FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de fin de grado del Grado en Ingeniería Informática

2015-2016

Interfaz software/hardware entre Raspberry Pi y FPGA Spartan-6 y su aplicación a simulación dirigida por eventos

—Autores—
Juan Samper González
Roberto de la Cruz Martínez

—Director—
Juan Carlos Fabero Jiménez

Agradecimientos

A Juan Carlos, por el tiempo que nos han dedicado, así como por las pautas y consejos que nos ha dado para llevar a cabo este proyecto.

También, a José Luis Risco, por hacernos la evaluación del correcto funcionamiento al final del curso, cuando menos tiempo tenía.

Y para finalizar, a todos los familiares, amigos y profesores que nos han ayudado todo este tiempo.

"Nuestra recompensa se encuentra

en el esfuerzo y no en el resultado.

Un esfuerzo total es una victoria

completa"

Mahatma Gandhi

UNIVERSIDAD
COMPLUTENSE
MADRID

AUTORIZACIÓN PARA LA DIFUSIÓN DEL TRABAJO FIN DE GRADO Y SU DEPÓSITO EN EL REPOSITORIO INSTITUCIONAL E-PRINTS COMPLUTENSE

Los abajo firmantes, alumno/s y tutor/es del Trabajo Fin de Grado (TFG) en el Grado en **INGENIERIA INFORMATICA** de la Facultad de **INFORMATICA**, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el Trabajo Fin de Grado (TF) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

Periodo de embargo (opcional):

- ☐ 6 meses
- ☐ 12 meses

TÍTULO del TFG: Interfaz software/hardware entre Raspberry Pi y FPGA Spartan-6 y su aplicación a simulación dirigida por eventos

Curso académico: 2015 / 2016

Nombre del Alumno/s:

ROBERTO DE LA CRUZ MARTINEZ

JUAN SAMPER GONZALEZ

Tutor/es del TFG y departamento al que pertenece:

JUAN CARLOS FABERO JIMENEZ

Departamento de Arquitectura de Computadores y Automática (DACYA)

Firma del alumno/s   Firma del tutor/es

Índice

Resumen ……………………………………………………………………..9
Palabras Clave ………………………………………………………………9
Abstract…………………………………………………………….........11
Keywords………………………………………………………………...11
1. Introducción…………………………………………………………...13
1.1. Antecedentes…………………………………………………..13
1.2. Objetivos……………………………………………….........14
1.3. Plan de trabajo……………………………………………........14
2. Introduction………………………………………………………........15
2.1. Background……………………………………………....…...15
2.2. Objectives………………………………………………..……16
2.3. Workplan………………………………………………………16
3. Estado del arte…………………………………………………..…..17
3.1. Comunicación entre dispositivos…………………………….…..17
3.1.1. Bus I2C…………………………………………………….…17
3.1.2. SPI…………………………..…………………………..18
3.1.3. UART……………………..………………………....…..18
3.1.4. Conclusiones…………………….…………………........…..18
3.2. Spartan-3…………………………………..……………....…19
3.3. Raspberry Pi…………………………………..…………....…19
3.4. Módulos del Kernel……………………………..…………….…20
3.5. Tipos de módulos en el kernel……………………..………………20
3.6. Major / Minor number……………………………………….…..…21
3.7. Diferencias entre espacio de usuario y kernel….………….…….…21
3.8. Descripción VHDL…………………………………...........…22
4. Planificación……………………………………..…………………….…23
4.1. Identificación del problema………………………….........……23
4.2. Desarrollo de alternativas……………………………..........……23
4.3. Elección de la alternativa más conveniente……………..…………24
4.4. Ejecución del plan……………………………………….....……25
4.5. Toma de decisiones………………………………………………27
4.6. Planificación……………………………………………………..27
5. Diseño y requisitos.……………………………………..…………..28
5.1. Realización del módulo y funcionamiento…………..…………..…28
5.2. Espacio de usuario………………………………………….………29
5.2.1. Sensor de temperatura……………………………………….29
5.2.2. Programa en espacio de usuario…………………….…………31
5.3. Módulo en el kernel………………………………………...…32

