concepto es útil en los casos en los cuales no se tiene acceso al código fuente del programa de usuario. El método propuesto, captura llamadas a

funciones de bibliotecas dinámicas para atenderlas por otras funciones software o por funciones hardware.

Durante el proceso de desarrollo de un SoPC se genera una aplicación que da solución al problema específico para el cual el SoPC fue creado. En este contexto se presenta la "*Metodología para el desarrollo de bibliotecas de funciones hardware*" para describir los procedimientos necesario al desarrollar funciones hardware y hacer que uno o más de los módulos hardware colaboren en el proceso de ejecución de una tarea específica.

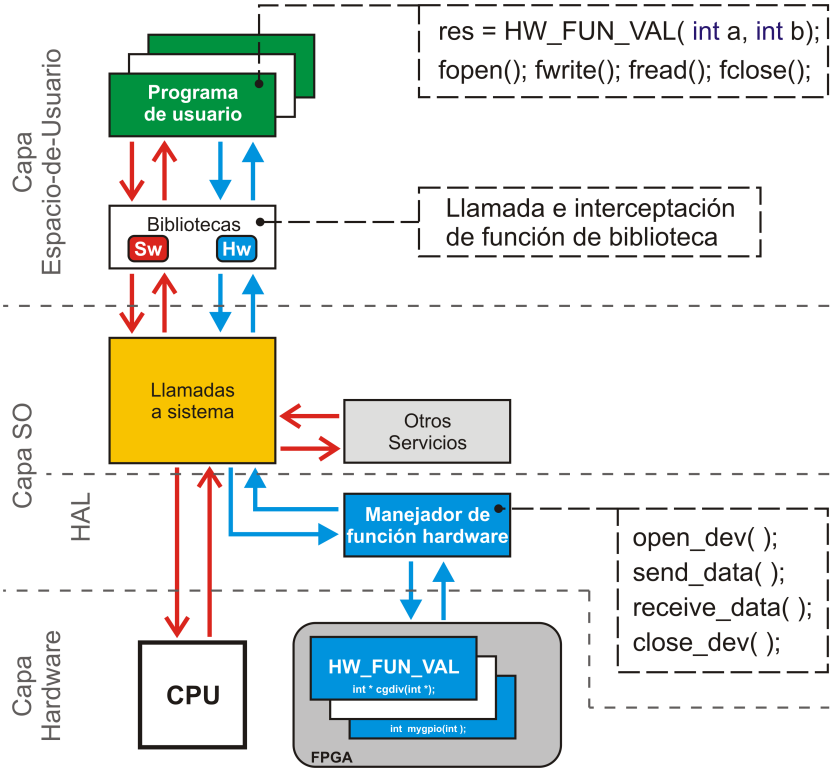


Figura 4.1: Modelo de interacción Hardware/Software implementados por la metodología propuesta.

4.1. Generación de la Plataforma Hardware

El SoPC de este proyecto se implementa en la tarjeta de desarrollo ML507 la cual contiene un FPGA Virtex 5 FX70. El FPGA incorpora un procesador PowerPC 440 y debido a ello se generará una plataforma hardware afín al procesador.

Las herramientas *EDK* y *SDK* de Xilinx son utilizadas para la integración de la plataforma hardware básica y en la configuración, desarrollo e integración de las funciones hardware propuestas. La fig. 4.2 muestra el primer paso para la generación de la plataforma hardware usando el *BSB* (*Base System Builder*) de Xilinx. Dicha figura establece el tipo de bus principal del sistema. El procesador PowerPC incluye una interfaz nativa para el bus PLB y por ello la fig. 4.2 muestra la selección del bus PLB para la plataforma hardware del SoPC.

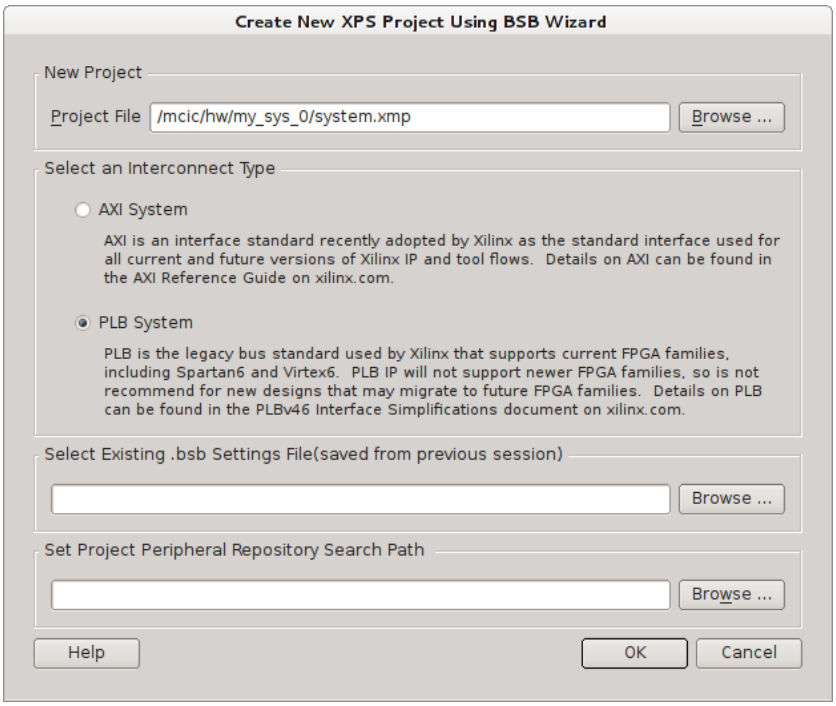


Figura 4.2: Configuración de la plataforma hardware.

Posterior a ello se procede a la creación de un nuevo diseño tras lo cual aparecerá una ventana como la mostrada en la fig. 4.3 donde se muestra la se-

lección de la tarjeta ML507. En seguida, el BSB de Xilinx permite configurar el número de procesadores en la plataforma y debido a que el Virtex 5 FX70 sólo incluye un procesador PowerPC 440, se ha seleccionado la arquitectura mono-procesador tal y como se muestra en la fig. 4.4.

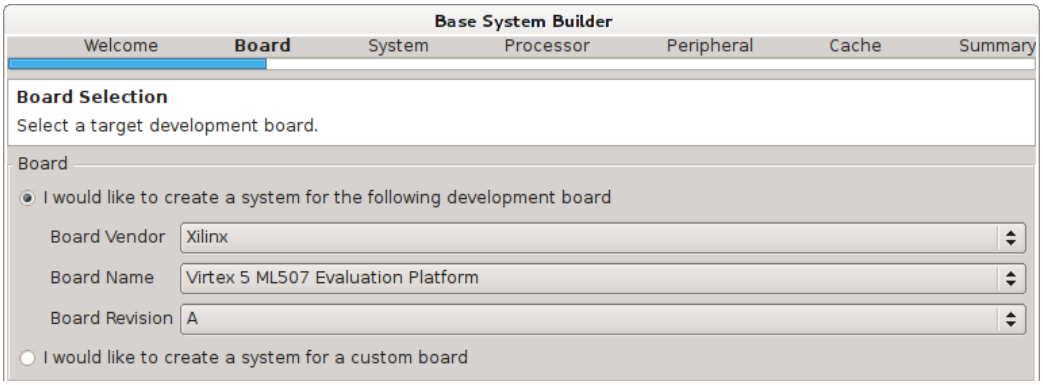


Figura 4.3: Selección del modelo de la tarjeta de desarrollo.

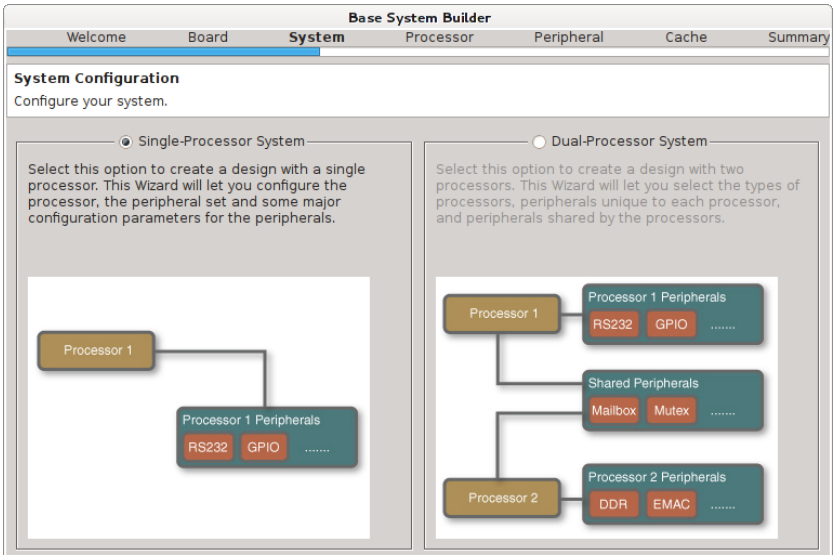


Figura 4.4: Selección de la arquitectura del procesador.

La herramienta BSB permite establecer la frecuencia de operación del procesador y del bus del sistema de manera independiente, sin embargo existen restricciones de diseño que deben tomarse en cuenta, por ejemplo, el

bus del sistema no puede operar a una frecuencia menor de la que opera el procesador.

Debido a las restricciones de diseño del IPCore `plb_v46` la frecuencia máxima de operación es de 357.5 MHz de acuerdo a lo reportado en [52]. Para este caso particular la plataforma hardware se configura a `125 MHz` tal y como lo muestra la fig. 4.5.

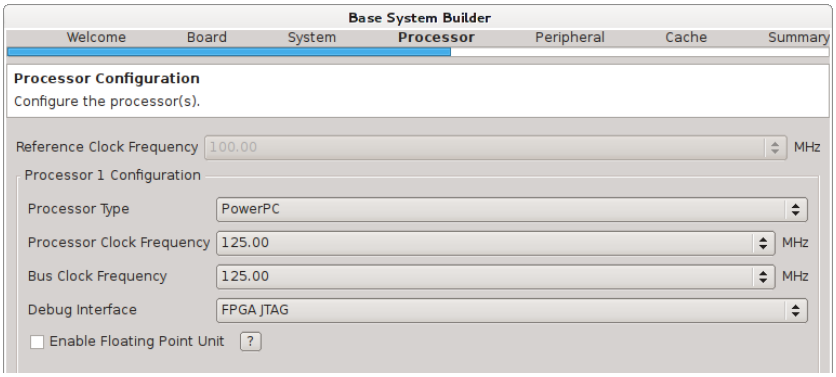


Figura 4.5: Parámetros de configuración del procesador.

El siguiente paso describe la configuración de los periféricos del sistema. Debido a los requerimientos de Linux se ha incorporado un *timer* y un *watch_ dog_ timer*. Los detalles de configuración de cada periférico se muestran en la fig. 4.6.

Con la finalidad de acelerar la ejecución de procesos se decidió habilitar la memoria *caché* para almacenar datos de la memoria RAM del sistema. La arquitectura del procesador sólo permite integrar un área de memoria *caché* de 32 KB. La fig. 4.7 muestra los detalles de esta configuración.

Al finalizar el proceso de configuración del BSB, el asistente presenta una pantalla con los detalles de la configuración del sistema tal y como se muestra en la fig. 4.8.

Posterior al proceso de configuración e integración de la plataforma base, se deberá lanzar el proceso de síntesis de acuerdo a lo establecido en [48, 49, 50, 51, 53]. Para ello se debe ejecutar la opción `Generate Bitstream` dentro del menú `Hardware`.

El proceso de síntesis termina con la generación del archivo `system.bit`. Este archivo contiene la configuración de los elementos internos del FPGA que implementan la funcionalidad requerida por el SoPC. Sin embargo, este

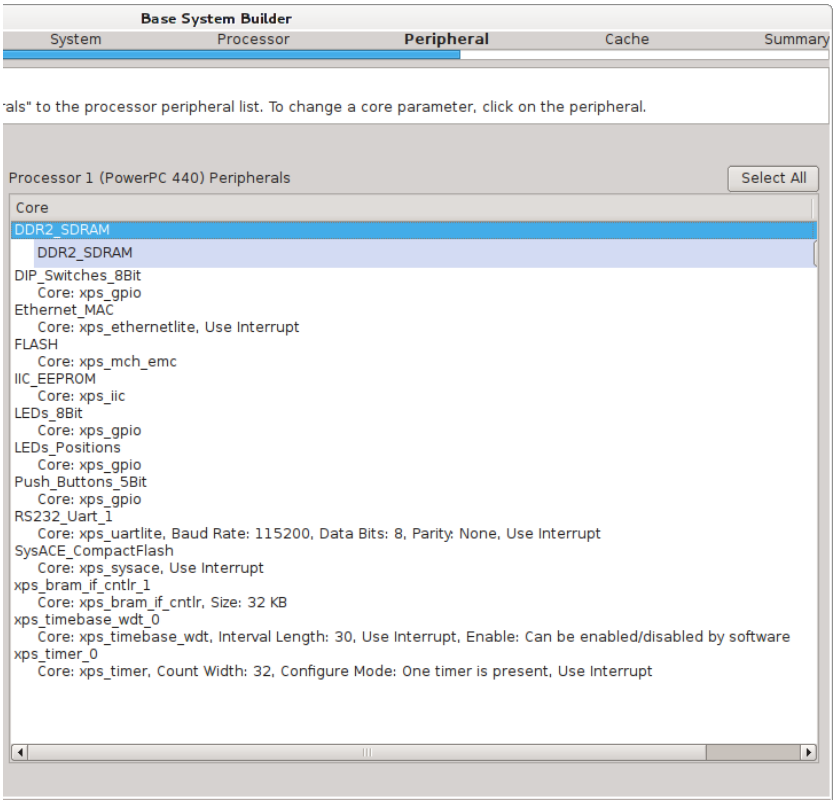


Figura 4.6: Periféricos del sistema.

archivo de configuración aún no tiene incrustado el código de ejecución inicial por lo que se deberá utilizar la herramienta *SDK* para la creación de un proyecto software que genere el archivo *ELF* que se utilizará para inicializar la memoria interna del FPGA.

Si bien el proceso de integración de una plataforma puede realizarse de múltiples maneras, el asistente tiene la ventaja de ayudar al diseñador hardware en la interconexión de módulos, la generación de restricciones de localización, asignación de señales de sincronización, estructuración de la jerarquía de compilación del BSP[51], entre otras funciones.

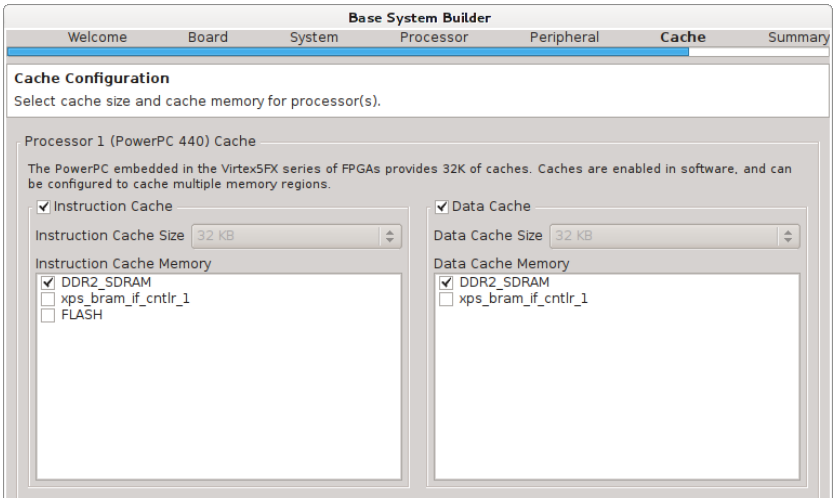


Figura 4.7: Habilitación de la *caché* del procesador.

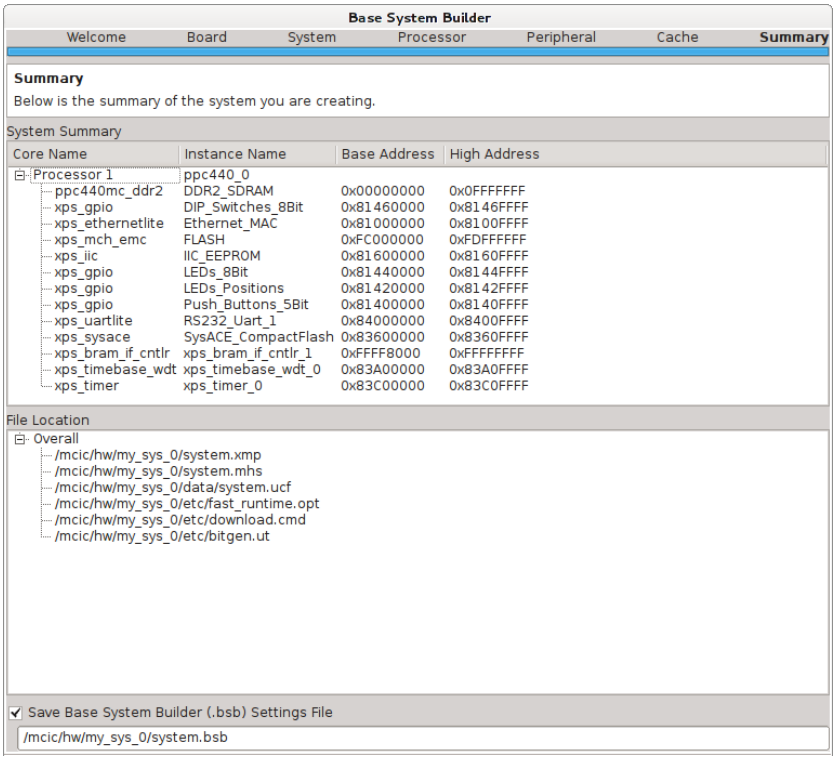


Figura 4.8: Resumen de la implementación de la plataforma hardware.

4.1.1. Creación de un proyecto en SDK

La herramienta de Xilinx, *XPS* incluye una rutina software que permite lanzar el entorno de desarrollo *SDK*. Este procedimiento se encuentra en la opción `Export Hardware Design to SDK` dentro del menú `Project`. La interfaz gráfica de *SDK* pedirá al usuario asignar una carpeta de trabajo donde se crearán las carpetas para cada proyecto software. En este caso se determinó colocar dicha carpeta dentro del directorio de trabajo de sistema generado en XPS, esto es en `/mcic/hw/my_sys_0/SDK/workspace`.

Las figs. 4.9, 4.10, 4.11, 4.12a y 4.12b muestran el procedimiento para la generación de un proyecto software.

En esta etapa la plataforma hardware no cuenta con sistema operativo por tanto las pruebas se realizan en modo *stand–alone*.

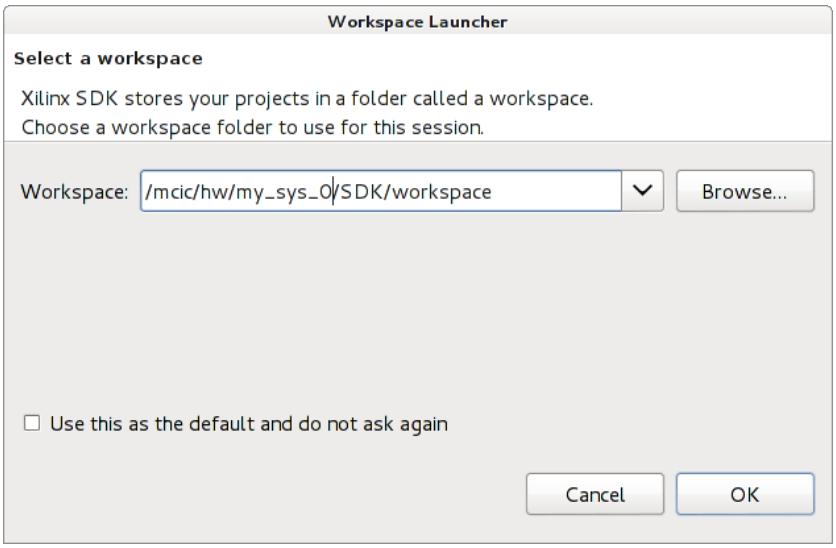


Figura 4.9: Directorio de trabajo de SDK.

Al finalizar este procedimiento el entorno de desarrollo automáticamente lanza el proceso de compilación para los proyectos dentro del directorio de trabajo activo. El resultado de este proceso es la generación de un archivo *ELF* que será utilizado para probar la funcionalidad de la tarjeta. Los detalles de las pruebas hardware sobre la plataforma se encuentran en 5.1.

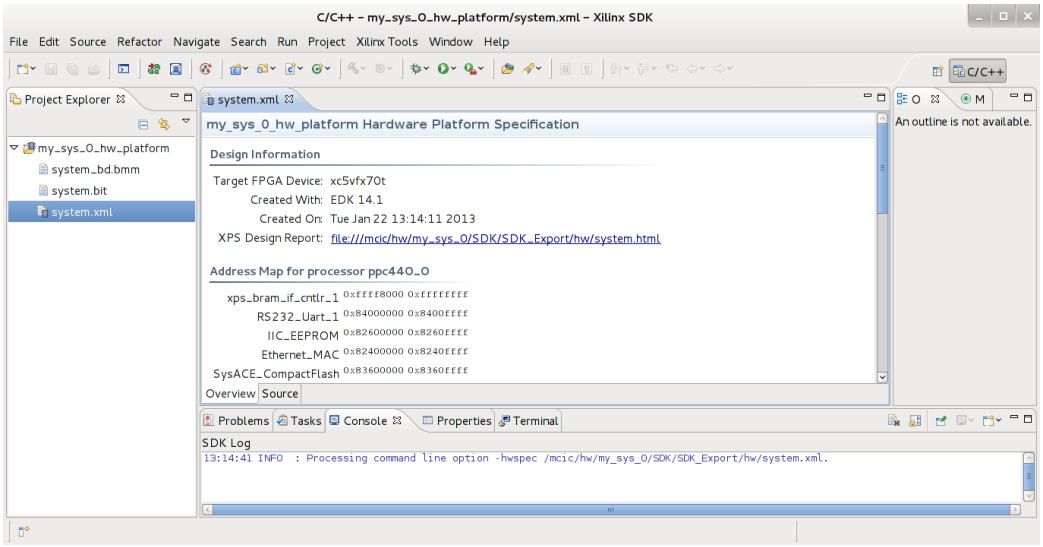


Figura 4.10: Resultado de la configuración de los parámetros de la plataforma hardware en el entorno de desarrollo de SDK.