5.3.1. Init……………………………………………………..…………33
5.3.2. Read…………………………………………………...…………34
5.3.3. ReceiveBit……………………………………….…………….35
5.3.4. Write…………………………………………..……………35
5.3.5. SendBit………………………………………..…………….35
5.4. Diseño VHDL.....…………………………………..……………36
5.4.1. Entity……………………………………………..……………36
5.4.2. Architecture Behavioral…………………………..…………...38
5.4.3. Mem_write……………………………………….……………38
5.4.4. Mem_load……………………………………….…………39
5.4.5. Mem_read…………………………………………………….…40
6. Resultados, discusión crítica y conclusiones…………………...….42
6.1. Resultados…………………………………………...……..42
6.2. Conclusiones…………………………………………..………43
6.3. Futuras líneas de investigación…………………..……..…..45
7. Results, critical discussion and conclusions…………………………………………………....……46
7.1. Results……………………………………………………...……46
7.2. Conclusion……………………………………………….………47
7.3. Future work………………………………………….....………48
8. Trabajo individual………………………………………..…………50
8.1. Roberto de la Cruz Martínez……………………..…………50
8.2. Juan Samper González……………………………...………52
Bibliografía………………………………………………………………55

Resumen

Hoy día vivimos en la sociedad de la tecnología, en la que la mayoría de las cosas cuentan con uno o varios procesadores y es necesario realizar cómputos para hacer más agradable la vida del ser humano. Esta necesidad nos ha brindado la posibilidad de asistir en la historia a un acontecimiento sin precedentes, en el que la cantidad de transistores era duplicada cada dos años, y con ello, mejorada la velocidad de cómputo (Moore, 1965). Tal acontecimiento nos ha llevado a la situación actual, en la que encontramos placas con la capacidad de los computadores de hace años, consumiendo muchísima menos energía y ocupando muchísimo menos espacio, aunque tales prestaciones quedan un poco escasas para lo que se requiere hoy día. De ahí surge la idea de comunicar placas que se complementan en aspectos en las que ambas se ven limitadas.

En nuestro proyecto desarrollaremos una interfaz *software/hardware* para facilitar la comunicación entre dos placas con distintas prestaciones, a saber, una Raspberry Pi modelo A 2012 y una FPGA Spartan XSA-3S1000 con placa extendida XStend Board V3.0. Dicha comunicación se basará en el envío y recepción de bits en serie, y será la Raspberry Pi quien marque las fases de la comunicación.

El proyecto se divide en dos partes:
La primera parte consiste en el desarrollo de un módulo para el *kernel* de Linux, que se encarga de gestionar las entradas y salidas de datos de la Raspberry Pi cuando se realizan las pertinentes llamadas de write o read. Mediante el control de los GPIO y la gestión de las distintas señales, se realiza la primera fase de la comunicación.
La segunda parte consiste en el desarrollo de un diseño en VHDL para la FPGA, mediante el cual se pueda gestionar la recepción, cómputo y posterior envío de bits, de forma que la Raspberry Pi pueda disponer de los datos una vez hayan sido calculados.

Ambas partes han sido desarrolladas bajo licencias libres (GPL) para que estén disponibles a cualquier persona interesada en el desarrollo y que deseen su reutilización.

Palabras Clave

Interfaz software/hardware,

Módulo del kernel,

Raspberry Pi,

FPGA Spartan 3,

Comunicación entre dispositivos,

Sensor de temperatura.

Abstract

Nowadays we live in society of technology, where most things have one or more processors and is necessary computations to make human life more enjoyable. This need has given us the opportunity to attend an unprecedented event in history, in which the number of transistors was doubling every 2 years, and thus improved computing speed (Moore, 1965). Such an event has led us to the current situation, in which we find boards with a computing power as high as the ones of first years, consuming far less power and occupying much less space. Although such benefits are a little short of what is required in our days. Hence arises the idea of communicating boards that complement in areas where both are limited.

In our project we will develop a software/hardware interface to facilitate communication between two boards with different features, namely a Raspberry Pi model A 2011 and a Spartan FPGA XSA-3S1000 board with extended XStend Board V3.0. This communication is based on sending and receiving serial bit, and the Raspberry Pi will set the phases of communication.

The project is divided into two parts. The first part is the development of a module for the Linux kernel, which manages the data inputs/outputs of the Raspberry Pi whenever write or read calls are made. By controlling the GPIO and management of the various signals, the first phase of the communication is done.

The second part is the development of a program in VHDL for FPGA, by which to manage the reception, computer and then sending bits, so that the Raspberry Pi can have the data once they have been calculated.