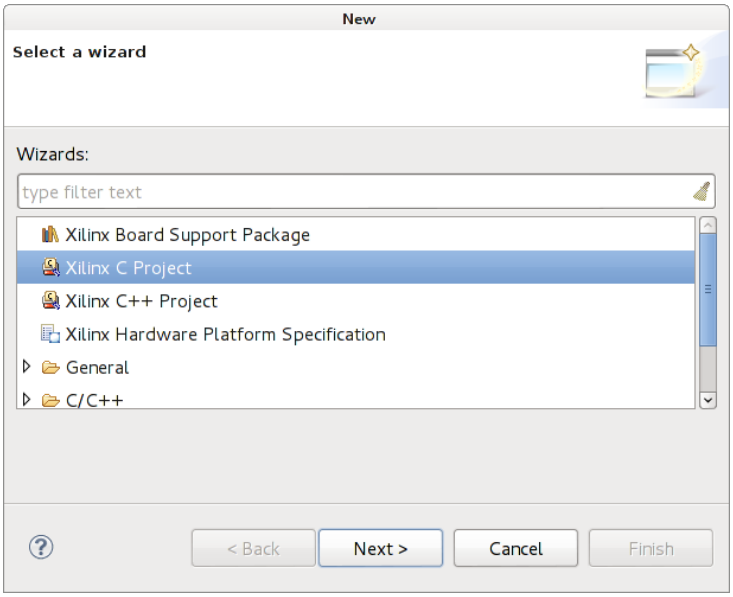
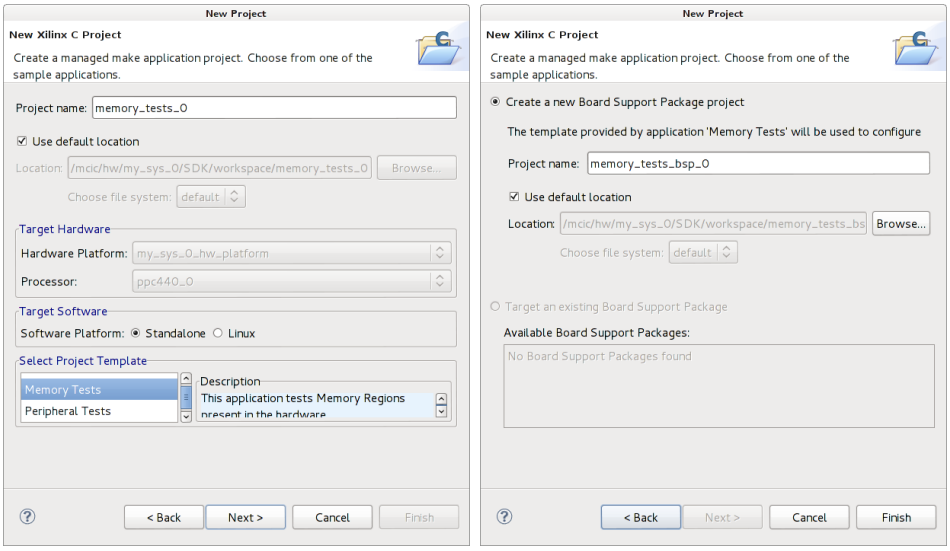


Figura 4.11: Selección del proyecto software.



(a) Proyecto base. (b) BSP del proyecto base.

Figura 4.12: Detalles del proyecto software.

4.2. Generación de funciones hardware

La generación de funciones hardware se implementa usando el modelo de un dispositivo periférico a la arquitectura de la plataforma. Esto significa que la función generada se conecta al bus PLB del sistema y está sujeta a las políticas de intercomunicación que el árbitro de bus implemente.

4.2.1. Xilinx CIP Wizard

Xilinx proporciona un asistente para crear o importar funciones hardware llamado CIP (*Create or Import Peripheral*) Wizard. Esta herramienta colabora en el diseño de un módulo IPCore para simplificar el proceso de integración del elemento funcional a la arquitectura del sistema.

El código generado por el asistente *CIP* implementa un periférico sin puertos de entrada o salida. La única función implementada es la captura de datos. Es decir, cuando recibe una petición de escritura, el módulo hardware captura el dato recibido y cuando se genera una petición de lectura, envía el dato escrito previamente. Este esqueleto sirve al diseñador hardware para implantar la funcionalidad requerida y acoplar el módulo hardware al bus del sistema.

Xilinx ha propuesto un esquema jerárquico de búsqueda para que el entorno de desarrollo de *EDK* pueda localizar los módulos IPCore generados por los usuarios. Se sugiere la creación de un repositorio en donde se almacenarán la descripción funcional de los módulos hardware. Dicho repositorio deberá seguir la estructura de directorios descrita en [54] y mostrada en las figuras 4.13, 4.14 y 4.15.

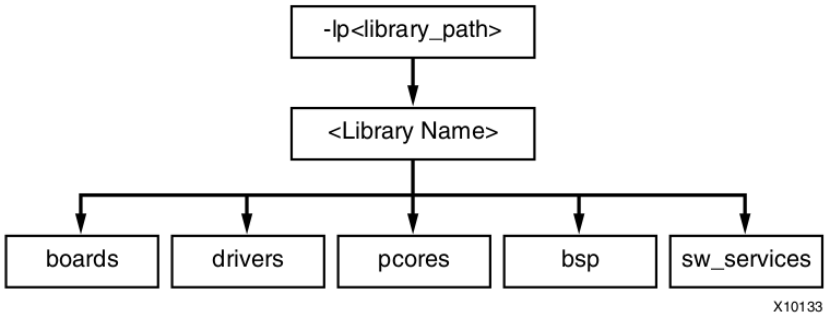


Figura 4.13: Estructura del repositorio de EDK.

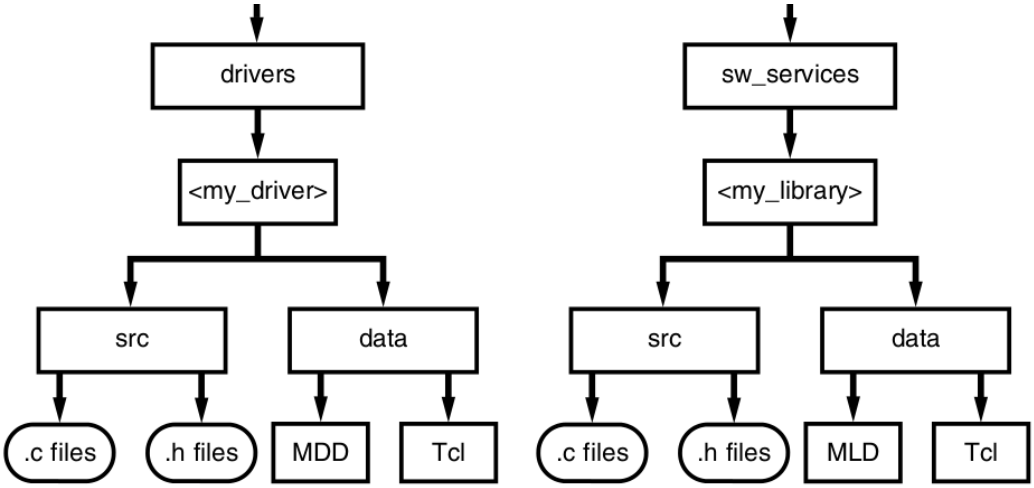


Figura 4.14: Detalle de *drivers* y *sw_ services*.

Las figs. 4.16, 4.17, 4.18, 4.19, 4.20, 4.21 y 4.22 muestran los detalles del proceso de generación de un módulo IPCore llamado `mygpio`. Este módulo es utilizado en las pruebas realizadas en 5.2.1.

Posterior a la generación del IPCore `mygpio` se editaron los archivos `mygpio.vhd` y `user_logic.vhd` los cuales se encuentran en la carpeta del IPCore. Estos archivos implementan la funcionalidad de un dispositivo genérico de entrada y salida. La descripción comportamental incluye la captura de datos de los `DipSw` y la escritura de datos hacia los `Leds8`. Por tanto se eliminaron de la plataforma original los IPCore que BSB había asignado a estos dispositivos externos para reasignar las entradas y salidas al nuevo dispositivo.

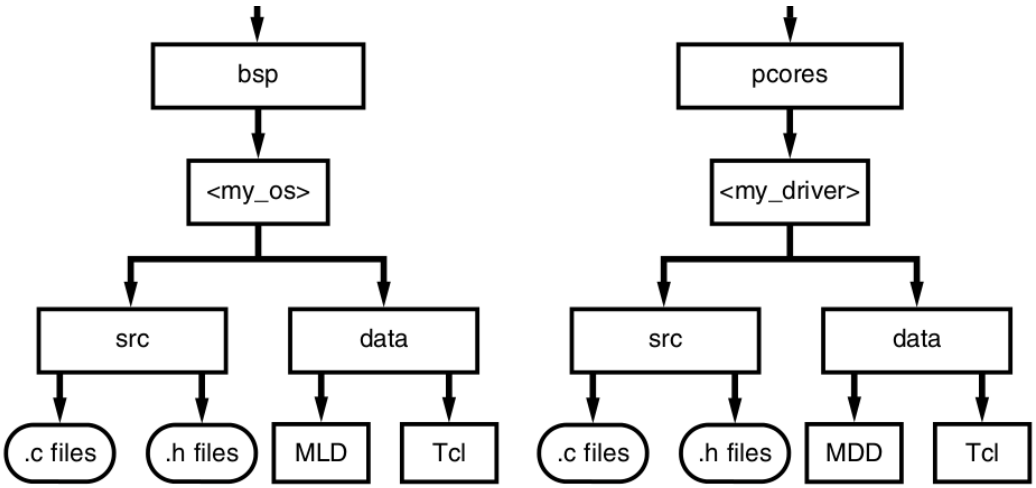


Figura 4.15: Detalle de *bsp* y *pcores*.

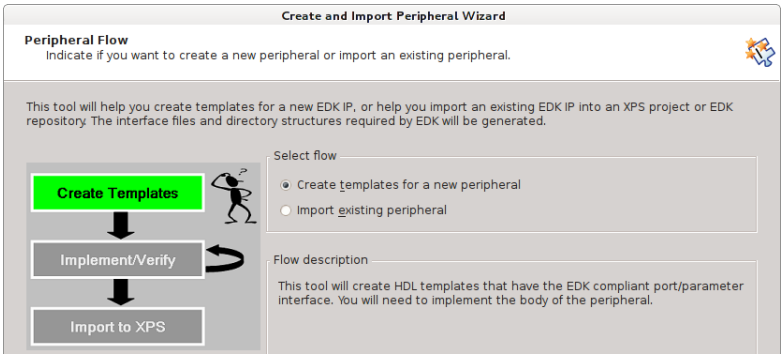


Figura 4.16: Creación del módulo hardware.

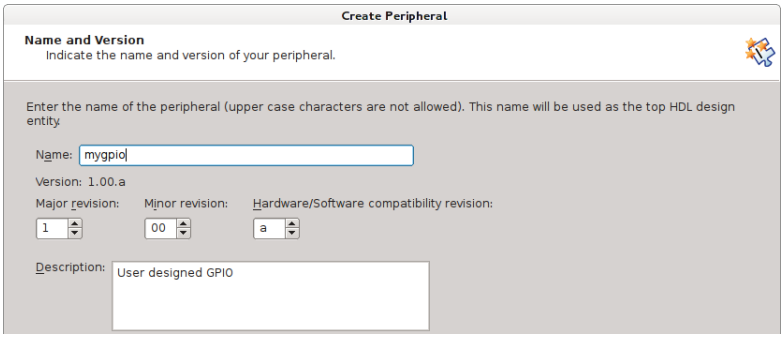


Figura 4.17: Identificación y número de versión del IPCore.

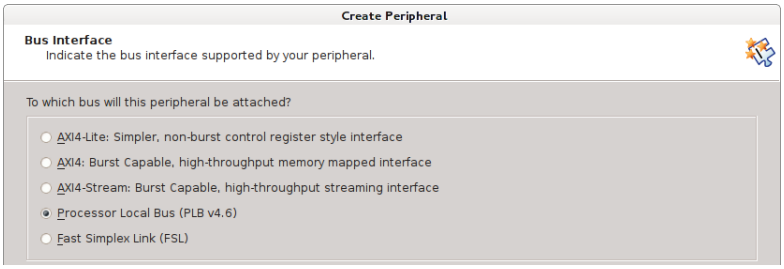


Figura 4.18: Selección del tipo de bus.

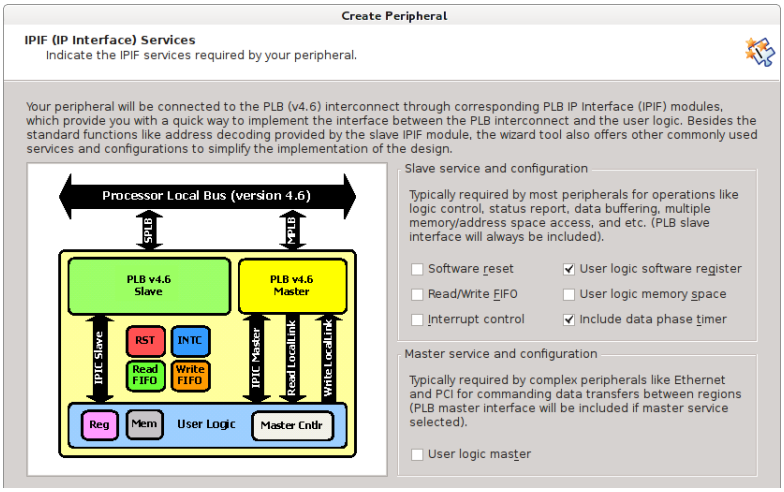


Figura 4.19: Configuración del módulo IPIF.

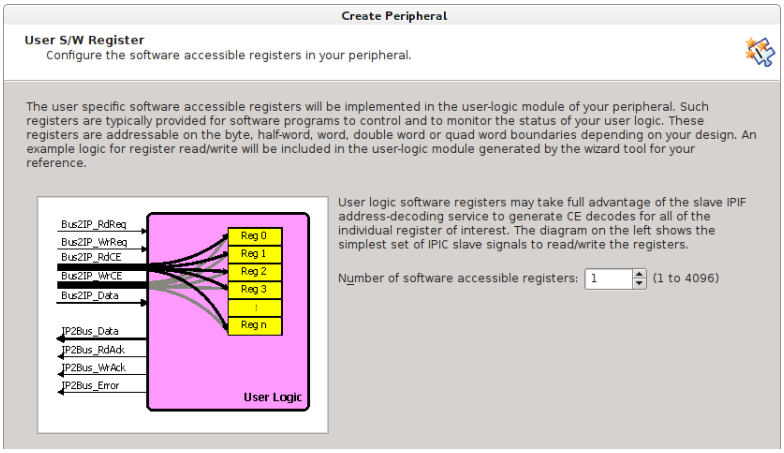


Figura 4.20: Número de registros.

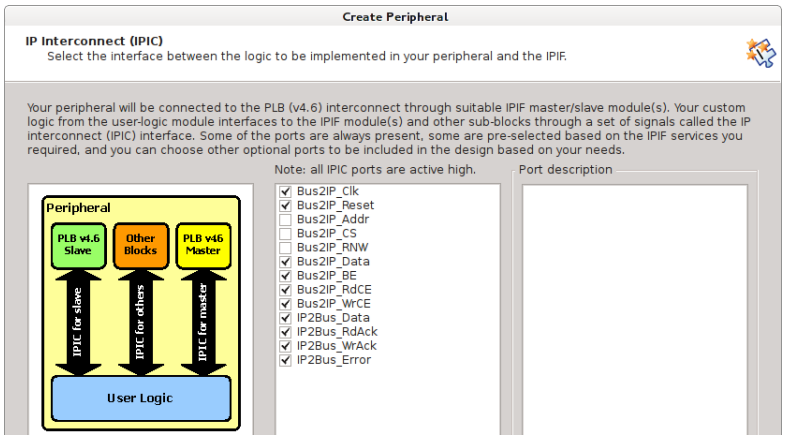


Figura 4.21: Detalles de implementación del IPIC.

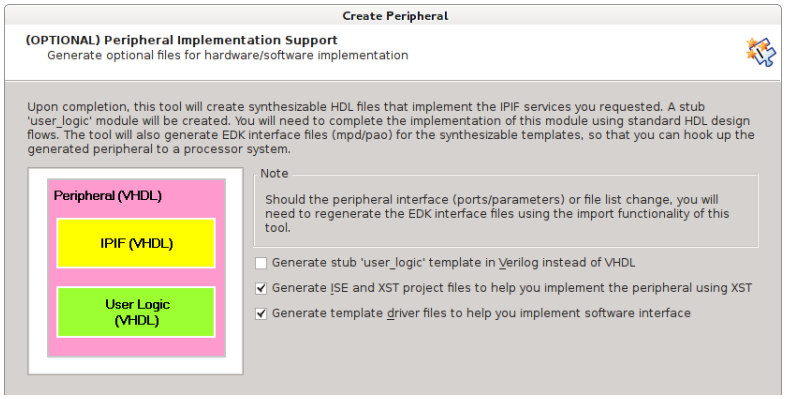


Figura 4.22: Inclusión del proyecto de desarrollo en ISE.

4.2.2. Xilinx Core Generator

Core Generator permite a los diseñadores hardware incluir módulos funcionales pre-compilados. Ello permite disminuir el tiempo total de diseño de un sistema. La biblioteca de funciones de Xilinx incluye entre otros elementos:

- Memorias y FIFOS.
- Operaciones aritméticas con enteros.
- Operaciones aritméticas de punto flotante.
- Controlador de ChipScope™ Pro.
- Analizador Lógico Integrado.
- Asistente de sincronización (*Clocking Wizard*).
- MIG (*Memory Interface Generator*).
- MGTs (*RocketIO™ Multi-Gigabit Transceivers*).
- Buses PCI, PCI-X, SPI, CAN, RapidIO, PCI-Express.
- Filtros Fir, DDS, FFT.
- Decodificadores Viterbi, Reed-Solomon.
- Comunicación Ethernet, Wireless, 3GPP, OBSAI.

Sin embargo, Core Generator sólo genera una caja negra con la funcionalidad deseada; para poder integrarla a la plataforma hardware del SoPC es necesario usar el *CIP Wizard* para crear la plantilla de interconexión e instanciar a través de `user_logic.vhd` la funcionalidad del módulo creado por Core Generator.

El procedimiento mostrado a continuación genera el IPCore `cgadder` para un sumador de números enteros usando la biblioteca de funciones hardware de Xilinx llamada `XilinxCoreLib` donde se encuentra la función hardware (*core*) `c_addsub_v11_0`. Los parámetros de configuración más importantes del *core* `c_addsub_v11_0` usados en el mapeo genérico (`generic map ();`) son mostrados en el listado 4.1.

Listado 4.1: Configuración de CGADDER.

```
1  c_a_type => 1,
2  c_a_width => 32,
3  c_add_mode => 0,
4  c_has_c_out => 1,
5  c_has_ce => 0,
6  c_has_sclr => 0,
7  c_has_sinit => 0,
8  c_has_sset => 0,
9  c_implementation => 0,
10 c_latency => 0,
11 c_out_width => 32,
12 c_sinit_val => "0",
13 c_xdevicefamily => "virtex5"
```

Además del procedimiento descrito en las figs. 4.16, 4.17, 4.18, 4.19, 4.20, 4.21 y 4.22, será necesario agregar el *netlist* al proceso de importación del periférico debido a que estos archivos serán agregados al archivo `cgadder_v2_1_0.bbd` (*Black Box Definition*).

El procedimiento para agregar un módulo hardware que integra una función de la biblioteca de Xilinx a la arquitectura del SoPC se describe a continuación.

- Crear esqueleto de periférico usado *CIP Wizard*.
- Usar *ISE* para modificar el proyecto generado por *CIP Wizard*.
- Agregar la función de biblioteca de Xilinx a través de Core Generator.
- Instanciar el *core* en `user_logic.vhd`.
- Sintetizar el módulo IPCore.
- Importar el *core* usando *CIP Wizard* y añadir el *netlist* del *core*.
- Agregar el IPCore a la arquitectura del SoPC.
- Configurar el IPCore (BUS, C_BASE_ADDR, etc).
- Sintetizar la plataforma hardware.

La fig. 4.23 muestra la selección que deberá hacerse al especificar los elementos a importar del IPCore; la fig. 4.24 especifica los archivos `*.mpd` (*Microprocessor Peripheral Definition*) y `*.prj` que describen la funcionalidad del IPCore y las reglas de síntesis que se deberán seguir para la generación del módulo hardware.

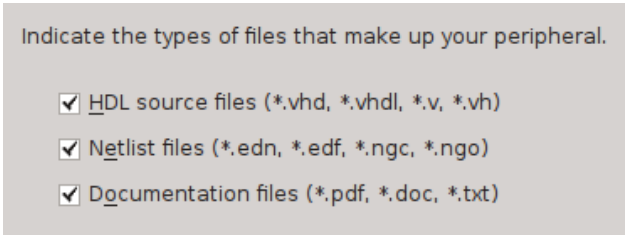


Figura 4.23: Especificación de tipos de archivos fuente para `cgadder`.

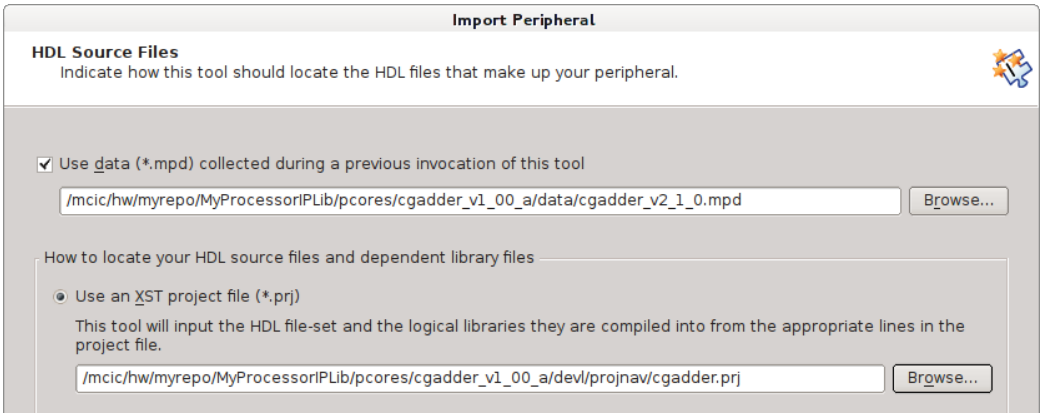


Figura 4.24: Especificación del proyecto de desarrollo.

La fig. 4.25 muestra la selección de bibliotecas lógicas que se hizo para el módulo en cuestión; la fig. 4.26 especifica el *netlist* de la función hardware.

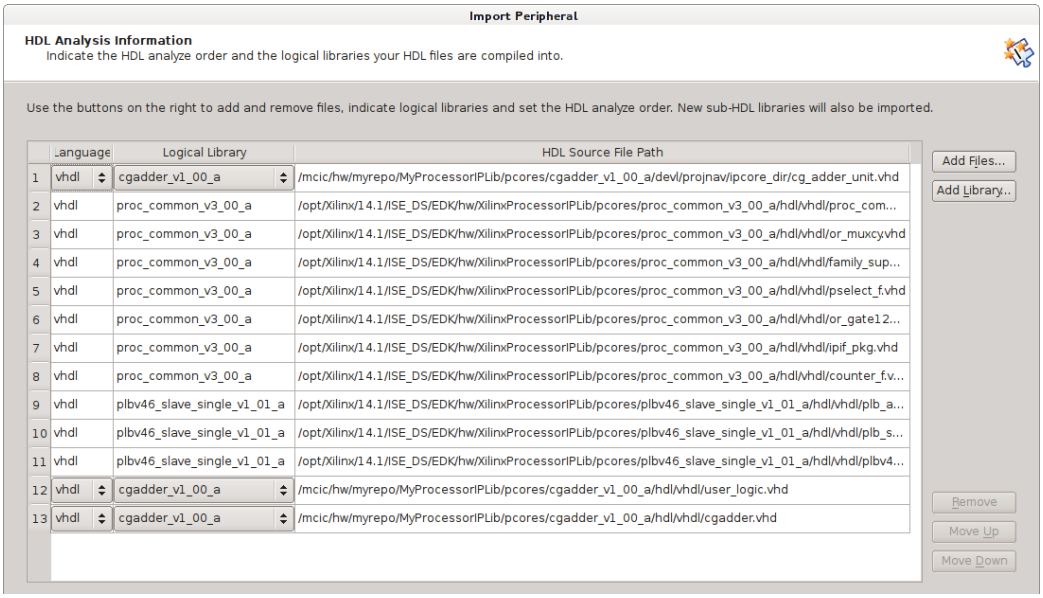


Figura 4.25: Selección de bibliotecas lógicas.

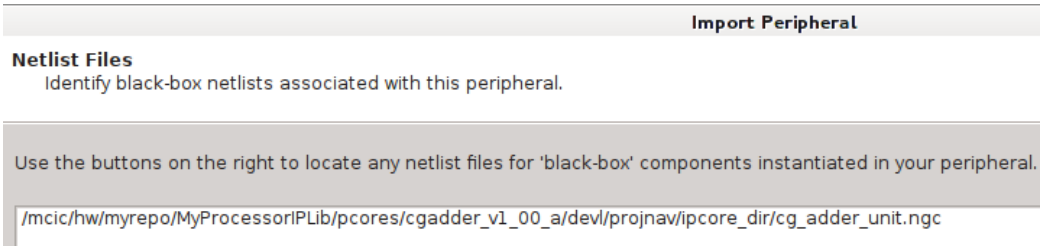


Figura 4.26: *Netlist* del *core* `cgadder`.

4.3. Parametrización de la arquitectura

Gibson y Herrenschmidt[17] propusieron un modelo de parametrización jerárquica para la descripción de una arquitectura hardware llamado *device tree*.

Las propiedades más importantes de la parametrización de la arquitectura obtenidos a partir del modelo propuesto en [17] permiten re-localizar elementos dentro de la descripción lo que permite al cargador de arranque o al kernel mover su estructura sin tener que hacer cambios a la descripción interna debidos a la relocalización. Esto permite añadir o eliminar nodos o ramas a la descripción de la arquitectura, propiedad que será útil para agregar funciones hardware a la plataforma hardware.

La descripción paramétrica binaria resultante de compilar la descripción textual de la plataforma hardware tiene un tamaño reducido lo que permite su aplicación en los sistemas embebidos y es llamada *device tree blob*.

La representación textual del *device tree* permite la interpretación, modificación y verificación de la estructura del sistema y para ello sólo es necesario manipular el texto descriptivo asociado al elemento de interés para obtener el resultado deseado. Sin embargo se deben seguir las reglas propuestas por Gibson y Herrenschmidt. La figura 4.27 muestra una representación gráfica de la estructura de una plataforma hardware y la fig. 4.28 muestra la estructura binaria resultante de la compilación del *device tree* en el primer bloque de datos.

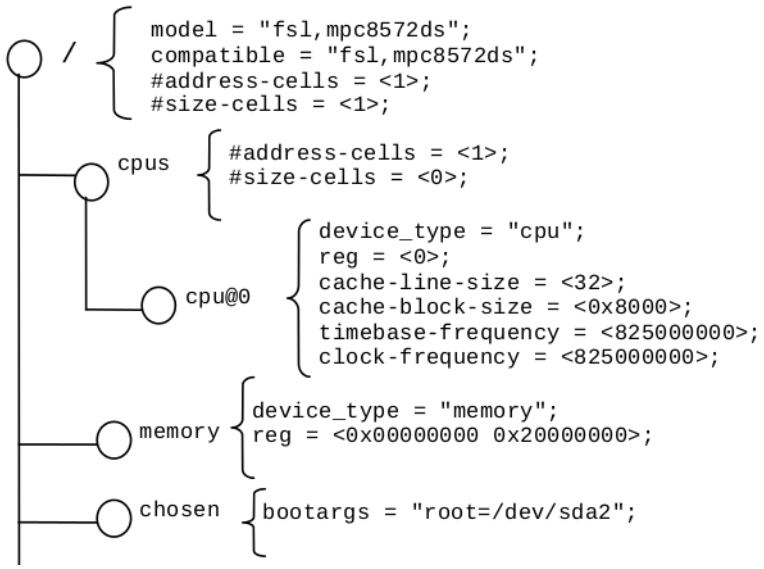


Figura 4.27: Representación de elementos dentro del *device tree*.

Listado 4.2: Fragmento de un archivo ejemplo *device tree*.

```
1  /dts-v1/;
2  / {
3  model = "fsl,MPC8572DS";
4  compatible = "fsl,MPC8572DS";
5  #address-cells = <1>;
6  #size-cells = <1>;
7
8  cpus {
9  #address-cells = <1>;
10 #size-cells = <0>;
11 cpu@0 {
12 device_type = "cpu";
13 reg = <0>;
14 d-cache-line-size = <32>; // 32 bytes
15 i-cache-line-size = <32>; // 32 bytes
16 d-cache-size = <0x8000>;  // L1, 32K
17 i-cache-size = <0x8000>;  // L1, 32K
18 ...
19 };
```

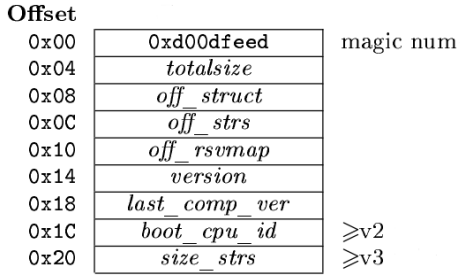


Figura 4.28: Primer bloque de la estructura binaria del archivo *device tree*.

Tabla 4.1: Parámetros de configuración del *device tree*.

Parámetro	Valor
bootargs	console=ttyUL0 root=/dev/xsa2 rw rootfstype=ext3 ip=off
console device	RS232_Uart_1

Xilinx ha promovido el desarrollo de un BSP (*Board Support Packages*) que se integra a *SDK* para generar la descripción textual que parametriza la arquitectura hardware de un SoPC. Este archivo de texto será de vital importancia para el correcto funcionamiento del cargador de arranque y del núcleo del sistema operativo.

Xilinx mantiene un repositorio *git* para el BSP del *device tree* que se integra en SDK. La copia del repositorio se creó en la carpeta de trabajo `/mcic/git`. El repositorio se clonó del servidor: `git://git.xilinx.com/device-tree.git` y se colocó en la carpeta del repositorio `myrepo` que se creó para alojar los IPCores. En el entorno de trabajo de SDK, agregar el nuevo repositorio a través de la opción `Repositories` del menú `Xilinx Tools`.

La figura 4.29 muestra la configuración del nuevo repositorio en SDK.

Para generar la descripción paramétrica de la plataforma hardware del SoPC habrá que agregar un nuevo proyecto software en SDK pero ahora se deberá seleccionar el BSP que genera el *device tree* como se muestra en la fig. 4.30. Al finalizar el proceso de creación del BSP aparecerá una ventana donde se deberá especificar los parámetros de configuración `bootargs` y `console device` con los parámetros mostrados en la tabla 4.1.

SDK automáticamente lanzará el proceso de compilación del BSP donde

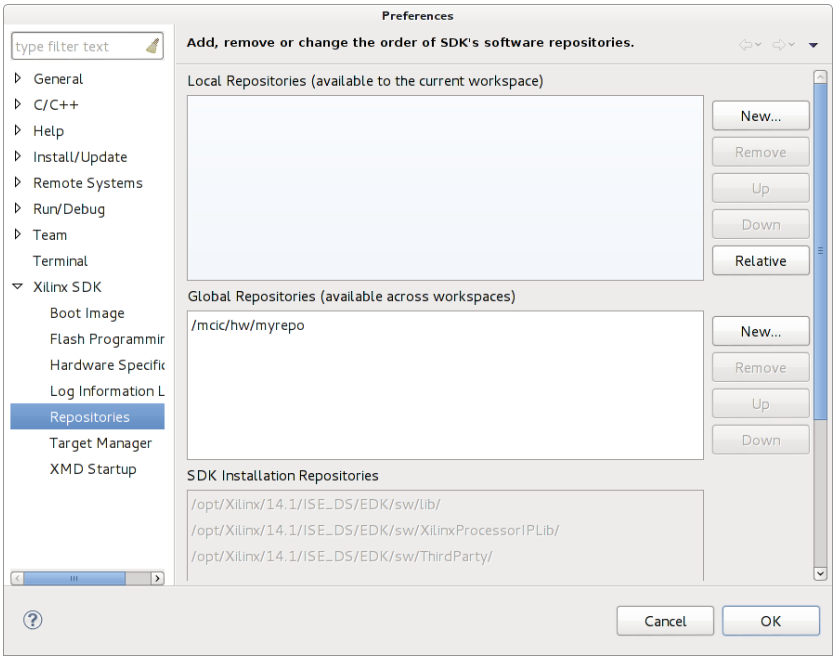


Figura 4.29: Repositorio `myrepo` para generar el *device tree* en SDK.

se generará el archivo `xilinx.dts` el cuál contiene la descripción textual de la arquitectura del SoPC.

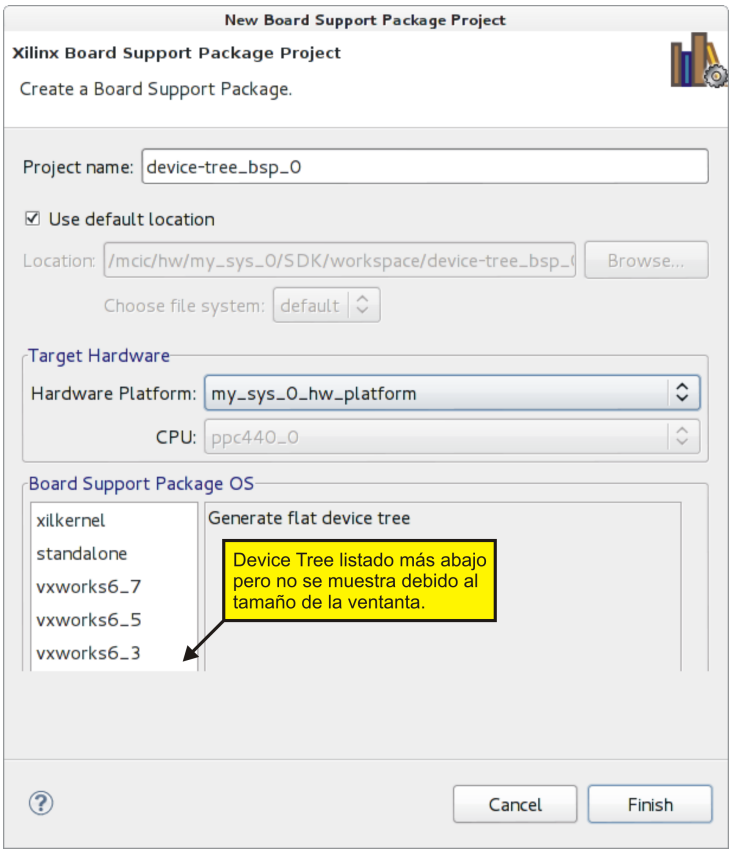


Figura 4.30: Generación del *device tree* en SDK.

4.4. Generación del Sistema Operativo

La generación del sistema operativo se realiza a través de las herramientas que proporciona el subproyecto *Yocto*[27] bajo el cual se integra el proyecto Poky[39] a través del cual se genera el sistema operativo.

Poky provee un entorno de desarrollo capaz de generar código para arquitecturas ARM, MIPS, PowerPC y x86. Además de ello, provee la descripción de los procesos de compilación para generar código objeto para las arquitecturas soportadas. Utiliza diversas herramientas que colaboran en la administración de repositorios, descarga, sincronización, configuración, aplicación de parches, compilación, encapsulación (paquetes `.rpm`, `ipk` o `.deb`) y generación de imágenes para las arquitecturas objetivo.

Hasta Junio de 2012 Poky tenía 27 ramas de desarrollo y cada rama tiene soporte para diversas arquitecturas, procesadores, tarjetas de desarrollo y además incluye la capacidad de definir nuevas arquitecturas a través de la especificación de BSPs (*Board Support Packages*). Debido a ello fue necesario configurar un repositorio local donde se mantuviera control sobre los procesos de actualización del sistema de desarrollo.

Este proyecto continua el desarrollo de A. Alonso[24], quien generó la descripción en poky para compilar un sistema operativo para la tarjeta de desarrollo ML507. La arquitectura originalmente propuesta incluye IPCores para interfaz con un monitor VGA, dispositivo de entrada (`ps2`) además de los IPCores que se utilizaron en el proyecto `my_sys_0`. El sistema operativo generado permite el uso del entorno gráfico para la visualización de contenido. La especificación para la compilación del kernel, el cargador de arranque y el archivo de configuración del FPGA. El repositorio generado por el proyecto "*Aligator OS: An ambedded operating system*" se encuentran en el servidor: `git://git.yoctoproject.org/meta-xilinx` .

La generación del sistema operativo utiliza el repositorio `meta-xilinx` el cual se clona y coloca dentro de la carpeta de trabajo de Poky a fin mantener la estructura propuesta en el proyecto y se basa en la información provista en [28, 29, 39].

A partir de los repositorios de Poky y el repositorio `meta-xilinx` se genera un sistema operativo básico. Poky tiene un archivo de configuración global a través del cual se especifican los parámetros que determinan las tareas a realizar para generar el sistema operativo.

La tabla 4.2 describe la función de cada parámetro y las tablas 4.3, 4.4 y 4.5 especifican los valores que deberán asignarse a las variables en cuestión

para la generación del sistema operativo y las utilerías entorno al núcleo de Linux `2.6.37` que en conjunto forman la distribución de Linux llamada `Poky-Linux`.

El entorno de desarrollo de Poky es un conjunto de subsistemas que está formado por repositorios locales y remotos de proyectos software, archivos de descripción de procesos de compilación, archivos descriptivos de arquitecturas hardware (BSP), políticas y parámetros de configuración como los mostrados en las tablas 4.3, 4.4 y 4.5. La figura 4.31 muestra el esquema de funcionamiento de Poky.

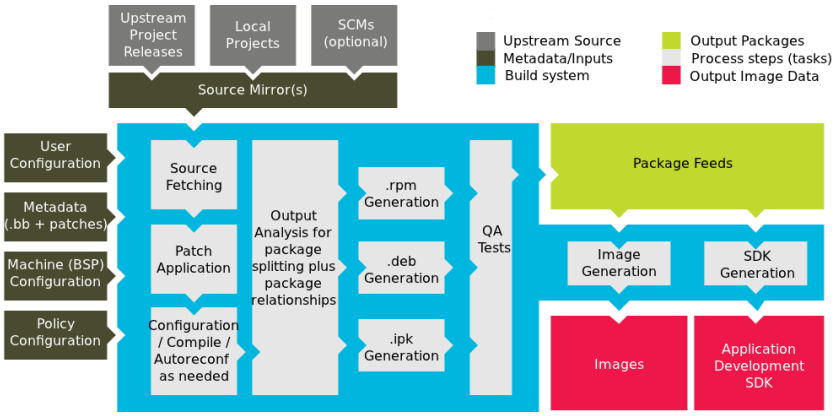


Figura 4.31: Esquema de operación de POKY.

4.4.1. Recetas de bitbake

`Bitbake` es una utilidad de Poky a través de la cual se especifica la receta a procesar. Una receta en `bitbake` es un archivo que contiene especificaciones para el proceso de compilación de una utilidad. Estas especificación incluye generalmente una descripción, la especificación del tipo de licencia (GPL, GPLv2, MIT, BSD, Apache, GFDL, LGPL, etc.), la versión del paquete, la versión de liberación, los archivos fuente de la utilidad a compilar, opcionalmente los parches que se aplicarán y las funciones que se ejecutarán durante el proceso de generación de la utilidad.

De manera general `bitbake` ejecuta una secuencia de comandos mediante los cuales se realizan las tareas de configuración y compilación del paquete o utilidad descrita en la receta invocada. Los comandos ejecutados son: sets-

Tabla 4.2: Parámetro de configuración en el archivo `local.conf` en Poky.

Descripción del parámetro de configuración.
`BB_NUMBER_THREADS` : Especifica el número máximo de hilos en ejecución.
`PARALLEL_MAKE` : Especifica el número máximo de procesos en ejecución paralela.
`DL_DIR` : Especifica el directorio donde se guardarán los paquetes descargados.
`SSTATE_DIR` : *Shared State*, especifica el estado en el que se encuentran los procedimientos ejecutados en Poky.
`MACHINE` : Arquitectura objetivo para el kernel.
`XILINX_BSP_PATH` : Ruta del Base Support Packages.
`XILINX_BOARD` : Define las características hardware de la plataforma sobre la que se ejecutará el kernel; incluye restricciones de diseño relativos a dispositivos de entrada/salida, funciones de la MMU, Controladores de dispositivos, etc.
`XILINX_LOC` : Define la ubicación de las herramientas de configuración, síntesis y compilación de Xilinx las cuales se utilizarán para generar una imagen en formato ACE para la configuración del FPGA.
`GCCVERSION` : Especifica la versión del compilador.
`SDKGCCVERSION` : Especifica la versión de entorno de desarrollo de gcc.
`BINUVERSION` : Especifica la versión de las utilidades binarias de la plataforma hardware.
`EGLIBCVERSION` : Especifica la versión del paquete EGLibC que es una biblioteca de funciones usada por gcc.
`UCLIBCVERSION` : Especifica la versión del paquete UCLibC, que es similar a EGLibC.
`LINUXLIBCVERSION` : Especifica la versión de bibliotecas de sistema operativo a generar.
`PREFERRED_VERSION_linux-yocto_${MACHINE}` : Especifica versión del kernel.

Tabla 4.3: Parámetros de generales de configuración en Poky.

Parámetro	Valor
BB_NUMBER_THREADS	"4"
PARALLEL_MAKE	"-j 4"
DL_DIR	"/mcic/downloads"
SSTATE_DIR	"/mcic/sstate-cache"

Tabla 4.4: Parámetros de configuración hardware de Poky.