Both parts have been developed under free licenses (GPL). In this way we make it available to anyone who wants to reuse the project.

Keywords

Software/hardware interface,

Kernel module,

Raspberry Pi,

FPGA Spartan 3,

Device communication,

Temperature sensor.

1. Introducción

El hombre es un ser social y por ello es inevitable que tenga la necesidad de comunicarse. El desarrollo de las tecnologías viene guiado por la observación del mundo que nos rodea, y los comportamientos que observamos en los seres vivos. Así pues, características o cualidades que son beneficiosas en nuestra especie también lo será para la tecnología. Es el caso de la comunicación.

Podemos definir la comunicación como la actividad destinada a intercambiar información entre dos o más participantes (en este caso, dos placas: Raspberry Pi y Spartan 3) con el fin de transmitir o recibir significados (bits) a través de un sistema compartido de signos y normas semánticas. Los pasos básicos de la comunicación están compuestos por la formación de una intención de comunicar, la composición del mensaje, la codificación del mensaje, la transmisión de la señal, la recepción de la señal, la decodificación del mensaje y finalmente, la interpretación del mensaje por parte de un receptor.

Para llevar a cabo el desarrollo de este proyecto, diseñaremos una interfaz *software-hardware*, estableciendo un puente entre la máquina y las personas, permitiendo a la máquina entender nuestras instrucciones y a nosotros entender el código binario traducido a información legible.

1.1. Antecedentes

En los tiempos actuales observamos que uno de los objetivos principales de la tecnología es la comunicación. Constantemente se desarrollan aplicaciones, interfaces, en definitiva *software* y *hardware* para favorecer esta comunicación, ya sea entre personas, dispositivos o personas y dispositivos. Podríamos decir que Internet es su máximo exponente, favoreciendo la comunicación entre máquinas alrededor de todo el mundo, así como la de las personas que manejan dichos equipos.

Es necesario que la comunicación sea eficiente, con buenas tasas de transferencia de datos, y completa, con mensajes, canales, interpretaciones de los mensajes, emisores y receptores.

1.2. Objetivos

En vista de las necesidades de cómputo o de la motivación de unas mayores prestaciones para la Raspberry Pi, se enumeran los siguientes objetivos a cumplir:

- Investigación de otros sistemas que tengan la misma motivación o similar a la de este proyecto.
- Investigación sobre módulos del núcleo del sistema (kernel).
- Investigación sobre los protocolos de comunicación disponibles.
- Diseño de un protocolo eficiente de comunicación.
- Publicación del código en un repositorio o en la web.

1.3. Plan de trabajo

La organización de este trabajo ha consistido en reuniones periódicas en las que el tutor del proyecto y los alumnos se han puesto al día de los avances. El desarrollo general ha comprendido los siguientes campos:

- Documentación sobre los protocolos de comunicación.
- Documentación y preparación de la Raspberry Pi A y de la Spartan 3, así como de los distintos entornos para una correcta comunicación entre ambas placas.
- Documentación sobre *drivers* y módulos del kernel de GNU/Linux.
- Pruebas de comunicación entre componentes electrónicos/circuitos externos y la Raspberry Pi.
- Elaboración del diseño para la FPGA, del *driver* para el *kernel* y del programa en espacio de usuario.
- Elaboración de la memoria.

2. Introduction

Communication is an intrinsic characteristic of the animal world and therefore it is inevitable that all living things have a need to communicate. The observation of the world and behaviors we see in living things around us guides the development of new technologies. Thus, characteristics or qualities that are beneficial in our species will be beneficial for technology. This is the case of communication.

We can define communication as an activity designed to exchange information between two or more participants (in this case, two boards: Raspberry Pi and Spartan 3) in order to transmit or receive messages (bits) through a shared system of signs and semantic rules. The basic steps of communication are made by forming an intention to communicate, the composition of the message, the message encoding, transmitting the signal, the signal reception, decoding the message and finally the interpretation of the message by a receiver.

To carry out the development of this project, we will design a software-hardware interface, establishing a bridge between the machine and people, enabling the machine to understand our instructions and we understand the binary code translated into readable information.

2.1. Background

Nowadays we observe that one of the main objectives in technology is communication. Applications, interfaces, software and hardware are constantly being developed to facilitate this communication between people, devices or people and devices We could say that the Internet is its greatest exponent, encouraging communication between machines all around the world, and between the people who use them.