Parámetro	Valor
MACHINE	"virtex5"
XILINX_BSP_PATH	"/mcic/hw/my_sys_0"
XILINX_BOARD	"ML507"
XILINX_LOC	"/opt/Xilinx/14.1/ISE_DS"

Tabla 4.5: Parámetros de selección específica de paquetes en Poky.

Parámetro	Valor
GCCVERSION	"4.5.1"
SDKGCCVERSION	"4.5.1"
BINUVERSION	"2.21"
EGLIBCVERSION	"2.12"
UCLIBCVERSION	"0.9.30.1"
LINUXLIBCVERSION	"2.6.37.2"
PREFERRED_VERSION_linux-yocto_${MACHINE}	"2.6.37%"

cene, fetch, unpack, patch, configure, populate_lic, compile, install, populate_sysroot, package y package_write_rpm.

El listado 4.3 contiene los elementos más importantes de la receta `app_git.bb` que se utilizará en el desarrollo de aplicaciones para el SoPC.

Listado 4.3: Receta ejemplo de `bitbake`.

```
1 DESCRIPTION = "A simple user application"
2 LICENSE = "GPLv2"
3 LIC_FILES_CHKSUM =
      "file://${POKYBASE}/meta/files/common-licenses/GPLv2;
      md5=751419260aa954499f7abaabaa882bbe"
4 PR = "r0"
5 PV = "0.1"
6 BRANCH = "master"
7 SRCREV = "${AUTOREV}"
8 SRC_URI =
      "git:///mcic/git/app;protocol=file;branch=${BRANCH}"
9 S = "${WORKDIR}/git"
```

4.4.2. El sistema de archivos

La tarjeta de desarrollo ML507 utiliza el subsistema de configuración *System ACE* para leer datos de una memoria *Compact Flash* (CF) y configurar el FPGA. Además de ello, este subsistema permite al SoPC acceder un sistema de almacenamiento permanente. Sin embargo para que *System ACE* pueda interpretar las estructuras de datos que describen el sistema de archivos en la *CF* la primera partición deberá ser del tipo `FAT16` y tener un único sector reservado. Las utilidades que dan formato a las particiones, reservan 2 o hasta 32 sectores dependiendo de las características y el tamaño de la partición; de manera predeterminada se reservan 2 para `FAT16` y 32 para sistemas `FAT32`. Por ello se deberá dar formato a la partición siguiendo esta especificación. En los sistemas Linux esto se puede hacer con el comando: `mkdosfs -av -R1 -F16 -n config /dev/sdd1`

La descripción de los parámetros del formato se presenta a continuación.

-a Alínea las estructuras de datos al tamaño del *cluster*.

-v Ejecución detallada.

-R Selecciona el número de sectores reservados.

-F Especifica el tipo de sistema de archivos a usar (12, 16 o 32 bit).

-n Establece el nombre del volumen para el sistema de archivos.

/dev/sdd1 especifica la partición de la CF.

A continuación se muestra un fragmento del resultado del particionamiento de una CF de 4GB para el SoPC obtenido al ejecutar `fdisk -l /dev/sdd`

```
   Device Boot      Start         End      Blocks   Id  System
/dev/sdd1            2048      116735       57344    6  FAT16
/dev/sdd2          116736     7244000     3563632+  83  Linux
/dev/sdd3         7244001     7372511       64255+  82  Linux swap
```

4.5. Aplicación de usuario

Las aplicaciones de usuario se desarrollan de manera habitual a través de la codificación en `C` de la funcionalidad de programa. Sin embargo para integrar los programas de usuario al sistema de desarrollo en Poky se debe crear una receta que describa la forma de compilar el paquete.

A continuación se describen los procesos necesarios para generar la aplicación `hello-mcic`. El programa implementa la más simple funcionalidad, imprimir un mensaje en la salida estándar e implementa un ciclo para imprimir cada uno de los argumentos que recibe el programa.

Se inicia con la codificación del programa de prueba en `C` cuya funcionalidad se muestra en la fig. 4.32. Posterior a ello se crea el artivo `Makefile` que describe la forma de compilar la aplicación, el listado 4.4 muestra esta descripción.

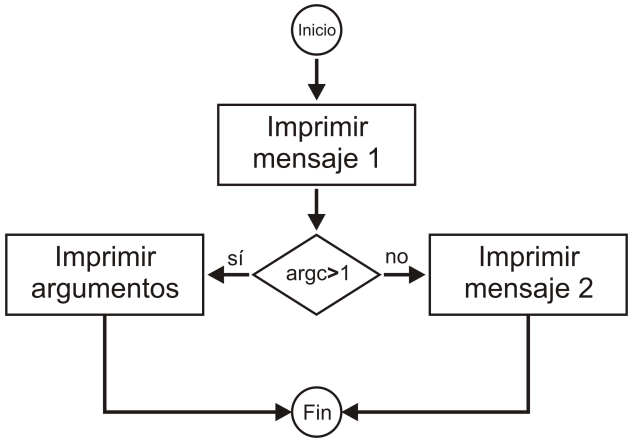


Figura 4.32: Funcionalidad del programa ejemplo `hello.c`.

Listado 4.4: Archivo `Makefile` asociado a `hello.c`.

```
1 CFLAGS += -O2
2
3 all: hello
4
5 hello: hello.c
6         ${CC} ${CFLAGS}  hello.c -o hello-mcic
7
8 clean:
9         rm -f hello-mcic
```

El siguiente paso es la creación de la receta que le indica a `bitbake` la forma como debe obtener, configurar, compilar y encapsular la aplicación `hello-mcic`. El listado 4.5 describe procedimiento del archivo `hello-mcic_git.bb`.

Listado 4.5: Receta `hello_mcic.bb` para `bitbake`.

```
1  DESCRIPTION = "Simple hello world app"
2  LICENSE = "GPLv2"
3  LIC_FILES_CHKSUM =
       "file://${POKYBASE}/meta/files/common-licenses/GPLv2;
       md5=751419260aa954499f7abaabaa882bbe"
4  PR = "r0"
5  PV = "0.1"
6  BRANCH = "master"
7  SRCREV = "${AUTOREV}"
8  SRC_URI = "git:///mcic/git/hello-mcic;protocol=file;
       branch=${BRANCH}"
9  S = "${WORKDIR}/git"
10 do_install() {
11           install -d ${D}${bindir}
12           install -m 0700 hello-mcic ${D}${bindir}
13           install -d /mcic/ftp/${PN}-${PV}-${PR}/
14           install -m 0644 hello-mcic
                 /mcic/ftp/${PN}-${PV}-${PR}/
15 }
```

De acuerdo a la información descrita en [28, 29, 39] se deberá ejecutar: `bitbake hello-mcic` para generar la aplicación en función de los parámetros que se especifican en el texto de `hello-mcic_git.bb`. Por ejemplo, durante el proceso de instalación `do_install()` además de copiar el programa `hello-mcic` a la carpeta de archivos ejecutables, se hace una copia a la carpeta `/mcic/ftp/hello-mcic-0.1-r0/` desde donde se podrá descargar usando la utilidad `wget` y la conexión de red en la tarjeta ML507. Esto requerirá la configuración de un servidor en el sistema de desarrollo que exporte el contenido de la carpeta `/mcic/ftp`.

4.6. Interceptación de llamadas a funciones

La interceptación de llamadas a funciones de bibliotecas dinámicas es un concepto que se presenta como solución al problema donde se desea agregar una función hardware a un SoPC pero no se tiene acceso al código fuente de la aplicación. Sin acceso al código no se puede adaptar el funcionamiento

del programa de usuario a fin de que pueda hacer uso del módulo hardware. El método propuesto funciona para llamadas a funciones de bibliotecas dinámicas que realiza un programa de usuario.

Las llamadas a funciones de bibliotecas dinámicas opera de la siguiente manera:

Ejecución de programa de usuario: El proceso inicia con la ejecución del programa de usuario que realiza llamadas a funciones de bibliotecas dinámicas.

Llamada a función: En algún momento durante la ejecución de la aplicación se realiza la llamada a una función de biblioteca. Cuando esto sucede, el subsistema *DLL* (*Dynamic Linker Loader*) de Linux busca en directorios pre-establecidos la biblioteca dinámica donde se encuentra la función solicitada. Cuando la biblioteca es localizada, DLL abre el `archive`[1] y carga el contenido de la función en memoria. Esta área de memoria será asignada al proceso solicitante de manera exclusiva o no, dependiendo de los parámetros especificados en el `archive` de la biblioteca. Una vez que el área de memoria donde reside la función solicitada se anexa al programa de usuario, DLL calcula las nuevas direcciones y establece estos parámetros en la tabla de símbolos asociada al programa de usuario para que la ejecución pueda llevarse a cabo.

Ejecución de función: El proceso de ejecución en software de una función de biblioteca dinámica no difiere del proceso habitual de ejecución del resto del programa.

Terminación: El proceso de terminación puede ser invocado de manera explicita o no. Cuando se invoca de manera explicita se utiliza la función `dlclose` pero debe existir un apuntador al área de memoria donde reside la funcionalidad que se abrió previamente con `dlopen`. Los servicios que DLL también provee son apertura, cargar, ejecución y terminación (cerrar una biblioteca dinámica) de manera no explicita.

A continuación se describe el proceso para interceptar llamadas a funciones de bibliotecas dinámicas.

[1] En el contexto del sistema operativo Linux un *archive* es un archivo que tiene una organización específica para almacenar datos y código ejecutable.

Ejecución de programa de usuario: El proceso inicia con la ejecución de manera habitual de un programa de usuario que realiza llamadas a funciones de bibliotecas dinámicas.

Llamada a función: Durante su ejecución, se realiza la llamada a una función de biblioteca, pero antes de que DLL abra la biblioteca de funciones en el directorio pre-determinado, busca la función requerida en las bibliotecas que indica la variable de sistema `LD_PRELOAD`. Es aquí cuando se carga la biblioteca que contiene funciones con el mismo prototipo que las originales, pero que difiere en su comportamiento (funcionalidad). Si DLL encuentra la función requerida, abre la biblioteca y la pone a disposición del programa solicitante para su ejecución y el proceso se desarrolla de manera habitual.

Ejecución de función: El proceso de ejecución de la llamada interceptada no difiere del proceso de ejecución habitual, lo que cambia es el comportamiento de la nueva función. En este caso particular, las llamadas a funciones hardware usarán el mismo principio de acceso que los programas de usuario mostrados en 5.6 con la única diferencia que su funcionalidad se provee a través de una *función envoltura* (*wrapper function*).

Terminación: El proceso de terminación es invocado de manera no explicita por DLL; la terminación de la aplicación que solicita la función se realiza de manera habitual.

El ejemplo mostrado en 5.7 hace uso de los servicios que provee DLL para abrir, cargar, ejecutar y terminar (cerrar) una biblioteca dinámica de manera no explicita y muestra el procedimiento necesario para capturar llamadas a funciones de bibliotecas dinámicas.

La figura 4.33 muestra el esquema de operación de la llamada a una función de biblioteca dinámica y también muestra el esquema propuesto de ejecución cuando se ha capturado la llamada a la función.

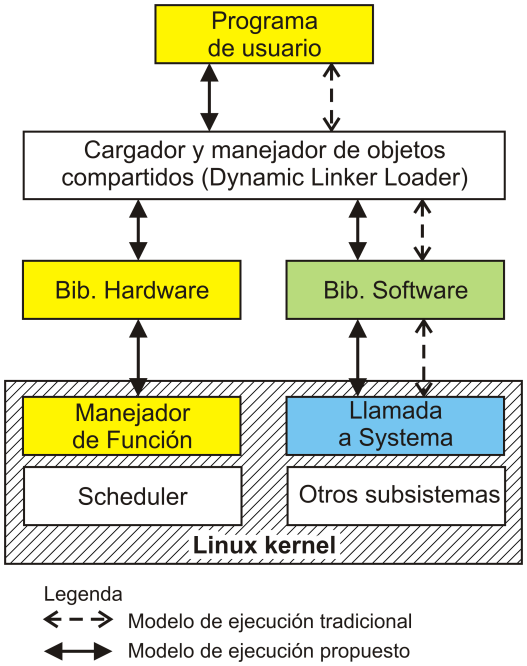


Figura 4.33: Interceptación de llamadas a funciones.

4.7. Resumen

Este capítulo presentó la metodología propuesta para la generación de bibliotecas de funciones hardware donde se describe el procedimiento para la generación de la plataforma hardware, la generación de los módulos de funciones hardware, la parametrización de la arquitectura, la generación del sistema operativo, la generación del módulo manejador de funciones hardware, la generación de aplicaciones de usuario y la interceptación de llamadas a funciones de bibliotecas dinámicas.

Capítulo 5

Experimentación y resultados

Este capítulo presenta los resultados experimentales para los elementos hardware propuestos y las pruebas realizadas al sistema operativo y a la capa software para la interacción entre funciones hardware y sistema operativo, también se incluyen las pruebas software para ilustrar el proceso de captura y ejecución de llamadas a funciones de bibliotecas dinámicas.

5.1. Generación de la Plataforma Hardware

Al seguir los pasos descritos en 4.1, se obtiene la plataforma hardware `my_sys_0`. Para disponer de una plataforma básica se eliminaron los cuatro módulos `xps_gpio` que se añaden de manera predeterminada para acceder a `Leds8, LedsPos, DipSw y PushBtn` puesto que serán sustituidos por dos módulos hardware en etapas subsecuentes del desarrollo. También se deberá seguir el procedimiento descrito en 4.1.1 para integrar la primera aplicación de prueba.

Tras concluir estos procedimientos se tiene un archivo *ELF* que está enlazado estáticamente, es decir, las funciones que requiere han sido incrustadas en el código de la aplicación para su correcto funcionamiento y por ello no requiere de ningún intérprete para su ejecución.

El listado 5.1 muestra las principales características del archivo ejecutable `memory_tests_0.elf` las cuales fueron obtenidas a través de la utilidad de sistema `readelf`. En dicho listado se muestra la clase del archivo, la versión *ABI*(*Application Binary Interface.*), el tipo de archivo, el tipo de arquitectura (tipo de procesador sobe el cual se ejecuta), la dirección del punto de entrada,

las secciones del programa y los símbolos dinámicos y estáticos.

Listado 5.1: Información del archivo `memory_tests_0.elf`.

```
ELF Header:
  Class:                             ELF32
  Data:                              2's complement, big endian
  OS/ABI:                            UNIX - System V
  Type:                              EXEC (Executable file)
  Machine:                           PowerPC
  Entry point address:               0xfffffffc

Section to Segment mapping:
  Segment Sections...
   00     .text .init .fini .rodata ...
   01     .tbss .boot0
   02     .tbss .boot

There is no dynamic section in this file.
```

Es importante conocer esta información debido a que se sebe utilizar un archivo tipo *elf* enlazado estáticamente para inicializar los *BRAMs* del `bitstream` que configura al FPGA.

El entorno de desarrollo en *SDK* permite generar diversos tipos de ejecutables en función de los parámetros del BSP, las bibliotecas usadas y las banderas de configuración indicadas al momento de hacer la compilación.

El proceso de síntesis de la plataforma hardware del SoPC ocupó los recursos físicos mostrados en la tabla 5.1.

La prueba inicial de operatividad sobre la plataforma hardware se basa en ejercitar la memoria principal del sistema a través de la escritura y lectura de patrones de datos. Las figuras 5.1a muestra el esquema de operación de la función principal mientras que la fig. 5.1b muestra la representación esquemática de las operaciones realizadas para escribir y leer datos de la memoria principal. Este proyecto sirve también para verificar la funcionalidad del puerto serial y el correcto funcionamiento de un componente vital para el SoPC, la memoria del sistema.

La tarjeta de desarrollo se conecta por el puerto serie hacia el sistema anfitrión de desarrollo a través de un adaptador USB-Serial y un cable serial cruzado (tipo *Null-Modem*, pero solo se requiere que Rx y Tx estén cruza-

Tabla 5.1: Recursos usados por el SoPC "my_sys_0".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100	1	1
PLL	16	1	6
BUFIOs	10	8	80
IDELAYCTRL	13	3	22
BUFG	18	6	32
BlockRAM	9	14	148
IOBs	32	210	640
Slice Registers	10	4,874	44,800
Slice LUTs	10	4,544	44,800
Slices ocupados	28	3,190	11,200

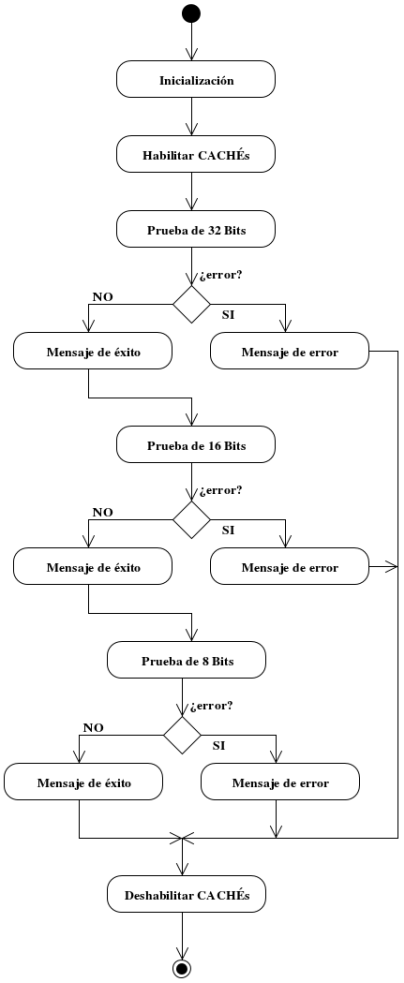
dos) como se muestra en la fig. 5.2 . En el sistema anfitrión se deberá usar alguna utilidad de sistema para abrir el puerto serie, un ejemplo de ello es el comando: `minicom -D /dev/ttyUSB0 -b 115200` . Este comando ejecuta el programa `minicom` para abrir el puerto serie `/dev/ttyUSB0` a `115200` BPS.

Los detalles para descargar un archivo `bitstream` de configuración del FPGA se detallan en [48].

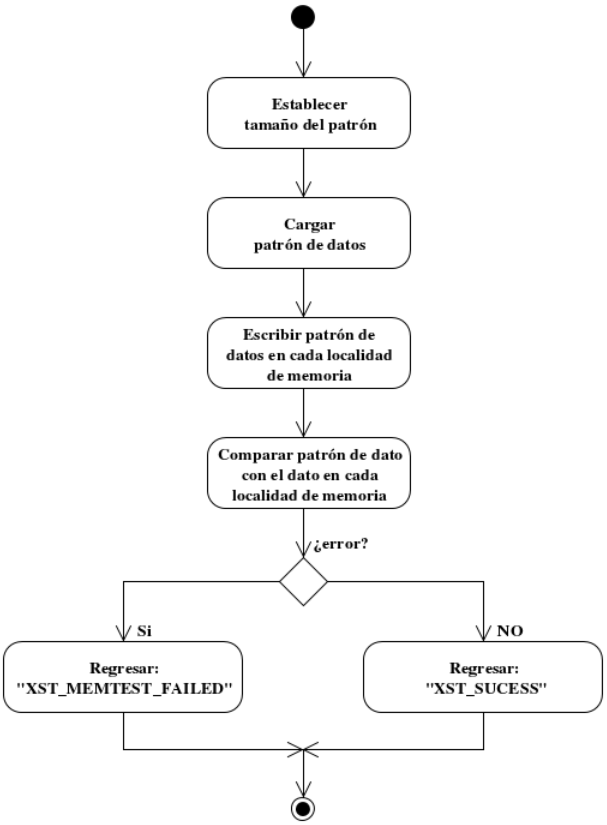
La consola del sistema anfitrión despliega la información mostrada en el listado 5.2 al ejecutar el programa de prueba `memory_tests_0`.

Listado 5.2: Salida del programa `memory_tests_0`.

```
-- Entering main() --
Starting MemoryTest for DDR2_SDRAM:
  Running 32-bit test... PASSED!
  Running 16-bit test... PASSED!
  Running 8-bit test... PASSED!
-- Exiting main() --
```



(a) Diagrama de operación principal.



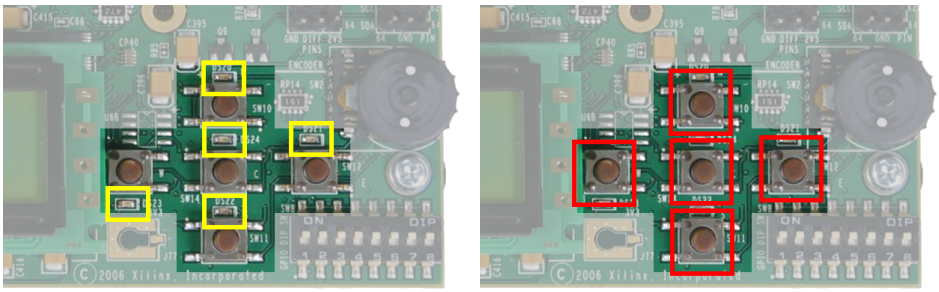
(b) Función de escritura/lectura para las pruebas de 32, 16 y 8 bits.

Figura 5.1: Programa de prueba de `TestApp_Memory.c`.

Figura 5.2: Interconexión entre la tarjeta de desarrollo ML507 y el sistema de desarrollo.

5.1.1. Integración del IPCore XGPIO

Posterior a la prueba funcional de la plataforma hardware se agrega un dispositivo genérico de entrada y salida de datos para hacer una prueba sobre `LedsPos` y `PushBtn` de la tarjeta de desarrollo ML507, los cuales se muestran en las figs. 5.3a y 5.3b respectivamente. El IPCore *xps_gpio* de Xilinx es un módulo hardware de propósito general para la entrada o salida de datos (*General Purpose Input Output*); incluye dos canales con comunicación bi-direccional; además, la parametrización del módulo permite seleccionar el ancho del vector de entrada/salida en cada canal. La figura 5.4 muestra un esquema funcional de `xgpio` donde se muestran las señales de interés y la estructura interna del *IPCore*. Este sistema utilizó como proyecto hardware `my_sys_0` al cual se añadió el IPCore `xps_gpio` para generar `my_sys_xgpio`. Producto del proceso de síntesis se genera un reporte con la cantidad de recursos utilizado cuyos datos más importantes se concentran en la tabla 5.2.



(a) `LedsPos`. (b) `PushBtn`.

Figura 5.3: Elementos del periférico `xgpio`.

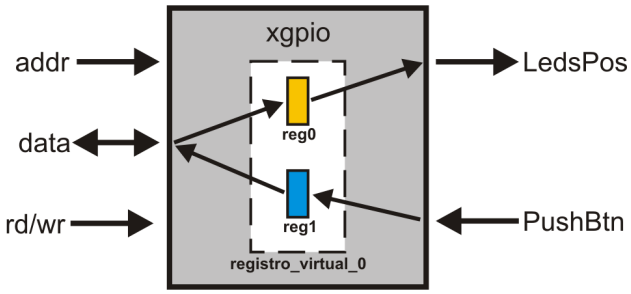


Figura 5.4: Esquema funcional de `xgpio`.

Tabla 5.2: Recursos usados por el SoPC "`my_sys_xgpio`".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100.00	1	1
PLL	16.67	1	6
BUFIOs	10.00	8	80
IDELAYCTRL	13.64	3	22
BUFG	18.75	6	32
BlockRAM	9.46	14	148
IOBs	34.37	220	640
Slice Registers	11.10	4,971	44,800
Slice LUTs	10.28	4,609	44,800
Slices ocupados	28.49	3,191	11,200

La prueba realizada consiste en generar un programa que envía y recibe datos al IPCore `xgpio`. Las figs. 5.5a y 5.5b muestran el diagrama de bloques de la función de lectura y escritura respectivamente.

El listado 5.3 muestra la salida observada en la terminal serial al ejecutar la aplicación `xgpio_test_0`.

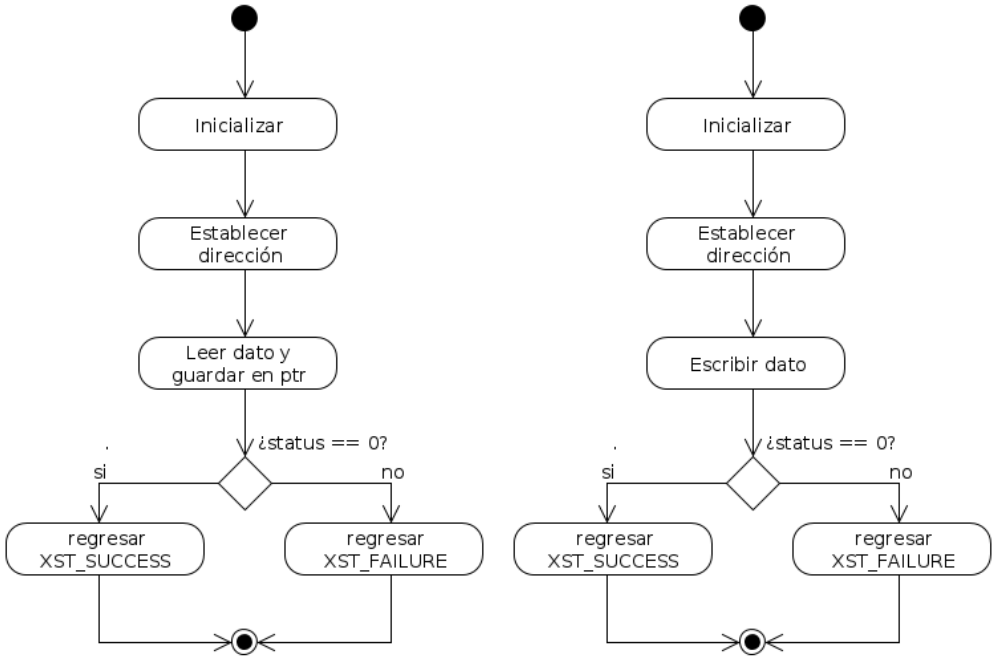
Listado 5.3: Salida del programa `xgpio_test_0`.

```
-- Entering main() --

Running GpioOutputExample() for LEDs_Positions...
GpioOutputExample PASSED.

Running GpioInputExample() for Push_Buttons_5Bit...
GpioInputExample  PASSED. Read data:0x0

-- Exiting main() --
```



(a) Lectura de datos para `xgpio_test_0`.

(b) Escritura de datos para `xgpio_test_0`.

Figura 5.5: Principales funciones de `xgpio_test_0`.

5.2. Biblioteca de funciones hardware

La metodología propuesta en esta tesis tiene como objeto desarrollar bibliotecas de funciones hardware las cuales asisten a una tarea software en el proceso de ejecución. Esta sección desarrolla los módulos hardware que forman parte del repositorio `myrepo` las cuales posteriormente serán usadas en la implementación final del SoPC.

5.2.1. Función Hardware MyGPIO

La función hardware `mygpio` implementa una interfaz de acceso a la función hardware donde sólo se tiene un registro virtual. A través de él se realizan la entrada y salida de datos, esto significa que cuando recibe una petición de escritura, el módulo hardware captura un dato del bus, y lo almacena en `reg0` para su posterior envío a `Leds8`; algo similar ocurre en el proceso de lectura, el módulo hardware captura los datos presentes en `DipSw`, los almacena en `reg1` y los pone a disposición del módulo *IPIF* para su posterior uso; la figura 5.6 muestra esta estructura.

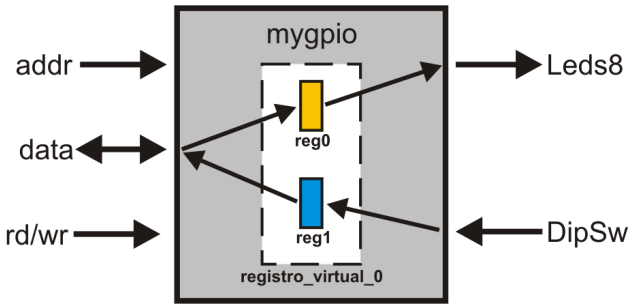


Figura 5.6: Esquema funcional de `mygpio`.

La fig. 5.7 muestra ubicación de los dispositivos periféricos conectados a `mygpio` en la tarjeta de desarrollo ML507.

Figura 5.7: `Leds8` y `DipSw` en la tarjeta ML507.

Tabla 5.3: Recursos usados por el SoPC "`my_sys_mygpio`".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100.00	1	1
PLL	16.67	1	6
BUFIOs	10.00	8	80
IDELAYCTRL	13.64	3	22
BUFG	18.75	6	32
BlockRAM	9.46	14	148
IOBs	36.88	236	640
Slice Registers	11.27	5,049	44,800
Slice LUTs	10.35	4,641	44,800
Slices ocupados	28.75	3,220	11,200

Esta función se integra a la plataforma hardware `my_sys_0` para formar `my_sys_mygpio`. Esta plataforma genera un archivo de configuración y un reporte de síntesis donde se encuentran los detalles de implementación para esta plataforma. La tabla 5.3 concentra los resultados más importantes derivados del proceso de síntesis de esta plataforma.

Para esta función hardware se desarrollo un programa de lectura y escritura de datos. La funcionalidad del programa captura datos de `DipSw` y escribe el dato capturado en `Leds8` mostrando copia de estos datos en la consola de la computadora. Este programa no utiliza la MMU como lo haría el sistema operativo Linux, sin embargo requiere de las instrucciones de sincronización mostradas en la tabla . Para este caso particular se utilizó `sync` y `eieio` para garantizar la correcta operación del programa de prueba.

El listado 5.4 muestra los detalles de la implementación de las funciones de entrada y salida de datos; y el listado 5.5 muestra la salida observada en la consola al ejecutar el programa `mygpio_test_0`.

Listado 5.4: Funciones de entrada y salida de datos dentro de `mygpio_test_0`.

```
1  unsigned int getdata(unsigned int addr) {
2      volatile unsigned int din;
3      din = *(volatile unsigned int *)(addr);
4      __asm__ __volatile__ ("eieio");
5      __asm__ __volatile__ ("sync");
6      return din;
7  }
8
9  void putdata(unsigned int dout, unsigned int addr) {
10     *(volatile unsigned int *)(addr) = dout;
11     __asm__ __volatile__ ("eieio");
12     __asm__ __volatile__ ("sync");
13 }
```

Listado 5.5: Salida del programa `mygpio_test_0`.

```
-- MyGPIO I/O test --
Data in : 0x03
Data out: 0x03
-- Test completed --
```

5.2.2. Función Hardware MyAdder

La función hardware `myadder` implementa la función suma para números enteros de 32 bits sin signo. La figura 5.8 muestra el esquema de implementación con las señales más relevantes al proceso de operación de la función hardware. Esta función se conecta al bus `PLB` de la plataforma hardware `my_sys_0` para integrar la nueva plataforma `my_sys_myadder`. La figura 5.9 muestra el resultado de la simulación del núcleo operativo de esta función.

La tabla 5.4 concentra los resultados más importantes derivados del proceso de síntesis de esta plataforma.

Debido a que esta función se integra como un periférico, se utilizan las funciones prototipo de para lectura y escritura de datos mostrados en el listado 5.4. La funcionalidad del programa `myadder_test_0` es muy parecida

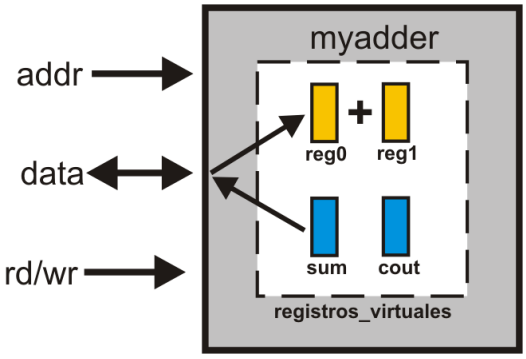


Figura 5.8: Representación esquemática de la estructura de la función hardware `myadder`.

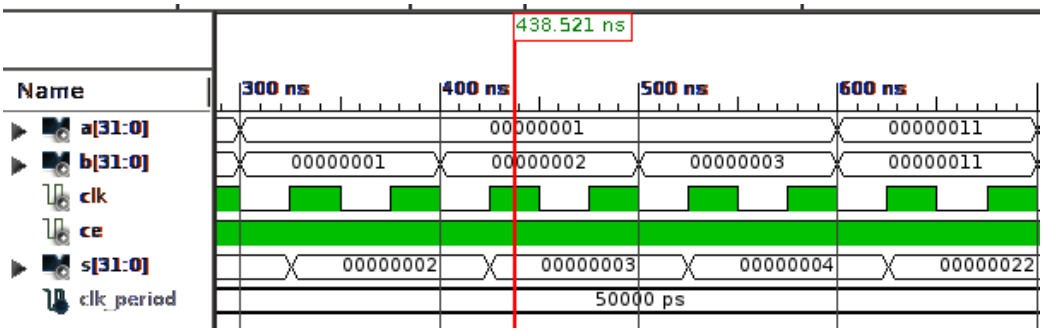


Figura 5.9: Simulación del núcleo operacional de la función hardware `myadder`.

a la funcionalidad del programa `mygpio_test_0` debido a que ambos hacen escritura y lectura de datos a un *elemento externo*. El listado 5.6 muestra la salida observada en la consola al ejecutar el programa `myadder_test_0`.

Listado 5.6: Salida del programa `myadder_test_0`.

```
-- MyAdder test --
reg0 = 0x33
reg1 = 0x22
sum  = 0x55
cout = 0x0
-- Test completed --
```

Tabla 5.4: Recursos usados por el SoPC "`my_sys_myadder`".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100.00	1	1
PLL	16.67	1	6
BUFIOs	10.00	8	80
IDELAYCTRL	13.64	3	22
BUFG	18.75	6	32
BlockRAM	9.46	14	148
IOBs	32.81	210	640
Slice Registers	11.31	5,067	44,800
Slice LUTs	10.40	4,662	44,800
Slices ocupados	27.75	3,108	11,200

5.2.3. Función Hardware BitSwp

La función hardware `bitswp` implementa la función de intercambio de bits, es decir, el bit 31 del vector de entrada se direcciona al bit 0 del vector de salida, el bit 30 de entrada va al 1 de salida y así sucesivamente para bits restantes del vector de entrada. La figura 5.10 muestra el esquema de implementación con las señales más relevantes al proceso de operación de la función hardware. Esta función se conecta al bus `PLB` de la plataforma hardware `my_sys_0` para integrar la nueva plataforma `my_sys_bitswp`. El listado 5.7 contiene la descripción comportamental para el núcleo operativo de esta función, y la fig. 5.11 muestra el resultado de la simulación para esta función.

Listado 5.7: Núcleo de operación de la función hardware *bitswapper*.

```
1 -- C_SLV_DWIDTH tiene valor 32
2 process(datain)
3 begin
4     for i in datain'low to datain'high loop
5         my_swap_sig(i) <= datain((C_SLV_DWIDTH-1)-i);
6     end loop;
7 end process;
8 dataout <= my_swap_sig;
```

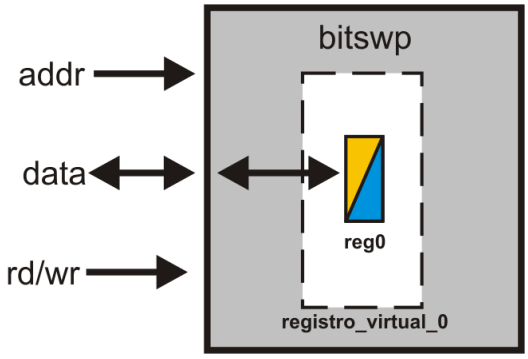


Figura 5.10: Esquema funcional de `bitswp`.

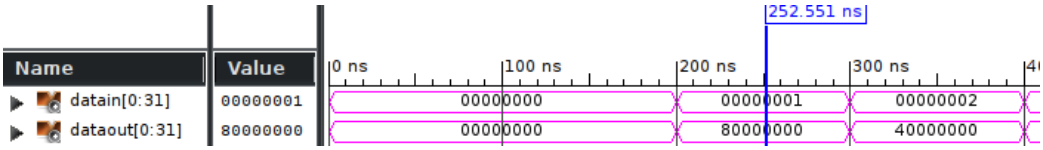


Figura 5.11: Simulación del núcleo operacional de la función hardware `bitswp`.

La tabla 5.5 concentra los resultados más importantes derivados del proceso de síntesis de esta plataforma.

Esta función al igual que las anteriores se integra como un periférico siguiendo los procedimientos descritos en 4.2 y se utilizan las funciones prototipo de para lectura y escritura de datos mostrados en el listado 5.4 para interactuar con esta función hardware. La funcionalidad del programa `bitswp_test_0` es muy parecida a la funcionalidad del programa `mygpio_test_0` debido a que ambos hacen escritura y lectura de datos a un *elemento externo* al procesador con la diferencia de que esta función se conecta al exterior del FPGA. El listado 5.8 muestra la salida observada en la consola al ejecutar el programa `bitswp_test_0`.

Tabla 5.5: Recursos usados por el SoPC "`my_sys_bitswp`".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100.00	1	1
PLL	16.67	1	6
BUFIOs	10.00	8	80
IDELAYCTRL	13.64	3	22
BUFG	18.75	6	32
BlockRAM	9.46	14	148
IOBs	32.81	210	640
Slice Registers	11.23	5,030	44,800
Slice LUTs	10.31	4,619	44,800
Slices ocupados	28.37	3,177	11,200

Listado 5.8: Salida del programa `bitswp_test_0`.

```
-- BitSwp test --
Dout = 0x0000cafe
Din  = 0x7f530000
-- Test completed --
```

5.2.4. Función Hardware CgAdder

La función hardware `cgadder` se construyó a partir de un núcleo operativo derivado de la biblioteca de funciones hardware de Xilinx. El procedimiento para agregar estas funciones se describe como parte de la metodología en 4.2.2. Se utiliza el modelo de un dispositivo periférico para integrar la función hardware a la arquitectura del sistema.

La tabla 5.6 concentra los resultados más importantes derivados del proceso de síntesis de esta plataforma.

Esta función implementa la función suma para número enteros donde cada número utiliza una representación de 32 bits. La fig. 5.12 muestra el esquema funcional de la función hardware `cgadder`, el prefijo `cg` indica que el núcleo operativo de esta función fue generado por *Core Generator* de Xilinx.

Tabla 5.6: Recursos usados por el SoPC "`my_sys_cgadder`".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100.00	1	1
PLL	16.67	1	6
BUFIOs	10.00	8	80
IDELAYCTRL	13.64	3	22
BUFG	18.75	6	32
BlockRAM	9.46	14	148
IOBs	32.81	210	640
Slice Registers	11.31	5,068	44,800
Slice LUTs	10.41	4,663	44,800
Slices ocupados	27.68	3,101	11,200

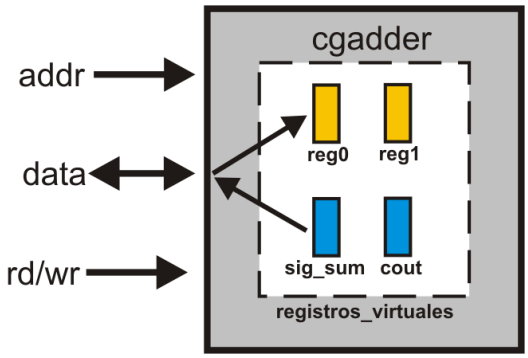


Figura 5.12: Esquema funcional de `cgadder`.

La prueba de funcionalidad implementada en el programa `cgadder_test_0` escribe y lee datos a la función hardware bajo prueba. Dicho programa se auxilia de las funciones de entrada y salida de datos mostradas en el listado 5.4. El listado 5.9 muestra la ejecución del programa como se muestra en la consola.

Listado 5.9: Salida del programa `cgadder_test_0`.

```
-- CgAdder test --
reg0    = 0x12341234
reg1    = 0x50c7d963
cgsum   = 0x62FBEB97
cgcout  = 0x00000000
-- Test completed --
```

5.2.5. Función Hardware CgMult

`Cgmult` utiliza la unidad funcional optimizada para la ejecución en hardware de multiplicaciones de *Core Generator*. El módulo hardware puede realizar multiplicaciones de dos número enteros de hasta 32 bits cada uno. La fig. 5.13 muestra el esquema funcional para `cgmult`. `Core Generator` recibió parámetros de configuración para la generación de un módulo combinacional de multiplicación, el cual fue instanciado en el IPCore `cgmult_v1_00_a` para su integración a la plataforma hardware `my_sys_cgmult` cuyos reporte de síntesis se presenta en la tabla 5.7.

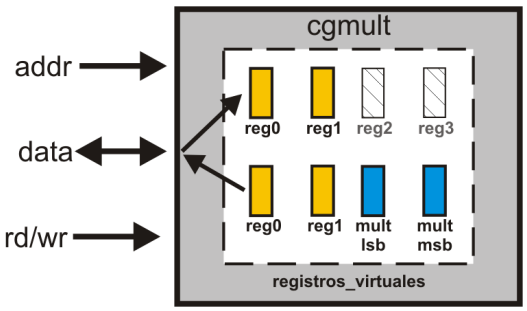


Figura 5.13: Esquema funcional de `cgmult`.

Tabla 5.7: Recursos usados por el SoPC "`my_sys_cgmult`".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100.00	1	1
PLL	16.67	1	6
BUFIOs	10.00	8	80
IDELAYCTRL	13.64	3	22
BUFG	18.75	6	32
BlockRAM	9.46	14	148
IOBs	32.81	210	640
Slice Registers	11.29	5,058	44,800
Slice LUTs	12.86	5,763	44,800
Slices ocupados	37.64	4,216	11,200

Listado 5.10: Salida del programa `cgmult_test_0`.

```
-- CgMult test --
reg0    = 0xcafe
reg1    = 0xbabe
MultLSB = 0x94133484
MultMSB = 0x0
-- Test completed --
```

5.2.6. Función Hardware CgDiv

La función hardware `cgdiv` utiliza un módulo funcional provisto por la biblioteca de funciones hardware de Xilinx. Este núcleo operativo en particular fue configurado para implementar su funcionalidad en `Slices`. Además de ello, los parámetros de diseño propuestos especifican la creación de un módulo funcional combinacional. La fig. 5.14 muestra el esquema funcional de este módulo hardware y la tabla 5.8 muestra los detalles más importantes del reporte de síntesis para la función hardware `cgdiv`.

La ejecución del programa de prueba `cgdiv_test_0` implementa la misma funcionalidad que los anteriores, sin embargo, lo hace para un dispositivo que está mapeado en la dirección base `0x80012000`.

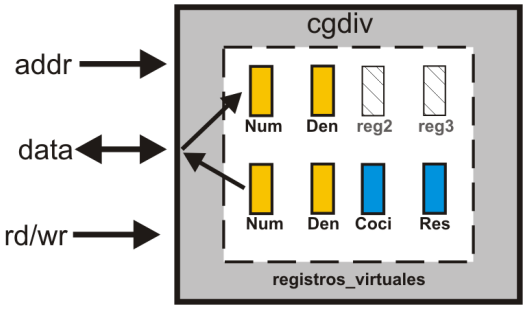


Figura 5.14: Esquema funcional de `cgdiv`.