It is necessary an efficient communication, with good data transfer rates, and complete communication with messages, channels, interpretations of messages, senders and receivers.

2.2. Objectives

In view of the computing needs or the motivation for a higher performance of the Raspberry Pi, the following objectives are listed:

- Research other systems with the same motivation or similar to this project.
- Research kernel modules.
- Research the available communication protocols.
- Design of an efficient communication protocol.
- Publication of the code in a repository or web.

2.3. Workplan

Through regular meetings between the tutor and students they have marked the steps of this project. It has carried out documentation and development in the following fields:

- Documentation on communication protocols .
- Documentation and preparation of the Raspberry Pi A and Spartan 3, as well as the different environments for proper communication between the two boards.
- Documentation drivers and kernel modules GNU/Linux.
- Testing communication between electronic components/external circuits and Raspberry Pi.
- Development of design for FPGA, the Raspberry Pi kernel module and the user space program.
- Development of this document.

3. Estado del arte

En este apartado vamos a hablar sobre las diversas aplicaciones existentes a través de un módulo en el *kernel* de Linux para su diversa comunicación con otros dispositivos *hardware*.
Haremos referencia a cuál es el principal objetivo de crear un *driver* así como las ventajas y desventajas que esto conlleva.
Mostraremos también cuál ha sido el plan de trabajo que hemos llevado a cabo, las fases por las que hemos pasado y en qué aspectos hemos hecho más hincapié para su correcta realización.

3.1. Comunicación entre dispositivos

Antes de comenzar con la realización de nuestro diseño, estuvimos buscando información acerca de otros protocolos de comunicación entre dispositivos, para ver qué tipo de desarrollos se habían realizado, cuáles eran las principales ventajas y desventajas de éstos, ver qué podríamos mejorar en nuestra implementación, y también entender qué aspectos son los más importantes a la hora de mantener una comunicación entre dispositivos.

3.1.1. Bus I^2C

El primer protocolo en el que nos fijamos fue en el protocolo I^2C. I^2C, en inglés *Inter-Integrated Circuit,* es un protocolo que desarrolló Philips en la década de los 80, cuya principal motivación era la de favorecer la comunicación interna entre circuitos integrados. A grandes rasgos, el protocolo está diseñado como un bus maestro-esclavo. Cuenta con dos líneas de señal: una línea de reloj, generada por el maestro, y una línea de datos. Debido a que solo uno de los dos (maestro o esclavo) puede enviar datos, es un protocolo *half-duplex.* En su modo estándar, la tasa de transferencia es de 100 kbits/s. Cada circuito integrado tiene datos únicos, como por ejemplo un código o dirección que será necesario para establecer la comunicación y diferenciarlo de los demás circuitos. Aunque la idea principal es mantener comunicación entre circuitos integrados de una misma placa, el protocolo también permite la comunicación entre dispositivos que están en placas separadas, como por ejemplo, una Raspberry Pi y un sensor. Hemos observado que haciendo uso del protocolo de esta forma, baja el rendimiento, teniendo el cuenta que las largas longitudes de los cables pueden provocar atenuación de la señal.

3.1.2. SPI

Éste es otro protocolo en serie y síncrono, igual que el I^2C, pero más sencillo, ya que no cuenta con direcciones de identificación para cada circuito. Dispone de 3 líneas de señal, que son: una línea de reloj, generada por el master, una línea de envío de datos y otra línea de recepción de datos, lo que permite una comunicación full-duplex. Habrá que añadir una línea (SS, *Slave Select*) más por cada circuito que se conecte al master (ya que carecen de direcciones).

Al tener más líneas para envío de datos, se obtiene una mayor tasa de envío de bits, siendo el mayor número de líneas de señal necesarias el principal inconveniente.

3.1.3. UART

Por último, el protocolo UART es el protocolo más sencillo de los tres, ya que solo usa dos líneas de señal (una de envio de datos y otra de recepción de datos) y es asíncrono.

3.1.4. Conclusiones

Una vez estudiados estos tres protocolos, consideramos la opción de hacer que el nuestro fuese síncrono, con la intención de conseguir una alta tasa de transferencia de bits, usando una línea de señal más para el reloj, que sería generado por la Raspberry Pi. Habría un *master*, la Raspberry Pi y un *slave,* la FPGA. No serían necesarias las direcciones para identificar cada circuito, pues nuestro protocolo conecta de manera directa la Raspberry Pi con la FPGA, y añadiríamos una línea de dirección para determinar las fases de envío o recepción de bits.