Tabla 5.8: Recursos usados por el SoPC "`my_sys_cgdiv`".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100.00	1	1
PLL	16.67	1	6
BUFIOs	10.00	8	80
IDELAYCTRL	13.64	3	22
BUFG	18.75	6	32
BlockRAM	9.46	14	148
IOBs	32.81	210	640
Slice Registers	18.60	8,336	44,800
Slice LUTs	13.46	6,033	44,800
Slices ocupados	40.38	4,522	11,200

El listado 5.11 muestra la salida observada en la consola de texto al ejecutar el programa `cgdiv_test_0`.

Listado 5.11: Salida del programa `cgdiv_test_0`.

```
-- CgDiv test --
reg0 = 0x0000CAFE
reg1 = 0x0000BABE
coci = 0x00000001
res  = 0x00001040
-- Test completed --
```

Tabla 5.9: Repositorio de funciones hardware `myrepo`.

IPCore	Reg. Virtuales	Reg. Físicos	Flip-Flops
mygpio	1	2	-
myadder	2	3	1
bitswp	1	1	-
cgadder	2	3	1
cgmult	4	4	-
cgdiv	4	4	-

5.2.7. Características del repositorio de funciones hardware

Derivado de este proceso de desarrollo se ha generado un repositorio que funciona como biblioteca de funciones hardware debido a que el diseñador de un SoPC sólo requiere agregar el IPCore a la plataforma en desarrollo para hacer uso de la funcionalidad provista.

El repositorio de este proyecto está formado por las funciones hardware mostradas en la tabla 5.9. Dicho repositorio sigue la especificación estructural descrita en [54] y mostrada en las figuras 4.13, 4.14 y 4.15.

La tabla 5.9 muestra el número y tipo de elementos de almacenamiento usados en la operación e implementación de las funciones hardware. Los registros virtuales de cada IPCore son los registros que el módulo `IPIF` proporciona como interfaz de la función hardware al resto del sistema. Los registros físicos son elementos internos a cada función cuya función es almacenar un dato para su posterior uso. Los *flip flops* tienen como objetivo capturar el bit de acarreo de salida (C_{out}) para las funciones de suma y ponerlo a disposición del módulo `IPIF`.

La tabla 5.10 concentra la información relativa a la máxima frecuencia de operación reportada por las herramientas de síntesis de Xilinx. Dicha tabla también muestra los tiempos de sincronización entre las señales de entrada y el reloj (Entrada–Reloj) del módulo hardware, así como el tiempo requerido en la sincronización entre las señales de salida y el reloj (Salida–Reloj). Finalmente esta tabla muestra el periodo mínimo de tiempo necesario para la operación de la función hardware.

Con la excepción de la función hardware `bitswp`, las funciones restantes requieren un ciclo de reloj para capturar los datos de entrada y un ciclo

Tabla 5.10: Máxima frecuencia de operación, tiempos de sincronización y retardo.

Función	Max f_{MHz}	Entrada–Reloj t_{ns}	Salida–Reloj t_{ns}	Retardo t_{ns}
xgpio	312.402	1.399	0.471	3.201
mygpio	427.533	2.72	3.33	2.339
myadder	310.174	3.057	3.358	3.224
bitswapper	427.533	2.797	3.347	2.339
cgadder	310.174	3.057	3.358	3.224
cgmult	99.736	3.162	3.376	10.026
cgdiv	331.973	3.162	3.376	3.012

más para ponerlo a disposición del módulo `IPIF` para su posterior envío al procesador.

Los detalles de implementación se encuentran concentrados en la tabla 5.11 para cada función de la biblioteca desarrollada a partir de la metodología propuesta.

A partir del análisis de la información en dicha tabla, se determina que la implementación de la función `mygpio` utiliza menos recursos que la función `xgpio` esto se debe a que `mygpio` no incluye elementos lógicos para implementar un canal de comunicación bidireccional. El IPCore `mygpio` fue explícitamente diseñado para enviar datos a `Leds8` y recibir datos de `DipSw` por ello no fue necesario implementar la comunicación bidireccional.

La comparativa resultante entre la implementación de la función hardware `myadder` y `cgadder` muestra que la implementación de la función suma para números enteros de 32 bits es muy similar en términos de recursos usados, sin embargo se aprecia la optimización de la biblioteca de funciones hardware en *Core Generator*.

La función hardware más rápida es `bitswp` debido a que no requiere capturar el dato para invertir la posición de los bits. Los recursos mostrados en la tabla 5.11 son el resultado de la implementación del módulo `IPIF` requerido para la integración de la función hardware al bus PLB del sistema `my_sys_all`. Esta función utiliza recursos de enrutamiento para implementar su funcionalidad.

La función hardware `cgmult` es la menos rápida; esto se debe a los pa-

Tabla 5.11: Recursos usados por las funciones hardware.

Función	Registers	LUTs	LUT+FF	% Uso
xgpio	103	55	119	0.26
mygpio	87	43	101	0.22
myadder	193	112	247	0.55
bitswapper	157	73	175	0.39
cgadder	184	113	228	0.51
cgmult	190	1209	1333	2.98
cgdiv	3468	1479	3651	8.15

rámetros de configuración usados en la generación del núcleo operativo de `cgmult`. La correspondiente entrada en la tabla antes mencionada muestra la cantidad de recursos lógicos usados en la implementación combinacional de esta función.

La función hardware `cgdiv` ocupa la mayor cantidad de recursos lógicos del FPGA; por si misma utiliza una cantidad equiparable a los recursos usados por la plataforma hardware base `my_sys_0` en lo relativo a número de *Slices*.

Analizando la información proporcionada por la tabla 5.11 se observa que las funciones `cgmult` y `cgdiv` ocupan el 11.13 % de los los recursos del FPGA, lo cual en magnitud es mayor al 10.14 % de recursos usados por la plataforma base `my_sys_0`.

Finalmente, la tabla 5.12 muestra los recursos usados en la plataforma hardware `my_sys_all` donde se concentra la información relativa al uso de elementos lógicos y recursos dedicados del FPGA en esta implementación.

5.3. Parametrizar la arquitectura hardware

La parametrización de la arquitectura hardware es la representación textual de la forma como están organizados los elementos en la plataforma hardware.

Tabla 5.12: Recursos usados por el SoPC "`my_sys_all`".

Tipo de recurso	Porcentaje	Usados	Total
PPC440	100.00	1	1
PLL	16.67	1	6
BUFIOs	10.00	8	80
IDELAYCTRL	13.64	3	22
BUFG	18.75	6	32
BlockRAM	9.46	14	148
IOBs	36.88	236	640
Slice Registers	20.61	9,232	44,800
Slice LUTs	17.33	7,764	44,800
Slices ocupados	42.62	4,773	11,200

5.3.1. Generación del *device tree*

Para generar el *device tree* de las plataformas hardware utilizadas en el desarrollo de este proyecto se utilizó el BSP desarrollado por Xilinx con los parámetros de configuración mostrados en la tabla 4.1.

La fig 5.15 muestra la representación gráfica de un fragmento de la estructura final de la plataforma hardware `my_sys_all` y el listado 5.12 muestra la representación textual descriptiva asociada a la figura antes mostrada.

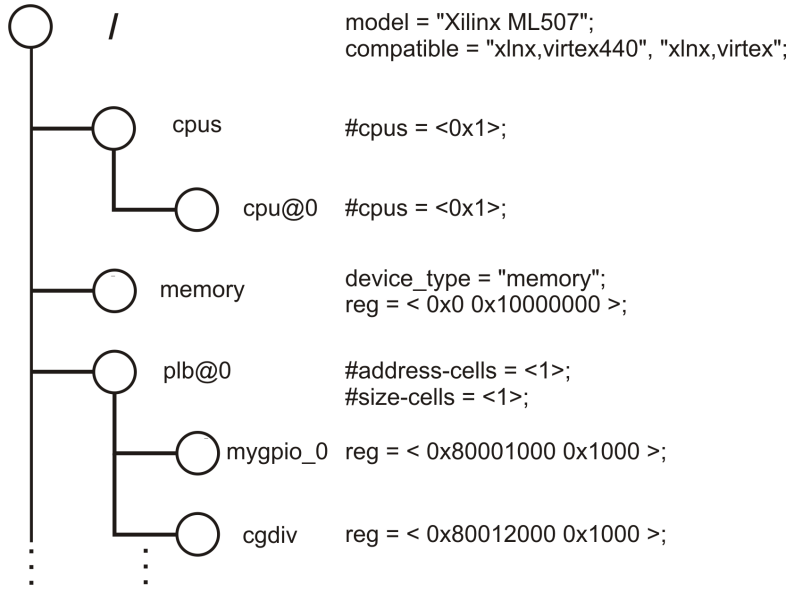


Figura 5.15: Representación de elementos dentro del *device tree*.

Listado 5.12: Fragmento del *device tree* para `my_sys_all`.

```
/dts-v1/;
/ {
    compatible = "xlnx,virtex440", "xlnx,virtex";
    dcr-parent = <&ppc440_0>;
    DDR2_SDRAM: memory@0 {
        reg = < 0x0 0x10000000 >;
    } ;
    chosen {
        bootargs = "console=ttyUL0 root=/dev/xsa2 rw
            rootfstype=ext3 ip=off";
        linux,stdout-path = "/plb@0/serial@84000000";
    } ;
    cpus {
    ppc440_0: cpu@0 {
            clock-frequency = <125000000>;
            compatible = "PowerPC,440", "ibm,ppc440";
            ...
    }
    plb_v46_0: plb@0 {
        compatible = "xlnx,plb-v46-1.05.a",
            "xlnx,plb-v46-1.00.a", "simple-bus";
        ranges ;
        mygpio_0: mygpio@80001000 {
            compatible = "xlnx,mygpio-1.00.a";
            reg = < 0x80001000 0x1000 >;
            xlnx,mygpio-width-i = <0x8>;
            xlnx,mygpio-width-o = <0x8>;
            ...
        } ;
        cgdiv_0: cgdiv@80012000 {
            compatible = "xlnx,cgdiv-1.00.a";
            reg = < 0x80012000 0x1000 >;
            xlnx,family = "virtex5";
            xlnx,include-dphase-timer = <0x1>;
        } ;
    ...
    }
```

5.4. Generación Sistema Operativo

El sistema operativo para cualquiera de las plataformas hardware mostradas anteriormente se puede generar en base al procedimiento descrito en 4.4.

En la versión final, se generó un sistema operativo para la plataforma hardware `my_sys_all` con los parámetros listados en las tablas 4.3, 4.4 y 4.5.

Este sistema operativo generado soporta el estándar *ABI* (*Application Binary Interface*) para la arquitectura PPC de 32 bits de manera general.

Los archivos ejecutables que cumplen con el estándar *ABI* permiten:

- Especificar tamaño de datos.
- Alinear estructuras de datos.
- Especifica cómo se deben pasar los parámetros de una función.
- Especifica cómo se debe establecer el valor de retorno de una función.
- Especifica cómo una aplicación debe interactuar con el sistema operativo (Llamada a sistema).

Y por ello se puede ejecutar en este sistema todo programa que cumpla con esta especificación, En algunos casos se requerirán funciones de bibliotecas adicionales, pero el sistema garantiza la compatibilidad a nivel binario para ejecutar un programa en diversas plataformas obteniendo los mismos resultados.

Por ello es necesario que el sistema operativo tenga soporte para la ejecución de archivos binarios que cumplan con la especificación *ABI*.

Linux ofrece un conjunto de módulos de kernel que permiten habilitar estas funciones y proporcionar servicios para la correcta ejecución de la aplicación.

El kernel de Linux contiene diversos subsistemas en conjunto suman 13.9 millones de líneas de código para la versión 2.6.37. Las funciones y servicios que proporciona generan una capa de abstracción que proporciona una interfaz común para otros programas o servicios facilitando la interacción entre ellos y también permite administrar los recursos hardware del sistema. Esta capacidad de administrar los recursos hardware es una de las ventajas que proporcionan los sistemas operativos que usan un núcleo de Linux.

Tabla 5.13: Archivos más relevantes generados a través de las herramientas de Poky y Xilinx.

Descripción
`u-boot-ml507.ace`: Archivo para configurar el FPGA, incluye el cargador de arranque `u-boot`.
`uImage-virtex5.bin`: Imagen en formato binario del núcleo del sistema operativo.
`uImage-virtex5.dtb`: `Device Tree Blob` de la arquitectura del SoPC.
`poky-image-minimal-virtex5.tar.gz`: Sistema de archivos del SoPC.
`modules-2.6.37+-r12-virtex5.tgz`: Módulos del núcleo del sistema operativo.

Para el caso particular del nuevo paradigma de ejecución hardware/software, representa una ventaja competitiva puesto que el sistema puede ser adaptado para ejecutar de manera concurrente diversos módulos funcionales tanto hardware como software para dar solución a un problema específico en el menor tiempo posible haciendo uso eficiente de los recursos presentes en el sistema.

Este trabajo no pretende realizar pruebas a los diversos subsistemas o al kernel en su conjunto. Sin embargo se estudian las estructuras internas y la organización que provee el núcleo operativo del sistema a fin de permitir la ejecución de funciones hardware a partir de llamadas a funciones de bibliotecas.

Producto de este proceso de compilación se generan diversos archivos, de entre ellos, los más relevantes al proceso de configuración del SoPC son descritos en la tabla 5.13.

El listado 5.13 presenta la salida observada en la consola donde se muestra la ejecución del cargador de arranque y el listado 5.14 muestra la ejecución del núcleo del sistema operativo.

Listado 5.13: Ejecución del cargador de arranque `u-boot`.

```
1  U-Boot 1.3.4-00332-g801fd9b-dirty (Jan 10 2013 -
       10:17:22)
2  CPU:    Xilinx PowerPC 440 UNKNOWN (PVR=7ff21912) at
       125 MHz
3          32 kB I-Cache 32 kB D-Cache
4  ### No HW ID - assuming ML507+alegzc
5  DRAM:   256 MB
6  FLASH:  32 MB
7  In:     serial
8  Out:    serial
9  Err:    serial
10 Hit any key to stop autoboot:  0
11 reading uimage.dtb
12 24027 bytes read
13
14 reading uimage.bin
15 1999429 bytes read
```

Listado 5.14: Ejecución del núcleo del sistema operativo.

```
1  ## Booting kernel from Legacy Image at 03020000 ...
2     Image Name:    Linux-2.6.37+
3     Image Type:    PowerPC Linux Kernel Image (gzip
          compressed)
4     Data Size:     1999365 Bytes =  1.9 MB
5     Load Address: 00000000
6     Entry Point:  00000000
7     Verifying Checksum ... OK
8  ## Flattened Device Tree blob at 03000000
9     Booting using the fdt blob at 0x3000000
10    Uncompressing Kernel Image ... OK
11    Loading Device Tree to 007f7000, end 007ffdda ... OK
12    ...
13 Yocto (Built by Poky 5.0.2) 1.0.2 virtex5 ttyUL0
14
15 virtex5 login:
```

5.5. Generación del módulo manejador de funciones hardware

El módulo manejador de funciones hardware implementa código que será ejecutado por el núcleo del sistema operativo. La figura 5.16 es una representación abstracta del modo de operación del manejador de funciones. El modelo aquí propuesto complementa la funcionalidad provista por el modelo de ejecución de dispositivos periféricos tipo caracter.

El subsistema de administración de acceso a memoria de Linux hará un mapeo del área de memoria virtual al espacio de direcciones de entrada/-salida. Este mapeo habilitará al módulo para hacer uso de las funciones de entrada y salida de datos que provee el núcleo del sistema operativo como parte de la API de interfaz con dispositivos externos.

A diferencia de las funciones de lectura y escritura de datos mostradas en el listado 5.4, una función de lectura y escritura de datos para el sistema operativo Linux requiere hacer uso de la funcionalidad provista por la MMU para realizar el mapeado de direcciones y de los mecanismos de sincronización mostrados en la tabla 3.4.

Esta funcionalidad se implementa de manera indirecta a través de la función `ioremap()` la cual regresa un apuntador al área de memoria virtual que está asociada a la memoria física de la función hardware.

El apuntador regresado por la función `ioremap()`, si es diferente de `NULL`, podrá ser utilizado como parámetro para las funciones de entrada y salida de datos que provee el núcleo de Linux. Además de ello se requiere garantizar la ejecución en orden de las peticiones recibidas lo cual se implementa con una o varias de las instrucciones mostradas en la tabla 3.4.

En este caso particular las operaciones de lectura y escritura de datos incluyen como parte de su procedimiento dos instrucciones en ensamblador que garantizan el orden de ejecución y sincronización de datos.

El núcleo de Linux provee dos familias de funciones para realizar envío y recepción de datos a dispositivos externos. Ambas familias de funciones leen o escriben una unidad de información (char = 1 byte, int = 4 bytes, double = 8 bytes, etc.) a un área de memoria especificada pero difieren en su implementación. Dos miembros del primer grupo de funciones son: `inb()` y `outb()`, mientras que `readw()` y `writew()` pertenecen al segundo grupo de funciones.

Los detalles de la arquitectura hardware del SoPC, desarrollado como

parte de la metodología propuesta, requieren el uso de funciones de entrada/salida del segundo grupo debido a la forma como se mapea el espacio de direcciones de los periféricos en memoria y a la forma como Linux administra la memoria virtual del sistema.

El API de acceso a periféricos del núcleo de Linux provee funciones de entrada/salida con sufijo `_be`. Este sufijo hace referencia al *endianismo* de la función, es decir, la función `writel_be` es una función de escritura para un tipo de dato *long int* (4 bytes) la cual reordena los bits del dato a escribir para cumplir con la especificación del formato `big endian`. Las funciones `writel_be()` y `readl_be()` se utilizaron en la implementación de las funciones de lectura y escritura de datos al igual que las instrucciones de ensamblador `eieio` y `sync`.

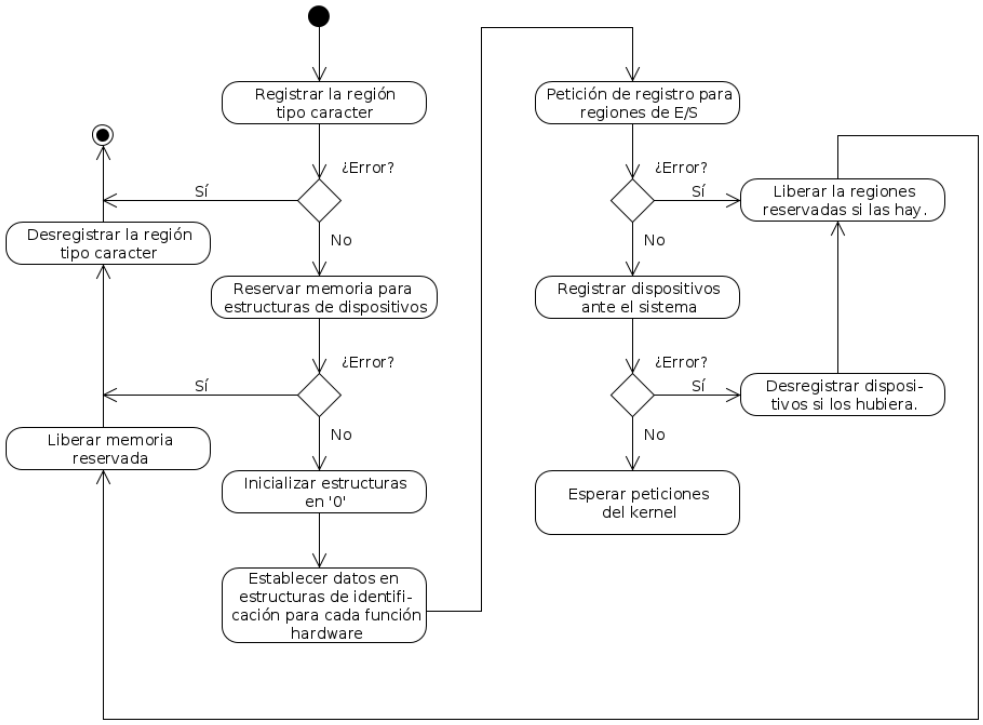


Figura 5.16: Abstracción del modelo de operación del módulo manejador de funciones hardware.

El subsistema de administración de recursos hardware ha evolucionado a

lo largo del tiempo y seguirá evolucionado para adaptarse a los requerimientos de las nuevas plataformas en desarrollo, sin embargo, las funciones provistas por el núcleo de Linux en su versión `2.6.37` permiten administrar los recursos a través de las APIs de acceso a periféricos, del manejo de memoria, el sistema de archivos y de la prioridad en los procesos de ejecución. Aunado a ello, la arquitectura del sistema también provee servicios de administración para el acceso a periféricos. Uno de esos ejemplos es el árbitro de acceso al bus PLB; este árbitro gestiona el acceso al bus.

De acuerdo a las especificaciones de la arquitectura de sistema *IBM CoreConnect* descritas en la sección 3.2.3, en una arquitectura de sistema donde se implementa el bus de comunicación PLB v4.6 únicamente los módulos maestros podrán iniciar secuencias de transferencias de datos. La arquitectura propuesta sólo tiene un módulo maestro que es el procesador y 16 módulo esclavos. Por ello la comunicación con las funciones hardware se lleva a cabo cuando el el módulo maestro ejecuta operaciones de carga o almacenamiento de datos.

El listado 5.15 presenta las regiones de memoria reservadas por el módulo manejador; dicha información se encuentra en el archivo `/proc/ioports` el cual se genera dinámicamente en función de las condiciones operativas del sistema. En dicho listado, el primer dato representa la dirección inicial, el segundo la dirección final y el tercero el nombre de la función hardware. La utilidad de sistema `lsmod` permite verificar que el módulo manejador se ha cargado en memoria, en este caso particular el listado 5.16 muestra la salida observada en la consola del sistema.