3.2. Spartan-3

En nuestro proyecto hemos utilizado la FPGA Spartan 3 [5] debido a que es la placa que nos prestaban en los laboratorios de la Facultad de Informática.

La FPGA está formada por una memoria SDRAM de 32 Mbytes y 2 Mbytes de Flash. Además de esto, también tiene un puerto VGA que reproduce gráficos en 512 colores. La carga del diseño se realiza a través de la herramienta *XSTOOLS,* conectando la placa al ordenador a través de los puertos USB y cargando el .bit.

La placa proporciona 65 GPIO que están disponibles para realizar una interfaz de comunicación con dispositivos externos. En el caso de nuestra placa, viene insertada en una placa extendida XStend Board 3.0.

3.3. Raspberry Pi

Raspberry Pi es un ordenador de tamaño reducido y de bajo coste, desarrollado en Reino Unido por la Fundación Raspberry Pi [4], con el objetivo de estimular la enseñanza de ciencias de la computación en las escuelas.

El diseño incluye un SoC (System-on-a-chip) Broadcom BCM2835 [3], que contiene:

- un procesador central ARM1176JZF-S a 700 MHz
- un procesador gráfico VideoCore IV
- 512 MB de memoria RAM

Estas especificaciones corresponden a la Raspberry Pi modelo A.

Por otra parte, el diseño de la Raspberry Pi no incluye un disco duro ni unidad de estado sólido, ya que usa una tarjeta SD para el almacenamiento, tanto del sistema operativo como de los ficheros personales.
En la tarjeta se instalará la distribución GNU/Linux llamada Raspbian Jessie, derivada de Debian Jessie, que dispone de la adaptación y optimización necesarias para procesadores ARM.

El periférico que se utilizará para las conexiones con *hardware* externo será el GPIO, por ser de propósito general y disponer de varios pines configurables para establecer la comunicación entre ambas placas. En último lugar, únicamente mencionar que la Raspberry Pi utiliza 3.3V en los pines del GPIO.

3.4. Módulos del *Kernel*

Los módulos del *kernel* son fragmentos de código que pueden ser insertados y eliminados del núcleo del sistema acorde a nuestras necesidades. De esta manera se podrán agregar o quitar características de su núcleo en el momento.
Estos módulos pueden cargarse al arranque del sistema operativo editando el archivo /boot/config.txt. En este archivo hemos añadido la línea de configuración que permite la comunicación mediante el protocolo 1-Wire (*dtoverlay* = *w1-gpio*).

El núcleo del sistema está formado por un conjunto de módulos escritos en código C que se ejecutan de forma privilegiada, es decir que tienen acceso a todos los recursos del sistema. Tienen la extensión .ko.
Los módulos pueden ser insertados en el sistema sin necesidad de reiniciar el mismo. De esta forma, podremos cargar nuestro módulo en el *kernel* para poder controlar el *hardware* (los GPIO), sin tener que reiniciar la Raspberry Pi.

3.5. Tipos de módulos en el kernel

El objetivo principal de crear un módulo en el *kernel* de Linux es facilitar el manejo de los dispositivos de E/S. Estos módulos se pueden agrupar en tres grandes grupos:

- Dispositivos de interfaz de usuario
- Dispositivos de almacenamiento
- Dispositivos de comunicaciones.

En nuestro caso el que hemos tenido que realizar es un dispositivo de comunicaciones.
Los *drivers* pueden estar embebidos en el núcleo del *kernel* o pueden ser modulares. Los *drivers* embebidos se compilan al momento de generar el *kernel*. Los *drivers* modulares se compilan por separado del *kernel* y se instalan en el sistema por medio del comando:

```
pi@raspberrypi:~/TFG/Driver-Raspberry $ sudo insmod <archivo.ko>
```

o bien

```
pi@raspberrypi:~/TFG/Driver-Raspberry $ sudo modprobe <archivo.ko>
```

La diferencia reside en que modprobe requiere que el módulo esté, además de creado (.ko), alojado en una carpeta determinada por el sistema operativo.

Para eliminar un módulo se ejecuta el comando:

pi@raspberrypi:~/TFG/Driver-Raspberry $ sudo rmmod <archivo>

3.6. Major / Minor number

Cada *driver* tiene un número de dispositivo principal (*major*) que sirve para identificarlo. Si el *driver* sirve a varios dispositivos, cada dispositivo tiene un número de dispositivo secundario (*minor*) que lo identifica. Juntos, el número principal y el secundario especifican de forma única cada dispositivo de E/S. Puede utilizarse el comando mknod para la creación de archivos de dispositivos a través de la terminal de comandos, aunque nosotros lo crearemos en la función init del módulo del *kernel,* y lo borraremos al descargar dicho módulo, ya que es el único programa que debe tener acceso a dicho dispositivo. El comando que crea dicho dispositivo es:

pi@raspberrypi: $ sudo mknod [opciones] nombre tipo [major minor]

pi@raspberrypi:~/TFG/Driver-Raspberry $ sudo mknod /dev/gpio c 60 1

El argumento que sigue a nombre especifica el tipo de fichero a construir. En nuestro caso, tipo c, ya que será un fichero especial de caracteres sin búferes.

3.7. Diferencias entre espacio de usuario y kernel

El espacio del *kernel* se encarga de gestionar los recursos del *hardware* de una forma eficiente. Constituye un puente o interfase entre el programador de aplicaciones para el usuario final y el *hardware*. La misión del núcleo es dar acceso a los recursos del sistema a los usuarios cuando éstos lo demandan, aunque siempre bajo ciertas condiciones.

En el espacio de usuario utilizamos librerías del sistema. Estas librerías pueden llamar una o más veces al sistema y estas llamadas ejecutan la función de la librería que ha sido invocada en modo supervisor. Una vez la llamada al sistema completa su tarea, la ejecución se transfiere de nuevo a modo de usuario.

3.8. Descripción VHDL

VHDL es el acrónimo que representa la combinación entre VHSIC (*Very High Scale Integrated Circuit*) y HDL (*Hardware Description Language*).

Una FPGA está compuesta por un conjunto de celdas o bloques lógicos configurables. Estos bloques se pueden configurar como memoria (*FLIP-FLOP*), como multiplexor o como puertas lógicas *AND, OR, NOT*.

El diseño tendrá distintos procesos (en nuestro caso: emisión, recepción, generación de resultados, generación de señal de carga...) y cada proceso estará formado por bloques lógicos. Debido a que cada proceso es independiente, pueden estar operando todos al mismo tiempo, pudiendo conseguir velocidades mucho más altas de procesamiento.

4. Planificación

4.1. Identificación del problema

Las prestaciones de cómputo que ofrece la Raspberry Pi para tareas en las que se requiera un alto procesamiento llevan a buscar mejores opciones, como por ejemplo la potencia que ofrecen las FPGA, realizando procesamiento paralelo. Así que en vez de renunciar a la versatilidad de la Raspberry, decidimos llevar a cabo la realización de un protocolo de comunicación entre una Raspberry Pi, modelo A y una FPGA Spartan 6 (aunque finalmente fue una Spartan 3, accesible para nosotros a través del préstamo de material de los laboratorios de la Facultad de Informática), mediante el cual gestionamos el envío de datos de la Raspberry Pi a la FPGA, el cómputo de los mismos (en la FPGA), y el envío de datos de vuelta a la Raspberry Pi. Se decidió desde el primer momento que el controlador de la Raspberry Pi se iba a realizar en el espacio del *kernel*.

4.2. Desarrollo de alternativas

A grandes rasgos, el desarrollo de alternativas se centran en el número de vías que íbamos a usar para la comunicación. Primero tuvimos que considerar si el envío y recepción de datos se iba a realizar de manera síncrona o asíncrona, teniendo en cuenta que sería necesario usar una línea de señal (cable) más para controlar la señal de reloj en el caso de la comunicación síncrona. Además, eran necesarias señales para el envío y recepción de datos, dejando todavía por determinar cuántas íbamos a usar. Por último, surgía la necesidad de usar una señal más para marcar la dirección de la comunicación entre ambas placas. Con todas estas señales, nos surgieron las siguientes opciones:

1 - Una línea para la señal de reloj (en adelante clk) y una línea para la señal de envío y recepción de datos.

2 - Una línea para la señal del clk, una línea para señal de envío de datos y otra para la recepción de datos.

3 - Una señal para el clk, una línea de señal de envío de datos, una línea de señal de recepción de datos, y una línea de señal de dirección.