Listado 5.15: Regiones de memoria reservadas para cada función.

```
80000000-80000003 : xil_gpio
80001000-80001003 : my_gpio
80002000-80002007 : my_adder
80004000-80004003 : my_bitswp
80010000-80010007 : cg_adder
80011000-8001100f : cg_mult
80012000-8001200f : cg_div
```

Listado 5.16: Carga del módulo manejador.

```
root@virtex5:~# insmod manager.ko
MANAGER: All devices are ready!
root@virtex5:~# lsmod
    Not tainted
manager 6237 0 - Live 0xd1151000
```

5.6. Generación de aplicación de usuario

Las aplicaciones de usuario propuestas en esta sección se implementan siguiendo el esquema mostrado en la fig. 5.17. Para su implementación se requirió de la biblioteca estándar de `C` donde se implementan `fopen()`, `fread()`, `fwrite()` y `fclose()`. Este modelo de integración Hardware/Software ejemplifican el modelo propuesto en la fig. 1.1b.

Los resultados de la ejecución de estas aplicaciones se presentan a continuación.

5.6.1. Ejecución Hardware/Software para xgpio

El programa `xgpio` hace uso de la función hardware `xil_gpio`. Esta función lee y escribe datos a `PushBtn` y `LedsPos` respectivamente. Este programa ejemplifica las funciones de entrada y salida hacia dispositivos periféricos usando un IPCore proporcionado por Xilinx. El listado 5.17 muestra la salida observada en la consola al ejecutar el programa `xgpio` y la fig. 5.18 muestra la salida observada en la tarjeta de desarrollo.

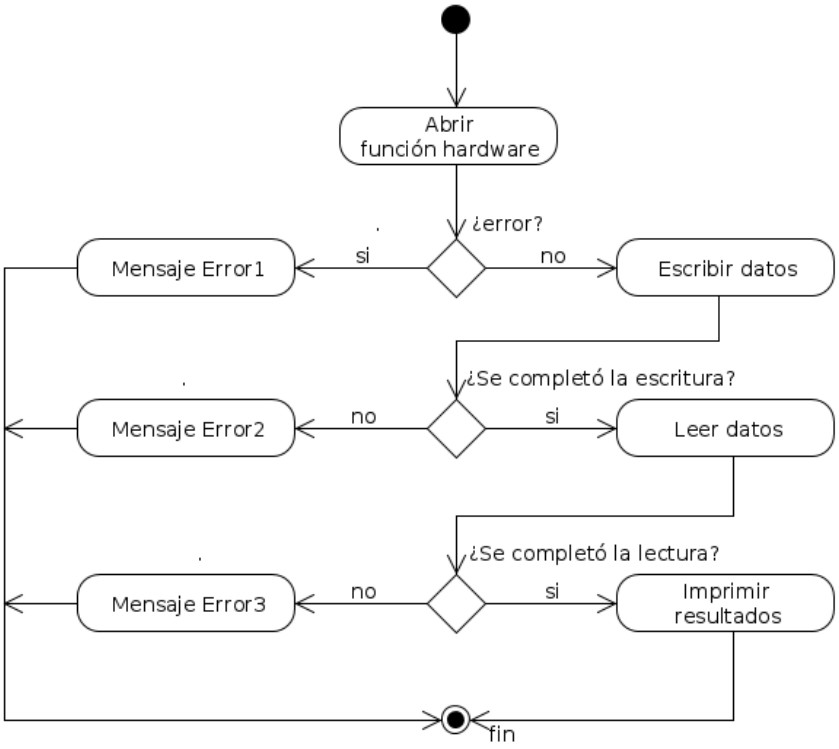


Figura 5.17: Esquema general de operación de las aplicaciones de usuario.

Listado 5.17: Salida del programa `xgpio`.

```
root@virtex5:~# xgpio 3
Xilinx GPIO test
    MANAGER(debug mode): Dev(60,1) opened!
    MANAGER(debug mode): Dev(60,1) is Xilinx GPIO 0
     xil_gpio: datain  = 0x00000000
     xil_gpio: access_ok(1)
     xil_gpio: dout = 0x00000003
Data in = 0x0
Dout(3) = 0x3
```

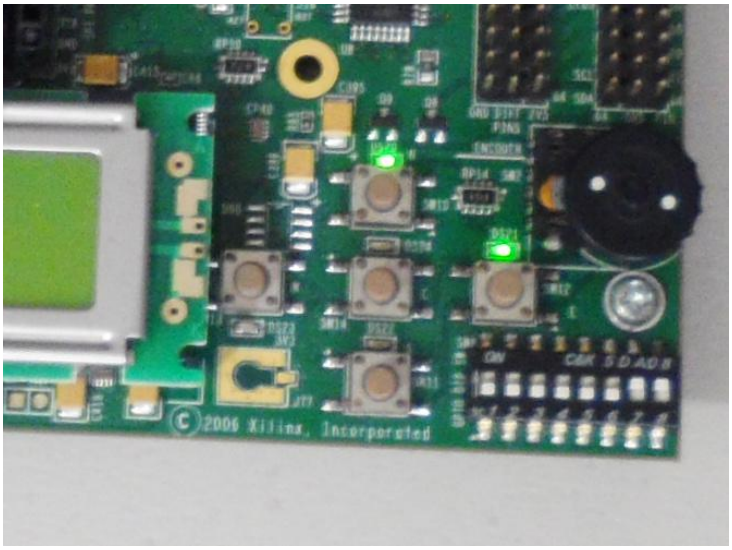


Figura 5.18: Salida observada en la tarjeta al ejecutar `xgpio`.

5.6.2. Ejecución Hardware/Software para mygpio

El programa `mygpio` hace uso de la función hardware `my_gpio` y de las funciones de la biblioteca estándar para interactuar con el dispositivo. Esta función lee y escribe datos a `Leds8` y `DipSw` respectivamente; también ejemplifica las funciones de entrada y salida hacia dispositivos periféricos usando un IPCore desarrollado para dispositivos hardware específicos. El listado 5.18 muestra la salida observada en la consola al ejecutar el programa `mygpio` y la fig. 5.19 muestra la salida observada en la tarjeta de desarrollo.

Listado 5.18: Salida del programa `mygpio`.

```
root@virtex5:~# mygpio 0xff
MY GPIO test
    MANAGER(debug mode): Dev(60,2) opened!
    MANAGER(debug mode): Dev(60,2) is My GPIO
    my_gpio: datain  = 0x00000007
    my_gpio: access_ok(1)
    my_gpio: dout = 0x000000FF
Data in = 0x7
Dout(0xff) = 0xFF
```

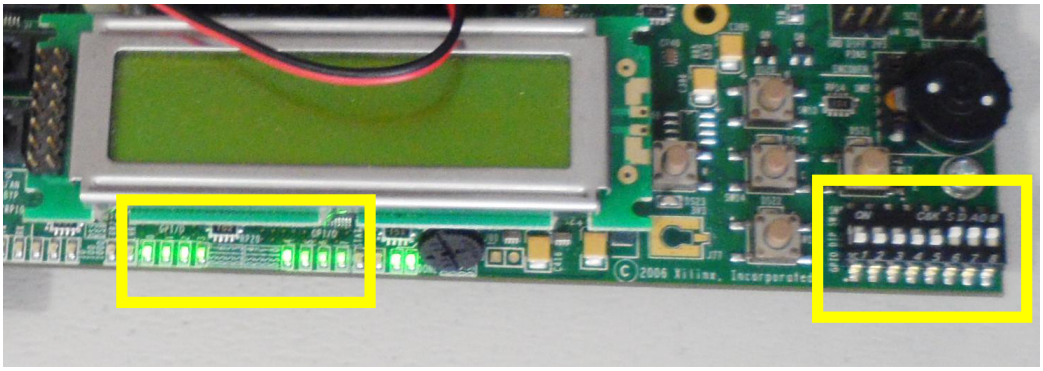


Figura 5.19: Salida observada al ejecutar el programa `mygpio`.

5.6.3. Ejecución Hardware/Software para myadder

Este programa escribe y lee datos a la función hardware `my_adder`; la función hardware implementa la suma de números enteros. El listado 5.19 muestra la salida del programa `myadder` como se observa en la consola de texto de la computadora. Esta función y las restantes de esta sección no están conectadas a dispositivos externos, sólo hacen uso de los elementos internos del FPGA para la implementación de su funcionalidad.

Listado 5.19: Salida del programa `myadder`.

```
root@virtex5:~# myadder  0xcafe 0xbabe
My Adder test
     MANAGER(debug mode): Dev(60,3) opened!
     MANAGER(debug mode): Dev(60,3) is My adder
      my_adder: s0 = 0x0000CAFE
      my_adder: s1 = 0x0000BABE
      my_adder: S[0] = 0x000185BC
      my_adder: S[1] = 0x00000000
A(0xcafe) = 0xCAFE
B(0xbabe) = 0xBABE
r2 = 0x000185BC
r3 = 0x00000000
```

5.6.4. Ejecución Hardware/Software para bitswp

Este programa fue usado como prueba de concepto para determinar la forma como interactúa el sistema operativo con las funciones hardware. A partir de los resultados observados al ejecutar esta función se determinó la necesidad de utilizar las funciones con sufijo `_be` (`writel_be()` y `readl_be()`) para transferir correctamente las palabras de datos entre el procesador y la función hardware. El listado 5.20 muestra la salida del programa `bitswp` como se observa en la consola de texto de la PC.

Listado 5.20: Salida del programa `bitswp`.

```
root@virtex5:~# bitswp 0xcafe
Bitswapper test
    MANAGER(debug mode): Dev(60,4) opened!
    MANAGER(debug mode): Dev(60,4) is Bitswapper
     my_bitswp: access_ok(1)
     my_bitswp: dout = 0x0000CAFE
     my_bitswp: datain  = 0x7F530000
Dout(0xcafe) = 0xCAFE
Din = 0x7F530000
```

5.6.5. Ejecución Hardware/Software para cgadder

Este programa escribe y lee datos a la función hardware `cg_adder`; la función hardware implementa la suma de números enteros. El listado 5.21 muestra la salida del programa `cgadder` como se observa en la consola de texto de la PC. Esta función demuestra la capacidad de la metodología propuesta para integrar módulos de la extensa biblioteca de funciones hardware de Xilinx.

Listado 5.21: Salida del programa `cgadder`.

```
root@virtex5:~# cgadder 0xcafe 0xbabe
Core Generator Adder test
    MANAGER(debug mode): Dev(60,5) opened!
    MANAGER(debug mode): Dev(60,5) is Core Generator Adder
     cg_adder: s0 = 0x0000CAFE
     cg_adder: s1 = 0x0000BABE
     cg_adder: S[0] = 0x000185BC
     cg_adder: S[1] = 0x00000000
A(0xcafe) = 0xCAFE
B(0xbabe) = 0xBABE
r2 = 0x000185BC
r3 = 0x00000000
```

5.6.6. Ejecución Hardware/Software para cgmult

Este programa escribe y lee datos a la función hardware `cg_mult`; la función hardware implementa la multiplicación de números enteros de hasta 32 bits. El listado 5.22 muestra la salida del programa `cgmult` como se observa en la consola de texto de la PC. En ella se observa la funcionalidad implementada por el programa `cgmult` y de la funcionalidad provista por el módulo manejador de funciones. Este módulo escribe dos datos y lee cuatro datos a partir de la dirección base virtual asignada al dispositivo. Si bien sólo se requiere leer los dos registros donde se almacena el resultado, la implementación lee cuatro registros para comparar los datos enviados con los recibidos cuando el módulo manejador de las funciones hardware se carga en su versión para depuración como se muestra en el listado 5.22.

Listado 5.22: Salida del programa `cgmult`.

```
root@virtex5:~# cgmult 0xcafe 0xbabe
CoreGen Multiplier test
    MANAGER(debug mode): Dev(60,6) opened!
    MANAGER(debug mode): Dev(60,6) is Core Generator
        Multiplier
    cg_mult: s0 = 0x0000CAFE
    cg_mult: s1 = 0x0000BABE
    cg_mult: S[0] = 0x0000CAFE
    cg_mult: S[1] = 0x0000BABE
    cg_mult: S[2] = 0x94133484
    cg_mult: S[3] = 0x00000000
A(0xcafe) = 0xCAFE
B(0xbabe) = 0xBABE
r0 = 0x0000CAFE
r1 = 0x0000BABE
r2 = 0x94133484
r3 = 0x00000000
```

5.6.7. Ejecución Hardware/Software para cgdiv

Este programa escribe y lee datos a la función hardware `cg_div`; la función hardware implementa la división de números enteros de hasta 32 bits. Los resultados están disponibles en un ciclo de operación debido a que la función se implementó de manera combinacional. El listado 5.23 muestra la salida del programa `cgdiv` como se observa en la consola de texto de la computadora.

Listado 5.23: Salida del programa `cgdiv`.

```
root@virtex5:~# cgdiv 0xcafe 0xbabe
CoreGen Divider test
    MANAGER(debug mode): Dev(60,7) opened!
    MANAGER(debug mode): Dev(60,7) is Core Generator Divider
    cg_div: s0 = 0x0000CAFE
    cg_div: s1 = 0x0000BABE
    cg_div: S[0] = 0x0000CAFE
    cg_div: S[1] = 0x0000BABE
    cg_div: S[2] = 0x00000001
    cg_div: S[3] = 0x00001040
A(0xcafe) = 0xCAFE
B(0xbabe) = 0xBABE
r0 = 0x0000CAFE
r1 = 0x0000BABE
r2 = 0x00000001
r3 = 0x00001040
```

5.6.8. Métricas de ejecución Hardware/Software

Las especificaciones del procesador[21] muestran que las operaciones de división se ejecutan en la unidad compleja para número enteros (*Complex Integer Unit*) como se muestra en la fig. 3.3. Esta unidad maneja todas las operaciones aritméticas, lógicas; también maneja las operaciones generadas en la atención de interrupciones, el manejo de la TLB y la escritura y lectura a registros del procesador.

Las operaciones de multiplicación para números enteros de 16×32 bits se ejecutan con una latencia de dos ciclos de reloj y las multiplicaciones de 32×32 bits con una latencia de tres ciclos de reloj. Sin embargo, las operaciones de división requieren 33 ciclos de reloj para completarse.

Las operaciones de carga y almacenamiento de datos a memoria se realizan por la unidad *L/S* (*Load/Store Unit*) mostrada en la fig. 3.3 sin embargo, las operaciones de *L/S* se rigen por el ciclo de operación del procesador. Como se sabe la arquitectura del procesador PowerPC puede ejecutar hasta dos instrucciones concurrentemente siempre y cuando las unidades funcionales requeridas estén libres.

La fig. 5.20 muestra el tiempo de ejecución total que le toma a la aplicación

de usuario realizar operaciones de división en el procesador (*Ejecución Sw*) y el tiempo total de ejecución que le toma realizar las operación de división accediendo a la función hardware `cgdiv` (*Ejecución Hw*).

La fig. 5.21 muestra detalles de los procesos de *Ejecución Sw* y *Ejecución Hw*. Se sabe que el CPU requiere 33 ciclos de reloj para completar la división de números enteros de hasta 32 bits, mientras que la función hardware `cgdiv` sólo requiere un ciclo de reloj para el mismo tipo de dato. En esta figura también muestra el tiempo requerido para realizar la transferencia de datos entre CPU↔Memoria y CPU↔Función hardware. En base a estos últimos datos se concluye que el tiempo de transferencia de datos entre CPU↔Función hardware causa un aumento significativo en el tiempo de *Ejecución Hw*. El factor real de aceleración proporcionado por la función hardware es `33` en términos del tiempo requerido para completar la operación de división, no obstante el tiempo total requerido por la aplicación de usuario para completar una operación de división hardware es 15 veces menos rápida que la división en Sw debido a la penalización en la transferencia de datos.

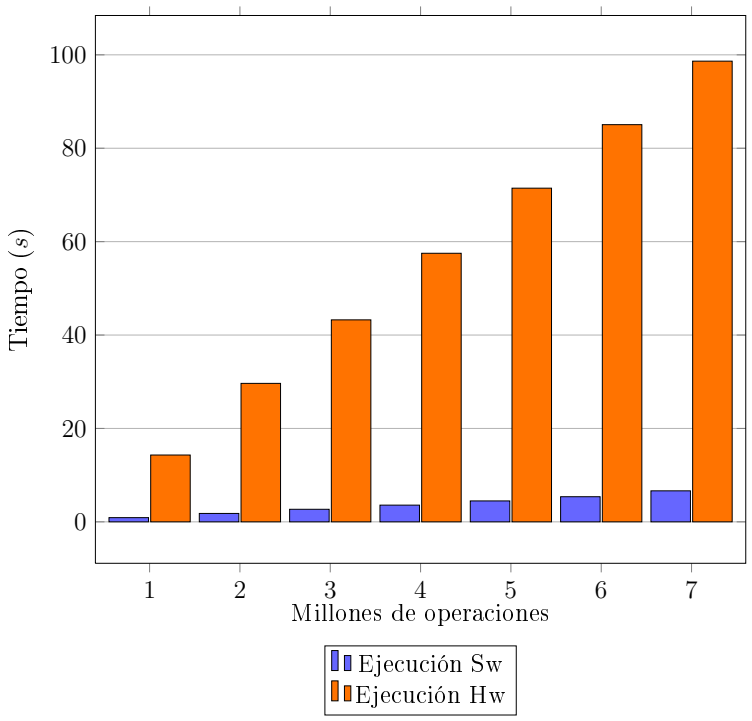


Figura 5.20: Métrica de ejecución para la función de división.

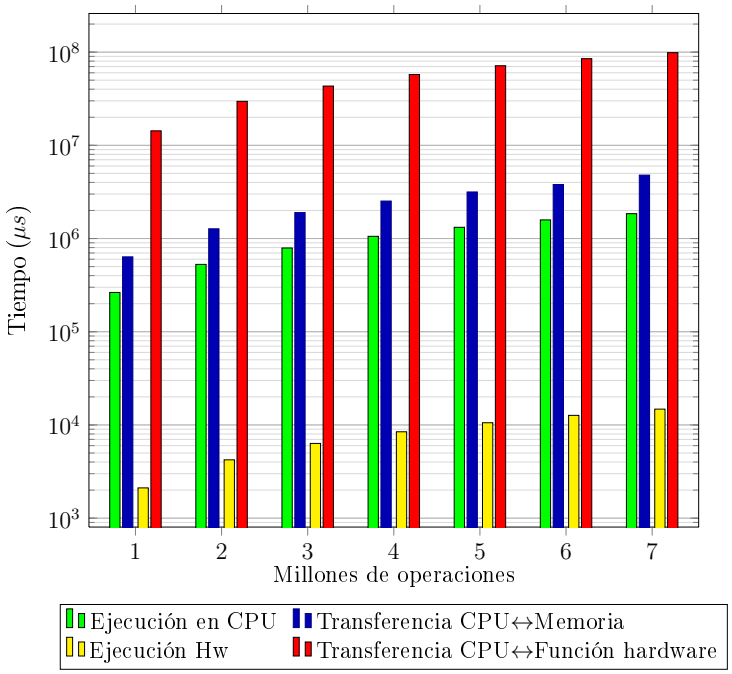


Figura 5.21: Detalles de ejecución para la función de división.

Esto se debe a que la unidad L/S debe iniciar una operación de lectura o escritura que pasa por los diferentes niveles de la jerarquía de memoria, al número de dispositivos maestros en el bus PLB, a las políticas de atención de peticiones del árbitro de bus, a la frecuencia de operación del bus[1], al tiempo de ejecución de una función hardware (latencia de ejecución), es por ello que el tiempo necesario para completar una operación de división hardware se incrementa en comparación con el tiempo requerido para completar una operación de división en la CPU.

Si además de ello el SoPC utiliza un sistema operativo, se deberá tomar en cuenta el tiempo requerido en la planificación de procesos, cambio de estados de ejecución, y a los mecanismos de sincronización y control de acceso a recursos hardware (puertos de entrada/salida). En promedio, el tiempo que toma realizar la transferencia de datos entre CPU↔Memoria es de $0.65\mu s$ por operación, y para CPU↔Función hardware es de $14.27\mu s$ por operación.

[1]Es posible que el procesador y el bus del sistema operen a frecuencias distintas y no obstante tener la capacidad para sincronizar sus transferencias a través de los elementos funcionales en cada módulo.

5.7. Interceptación de llamadas a funciones

La interceptación de llamadas a funciones presenta una solución muy interesante para combinar la ejecución software con ejecución hardware en programas que hacen llamadas a funciones de bibliotecas dinámicas.

Usando los procedimientos descritos en la sección 4.6 y en base a [14] se desarrolló una biblioteca dinámica llamada `libcdhwfn.so` (***lib****rary* ***c*** ***d****ynamic* ***h****ardware* ***fun****ction*) y un programa de prueba que hace llamadas a las funciones de esa biblioteca `dmain` (***d****ynamic* ***main***).

El listado 5.24 muestra la funcionalidad de la biblioteca `libcdhwfn.so` y el listado 5.25 muestra la funcionalidad del programa de usuario `dmain`.

De manera tradicional, las llamadas a funciones serán atendidas por DLL tal y como se describe en la sección 4.6. El resultado de ejecutar la aplicación `dmain` con con las funciones provistas por `libcdhwfn.so` se muestra en el listado 5.26.

La biblioteca `libcmydhwfn.so` (***lib****rary* ***c*** ***my*** ***d****ynamic* ***h****ardware* ***fun****ction*) implementa los mismos prototipos de funciones que `libcdhwfn.so` sin embargo, difiere en funcionalidad debido a que en el primer caso, la ejecución de la función `lib_fn_xgpio()` incrementa en uno el argumento que recibe mientras la nueva función incrementa su valor en **7** unidades. Esta función muestra la captura de llamadas a funciones y su posterior ejecución en software.

Para mostrar la capacidad de ejecución en hardware se utilizó el prototipo de función `lib_fn_mygpio` de la biblioteca original para implantar una nueva función en la nueva biblioteca con el mismo prototipo pero con diferente funcionalidad.

La nueva función escribe al elemento hardware `Leds8` el valor que recibe como argumento en lugar de incrementar en dos unidades el valor que recibe, pero esta nueva función también regresa el valor capturado de `DipSw`.

Este comportamiento puede ser sustituido por cualquier función hardware en la biblioteca, sin embargo se determinó usar la función hardware `my_gpio` para ilustrar de manera física la captura y ejecución de la llamada a función de biblioteca iluminando los leds (`Leds8`) y capturando el valor establecido en los interruptores (`DipSw`) de la tarjeta ML507.

El listado 5.27 muestra la funcionalidad de la nueva biblioteca de funciones llamada `libcmydhwfn.so` y el listado 5.28 muestra la salida observada en la consola de la PC, además se incluye la fig. 5.22 donde se observa la respuesta generada tras la captura y ejecución de la función `lib_fn_mygpio()`.

5.7.1. Bibliotecas dinámicas

Listado 5.24: Biblioteca `dhwfn.h`.

```
1  /* Biblioteca de funciones */
2  unsigned int lib_fn_xgpio(unsigned int dout) {
3          return dout+1;
4  }
5
6  unsigned int lib_fn_mygpio(unsigned int dout) {
7          return dout+2;
8  }
```

Listado 5.25: Programa `dmain.c`.

```
1  /* dmain.c  an example prog to show fn. call & int. */
2
3  #include <stdio.h>
4  #include <dhwfn.h>
5
6  unsigned int lib_fn_xgpio(unsigned int dout);
7  unsigned int lib_fn_mygpio(unsigned int dout);
8
9  int main(int argc, char *argv[]) {
10    int r,d,t;
11
12     if(argc > 1)  t = sscanf(argv[1], "%i", &d);
13     else  d = 5;/* default */
14
15     r = lib_fn_xgpio(d);
16     printf("lib_fn_xgpio(0x%X) = %d\n",d,r);
17
18     r = lib_fn_mygpio(d);
19     printf("lib_fn_mygpio(0x%X) = %d\n",d,r);
20
21     return 0;
22 }
```

Listado 5.26: Ejecución tradicional para bibliotecas dinámicas.

```
root@virtex5:~# dmain 1
lib_fn_xgpio(0x1) = 2
lib_fn_mygpio(0x1) = 3
```

Listado 5.27: Biblioteca dinámica `mydhwfn.h`.

```
1 /* Function interception example to be executed in Sw
      or Hw */
2 unsigned int hw_mygpio(unsigned int);/*Hw execution*/
3 unsigned int lib_fn_xgpio(unsigned int dout) {
4         return dout+7;
5 }
6 unsigned int lib_fn_mygpio(unsigned int dout) {
7         return hw_mygpio(dout);
8 }
```

Listado 5.28: Interceptación de funciones y ejecución hw/sw.

```
root@virtex5:~# export LD_PRELOAD=/home/root/libcmydhwfn.so

root@virtex5:~# dmain 1

lib_fn_xgpio(0x1) = 8

    MANAGER(debug mode): Dev(60,2) opened!
    MANAGER(debug mode): Dev(60,2) is My GPIO
    my_gpio: datain  = 0x0000000F
    my_gpio: access_ok(1)
    my_gpio: dout    = 0x00000001

    GPIO test using dynamic libs
    Data in = 0xF
    Data to write = 0x00000001

lib_fn_mygpio(0x1) = 15
```

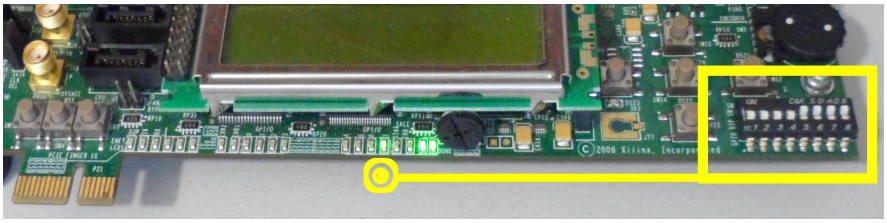


Figura 5.22: Ejecución hardware de la función interceptada.

5.8. Resumen

Este capítulo presentó los resultados de las pruebas realizadas a la plataforma hardware base, las funciones hardware en modo *stand–alone*, el sistema operativo, el manejador de funciones hardware, las aplicaciones de usuario para acceder a las siete funciones hardware y la ejemplificación del proceso de interceptación y ejecución de llamadas a funciones de bibliotecas dinámicas para su ejecución en software o hardware.

Conclusiones y trabajo a futuro

La industria ha puesto su atención en el nuevo paradigma de ejecución Hardware/Software para buscar metodologías, modelos y procedimientos que permitan hacer más eficiente la operación de un SoPC o dispositivo móvil. En este sentido el Centro de Investigación en Computación ha desarrollado proyectos de investigación que exploran aspectos específicos de los modelos de interacción Hardware/Software y propone el uso de estas tecnologías para colaborar en el desarrollo tecnológico del país.

La metodología generada en este trabajo describe las diferentes etapas a desarrollar para generar una biblioteca de funciones hardware e integrar dichas funciones a la arquitectura de un SoPC.

Se analizan tres modelos de interacción Hardware/Software de los cuales, dos fueron implementados para realizar las pruebas de concepto.

Además la propuesta se complementa con el modelo de interceptación de llamadas a funciones de bibliotecas dinámicas para su ejecución en hardware o software.

Es por ello que además de generar la metodología que describe los procedimientos necesarios para la generación de funciones hardware ejemplifica los mecanismos de operación de una función hardware dentro del sistema operativo.

También se colabora en la construcción de la infraestructura necesaria para poder generar, probar y validar esquemas de interacción Hardware/-Software para SoPC.

Derivado del análisis de las gráficas en 5.20 y 5.21 se concluye que el tiempo de ejecución aumentó significativamente debido a la penalización en la transferencia de datos entre el procesador y las funciones hardware. Esto en parte se debe a que el IPCore que administra el acceso al bus PLB (árbitro de bus) no implementa una interfaz nativa de 128 bits como se especifica en [21], en vez de esto implementa una interfaz de 32 bits para el bus de datos.

Sin embargo, la metodología aquí planteada no pretende sustituir las funciones que realiza el procesador del SoPC, la idea detrás de la propuesta es explorar el nuevo paradigma de ejecución Hardware/Software donde una parte de la tarea se ejecuta como procedimientos software y otra parte se ejecuta en un módulo hardware. Ello ha permitido analizar y comprender las implicaciones que tiene la arquitectura del SoPC en el proceso de co-ejecución Hardware/Software.

La plataforma hardware `my_sys_all` utiliza el 42 % de los recursos lógicos del FPGA. Esta plataforma se configuró para operar a 125MHz tanto a nivel del bus PLB como a nivel del procesador para evitar los efectos colaterales de la sincronización de las señales de captura.

Las siete funciones hardware desarrolladas utilizan el 14.13 % de los *Slices* del FPGA, pero las redes de enrutamiento pasaron de 8738 en el sistema `my_sys_0` a 15320 en el sistema `my_sys_all`, es necesario recordar que algunas de estas redes de interconexión hacen uso de recursos lógicos ($LUTs$) cuando las restricciones de diseño impiden el ruteo de las señales haciendo uso exclusivo de las líneas de transmisión internas del FPGA.

De este análisis también se observó que no existe el mismo nivel de crecimiento en el número de componentes interconectados en los sistemas base y final. El número de componentes interconectados (IOBs, Slices, PPC, BRAM, BUFG, PLL, DCM, etc.) dentro del FPGA pasaron de 3821 en el sistema base a 5451 en el sistema final.

Productos generados

1. Parámetros de configuración de paquetes y utilidades para el entorno de desarrollo en Poky. Estos parámetros son de mucha importancia debido a que a través de esta especificación se podrá regenerar un sistema operativo usando las herramientas de Poky en su versión 5.0.2. Además de ello, se modificaron las recetas del kernel especificadas en `linux--xilinx`, el cargador de arranque `u-boot-xilinx`, y se configuraron repositorios locales para tener un control sobre el código fuente de los principales paquetes que dan origen al núcleo de sistema operativo con la finalidad de estabilizar el sistema de desarrollo en Poky y evitar que los cambios generados en otras ramas de desarrollo generen errores durante la compilación del sistema operativo o de las utilidades que éste requiere.

2. Metodología para la implementación de bibliotecas de funciones hardware y el modelo de interceptación de llamadas a funciones de bibliotecas dinámicas para su ejecución en hardware o software. También se analizan y describen los modelos de interacción Hardware/Software para SoPC.

3. Biblioteca experimental de funciones hardware. La función de esta biblioteca es permitir la experimentación de nuevos modelos de interacción Hardware/Software para analizar las implicaciones de la arquitectura hardware en estos modelos.

4. Artículo en congreso 12^{th} International Congress on Computer Science CORE2012 titulado "*A Hardware/Software Co–Execution Model Using Hardware Libraries for a SoPC Running Linux*".

Trabajo a futuro

La metodología generada explora los procesos de interacción Hardware/-Software haciendo un especial énfasis en la capa de servicios software que habilitan los servicios de acceso a las funciones hardware. Sin embargo se observó que el esquema de implementación a nivel hardware causa un aumento significativo en el tiempo de ejecución debido a la penalización por transferencia de información entre el sistema y la función hardware.

Optimización al modelo de interconexión de funciones hardware para SoPC. La primera propuesta de trabajo a futuro ataca la deficiencia en la transferencia de información entre SoPC↔Función hardware observada en la fig. 5.21. Para ello, se deberá optimizar o proponer un nuevo modelo experimental de interconexión de funciones hardware a nivel de arquitectura del SoPC para disminuir el tiempo requerido en la transferencia de información. Se sugiere investigar, analizar, comparar y seleccionar de entre los modelos propuestos tanto por la comunidad de investigadores como por la industria, el modelo de interconexión que permita disminuir el tiempo de transferencia de información. Por ejemplo, el uso de la interfaz APU (*Auxiliary Processor Unit*) del PowerPC para interconectar *co-procesadores*, el bus DCR (*Device Configuration Register*), el bus PCI-Express, el bus FSL (*Fast Simple Link*) del *soft core MicroBlaze*, el bus WishBone, o DMA y PLB.

Modelo de ejecución para funciones hardware con transferencia en base a bloques de datos. Investigar, analizar y proponer de entre los modelos generados por la comunidad de investigadores y por la industria, un modelo de ejecución Hardware/Software que permita integrar funciones hardware (con transferencia en base a bloques de datos) y aplicaciones de usuario en un SoPC. Esto implica investigar, analizar y comparar los modelos de interacción a nivel hardware (PLB multi–maestro, APU, PCI-Express, FSL, DMA+PLB, etc.). Producto del modelo, generar una biblioteca de funciones hardware con transferencia en base a bloques de datos. La tarjeta de desarrollo ML507 permitirá explorar las propuestas que hacen uso de canales DMA para la transferencia de información entre Función hardware↔Memoria, el uso de la interfaz APU (*Auxiliary Processor Unit*) del PowerPC para interconectar *co-procesadores* o el bus PCI-Express.

Capa de servicios software para comunicación y administración de funciones hardware en SoPC. En lo relativo a la capa de servicios software y subsistemas dentro del sistema operativo, analizar las propuestas actuales de modelos de comunicación, operación y administración de acceso a funciones hardware a nivel de sistema operativo. Para ello, se deberá experimentar en base a los modelos propuestos y tomando en cuenta la funcionalidad provista por el kernel de Linux en el SoPC; como por ejemplo, los servicios de mapeo de puertos de entrada/salida (`io_remap()`) pero sin asociar dicha funcionalidad a un archivo especial tipo caracter (`/dev/mydev`) a fin de no usar los servicios de acceso al sistema de archivos y disminuir con ello la sobrecargar transaccional al enviar y recibir datos. La funcionalidad que provee el núcleo de Linux incluye, llamadas a sistema, funciones a nivel del núcleo (*kernel symbols*), los manejadores de dispositivos tipo caracter y tipo bloque.

Modelo de ejecución concurrente para SoPC. A nivel de aplicaciones de usuario, se propone explorar y evaluar los servicios de transferencia de información, planificación de tareas, mecanismos de sincronización y bloqueo, para lograr la creación de procesos multi–hilo con ejecución no bloqueante a fin de implantar un modelo de co-ejecución Hardware/Software que permita la concurrencia de rutinas software con elementos funcionales hardware (*hardware thread + software thread*).

El modelo propuesto hace uso de mecanismos de sincronización y bloqueo de acceso recursos hardware/software, la llamada a sistema `clone` con parámetros `CLONE_THREAD,CLONE_VM`, y la llamada a sistema `mmap()` para lograr la creación de hilos hardware y software que colaboran en la solución de un problema particular de manera concurrente.

Bibliografía

[1] μCLinux, Embedded Linux/Microcontroller Project. *Recurso en línea* `http://www.uclinux.org/`.

[2] Corp. Altera. SOPC Builder User Guide. Consultado el 25/07/2010. *Recurso en línea*. `www.altera.com/literature/ug/ug_sopc_builder.pdf` .

[3] Corp. Altera. Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions, May 2006. Consultado el 2/03/2011. *Recurso en línea*. `www.altera.com/literature/wp/wp-aghrdwr.pdf` .

[4] Peter Arato, Zoltan Adam Mann, and Andras Orban. Algorithmic aspects of hardware+software partitioning. Technical report, Budapest University of Technology and Economics. Department of Control Engineering and Information Technology, 2005.

[5] James H. Aylor, Barry W. Johnson, W.A. Wulf, and Sanjaya Kumar. *The Codesign of Embedded Systems: A Unified Hardware/Software Representation*. Springer, November 1995.

[6] Massimo Baleani, Frank Gennari, Yunjian Jiangt, Yatish Patel, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. 2002.

[7] Inc. Berkeley Design Technology. An independent evaluation of: The autoesl autopilot high-level synthesis tool. Technical report, Berkeley Design Technology, Inc., 2010.

[8] Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. A dynamic reconfiguration run-time system. 1997.

[9] DENX Software Engineering Corp. Embedded Linux Development Kit. Consultado el 07/10/2012. *Recurso en línea*. `http://www.denx.de/wiki/DULG/ELDK` .

[10] Intel Corp. Intel Presents Stellarton Chip in The Consumer Electronic Show. Las Vegas, NV. 2010. *Recurso en línea* `http://www.intel.com/idf` visitado en Diciembre 2010.

[11] Tom Davidson, Karel Bruneel, and Dirk Stroobandt. Run-time reconfiguration for automatic hardware/software partitioning. 2010.

[12] Qingxu Deng, Shuisheng Wei, Hai Xu, Yu Han, and Ge Yu. A reconfigurable rtos with hw/sw co-scheduling for sopc. 2005.

[13] Qingxu Deng, Yi Zhang, Nan Guan, and Zonghua Gu. A unified hw/sw operating system for partially runtime reconfigurable fpga based computer systems. 2008.

[14] Ulrich Drepper. How to write shared libraries. Technical report, December 2011. *Online resource*. `http://people.redhat.com/drepper/dsohowto.pdf`.

[15] Frederick Dufour and Martin Radetzki. Seminar "embedded systems": Hardware/software partitioning. Technical report, University of Stuttgart. Institute of Computer Architecture and Computer Engineering, 2006.

[16] Rolf Ernst, Jörg Henkel, and Thomas Benner. Hardware/software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, 1993.

[17] David Gibson and Benjamin Herrenschmidt. Device trees everywhere. page 7, Feb 2006.

[18] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design and Test of Computers*, 1993.

[19] Jörg Henkel and Rolf Ernst. The interplay of run-time estimation and granularity in hw/sw partitioning. 1997.

[20] Jörg Henkel and Rolf Ernst. A hardware/software partitioner using a dynamically determined granularity. 1998.

[21] Corp. IBM. Ibm powerpc 440 embedded core, 2006. *Recurso en línea* https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/f72367f770327f8a87256e63006cb7ec/$file/powerpc440_nov2006.pdf.

[22] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Electronic Systems*, 7(4):605–627, 2002.

[23] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre-Yves Droz. RAMP blue: A message-passing manycore system in FPGAs. 2007. *Recurso en línea* http://www.scidacreview.org/0904/html/hardware2.html.

[24] Adrián Alonso Lascano. Alligator os: An embedded operating system. Master's thesis, Instituto Politécnico Nacional. Centro de Investigación en Computación, 2011.

[25] David Lau, Jarrod Blackburn, and Charlie Jenkins. Using c-to-hardware acceleration in fpgas for waveform baseband processing. 2006.

[26] Trong-Yen Lee, Yang-Hsin Fan, and Chia-Chun Tsai. Adaptive multi-constraints in hardware-software partitioning for embedded multiprocessor fpga systems. *WSEAS Transactions on Computers*, 8(2), 2009.

[27] Linux Foundation. The yocto project, 2011. *Recurso en línea* https://www.yoctoproject.org.

[28] Linux Foundation. Yocto Project 1.0 Release Notes for Poky 5.0, 2011. *Recurso en línea.* https://www.yoctoproject.org/download/yocto-project-10-release-notes-poky-50.

[29] Linux Foundation. Yocto Project Quick Start, 2011. *Recurso en línea* http://www.yoctoproject.org/docs/1.0/yocto-quick-start/yocto-project-qs.html.

[30] Zoltan Adam Mann, Andras Orban, and Viktor Farkas. Evaluating the kernighan–lin heuristic for hardware/software partitioning. *International Journal of Applied Mathematics and Computer Science*, 2007.

[31] MICROSE-Lab. Alligator_SP. *Recurso en línea* `http://www.microse.cic.ipn.mx/files/documents/projects/alligator/reportes_tec/SIP-2007-TR-Alligator.pdf` , Feb 2008.

[32] Vaibhawa Mishra, Kota Solomon Raju, and Pramod Tanwar. Implantation of dynamically reconfigurable systems on chip with os support. *International Journal of Computer Applications*, 2012.

[33] Nicholas Moore, Albert Conti, and Miriam Leeser. Writing portable applications that dynamically bind at run time to reconfigurable hardware. *International Symposium on Field-Programmable Custom Computing Machines*, 2007.

[34] Andrew Morton. *Hardware/Software Partitioning and Scheduling of Embedded Systems*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada., 2004.

[35] Stephen Neuendorffer. Implementation issues in hybrid embedded systems. 2003.

[36] Raúl García Ordaz. Diseño de un rob-distribuido para procesadores superescalares. Master's thesis, Instituto Politécnico Nacional. Centro de Investigación en Computación, 2011.

[37] Andreas Popp, Yannick Le Moullec, and Peter Koch. Fast feasibility estimation of reconfigurable architectures. 2009.

[38] Abhijit Prasad, Wangqi Qiu, and Rabi Mahapatra. Hardware software partitioning of multifunction systems. 2002.

[39] Richard Purdie, Tomas Frydrych, Marcin Juszkiewicz, and Dodji Seketeli. Poky Handbook, 2010. Consultado el 07/03/2011. *Recurso en línea*. `http://pokylinux.org/doc/poky-handbook.html`.

[40] Proshanta Saha. Automatic software hardware co-design for reconfigurable computing systems. *International Conference on Field Programmable Logic and Applications (FPL 2007)*, 2007.

[41] Hayden Kwok-Hay So. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, University of California, Berkeley, 2007.

[42] Hayden Kwok-Hay So, Artem Tkachenko, and Robert Brodersen. A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers Using BORPH. 2006.

[43] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, November 2004.

[44] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. Technical report, 2012.

[45] Dimitris Theodoropoulos, Yana Yankova, Georgi Kuzmanov, and Koen Bertels. Automatic hardware generation for the molen reconfigurable architecture: a g721 case study. 2007.

[46] Herbert Walder and Marco Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. 2003.

[47] Grant Wigley and David Kearney. The first real operating system for reconfigurable computers. *Aust. Comput. Sci. Commun.*, 23(4):130–137, January 2001.

[48] Inc. Xilinx. ML505/ML506/ML507 Getting Started Tutorial. `http://www.xilinx.com/support/documentation/boards_and_kits/ug348.pdf`.

[49] Inc. Xilinx. Xilinx Embedded Development Kit (EDK). Consultado el 22/07/2010. *Recurso en línea*. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/edk_ctt.pdf` .

[50] Inc. Xilinx. Xilinx Platform Studio (XPS) Tool Suite. Consultado el 7/07/2010. *Recurso en línea*. `http://www.xilinx.com/tools/platform.htm` .

[51] Inc. Xilinx. Board Support Package. *Recursos en línea* `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/oslib_rm.pdf` `http://www.xilinx.com/ise/embedded/edk91i_docs/standalone_v1_00_a.pdf`, 2010.

[52] Inc. Xilinx. LogiCORE IP Processor Local Bus (PLB) v4.6, 2010.

[53] Inc. Xilinx. Edk concepts, tools, and techniques, 2011. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/edk_ctt.pdf.

[54] Inc. Xilinx. Embedded system tools reference manual, 2011. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/est_rm.pdf.