4 - Dos líneas de señales de clk (una para la Raspberry y otra para la FPGA), una línea de señal de envío de datos y otra de recepción de datos.

Especificamos que cuando hablamos de envío o recepción de datos, lo hacemos entendiendo que es desde la Raspberry Pi donde se llevan a cabo dichas acciones, siendo análogo si se hiciese desde la FPGA.

4.3. Elección de la alternativa más conveniente

Como se puede observar, al final todas las opciones incluían una línea para la señal de reloj, pues sin un pulso de reloj era más difícil identificar cuando se estaba enviando o recibiendo datos, o diferenciar de secuencias de muchos 1's ó 0's seguidos. Es decir, optamos por una comunicación en serie (bit a bit) y síncrona.

Finalmente, la alternativa escogida fue una línea para la señal del clk, una señal de envío de datos, una señal de recepción de datos, y una señal de dirección. El motivo por el cual hay dos señales, para el envío y recepción de datos, es porque en un primer momento se planteó la opción de hacer un protocolo de comunicación bidireccional (*full-duplex*). Debido a las posteriores decisiones, el diseño finalmente quedó fijado a una comunicación unidireccional (*half-duplex*). Bien es cierto que tanto la Raspberry Pi, a través de las funciones *gpio_direction_input* y *gpio_direction_output* puede fijar un GPIO de entrada o salida, y que la FPGA, en el diseño en VHDL puede fijar una señal como *inout* en su declaración, para que sea a la vez de entrada y de salida, pudiendo, de esta forma, reducir el número total de líneas a 3: una línea para la señal del clk, otra línea para la señal de dirección, y una última línea para la emisión y recepción de bits.

Pero mantuvimos las dos líneas por dos motivos: Primero, para mantener la posibilidad de realizar un nuevo diseño en una futura vía de desarrollo, que cumpla con una comunicación *full-duplex*, ya que el coste que supone el *hardware* adicional no es elevado (sería un cable más). Y segundo, porque el constante cambio de dirección del GPIO (fijarlo de entrada o de salida) supone una pérdida de eficiencia en la comunicación. En un proceso de comunicación breve, en el que la Raspberry Pi envíe un dato y la FPGA lo envíe de vuelta a la Raspberry Pi, evidentemente no se notaría esta pérdida de eficiencia, pero imaginando un caso peor, en el que se haga un millón de veces este proceso, la Raspberry Pi tendría que fijar un millón de veces el pin correspondiente de salida y otro millón de veces el pin correspondiente de entrada.

El resto de casos fueron descartados por diversos problemas o dificultades. Tanto el caso 1 y 2 carecen de la señal de dirección, que facilita mucho el diseño que controla la FPGA. Por uĺtimo, la opción 4 en la que la FPGA genera su propia señal de reloj, se vuelve inviable, ya que la frecuencia de reloj es mayor de lo que puede gestionar la Raspberry Pi.

4.4. Ejecución del plan

Los pasos seguidos han sido la adquisición, aprendizaje/familiarización y preparación de los elementos necesarios para el desarrollo del proyecto: protoboard, leds, resistencias, cables, Raspberry Pi, sensor…

Después tuvimos que comprender y documentarnos sobre las diferencias de los distintos espacios de ejecución de un programa: espacio de usuario y el espacio del *kernel*. Realizamos las primeras pruebas con la Raspberry Pi. Encender leds conectados a la protoboard mediante programas en Python o en C en espacio de usuario (fácil con el uso de librerías).

Acto seguido tuvimos una primera toma de contacto con *drivers* para el *kernel* del sistema operativo. Empezamos buscando documentación para la distribución de Linux, Ubuntu, donde a través del comando *make* compilamos el módulo del *kernel*. Una vez trasladados estos conocimientos a la distribución del SO de la Raspberry Pi (Raspbian), surgieron problemas, pues en la última actualización de dicho SO no se encontraban instaladas las librerías necesarias para compilar el módulo. Conseguimos instalarlas de distintos modos: El primero, descargando las librerías adecuadas y creando el enlace pertinente:

```
ln -s /lib/modules/3.12.28+/build
```

Y la segunda, a través del proyecto rpi-source, que como indican en la wiki del repositorio de github:

“rpi-source installs the kernel source used to build rpi-update kernels and the kernel on the Raspbian image. This makes it possible to build loadable kernel modules.” [7]

Una vez éramos capaces de compilar, y por consiguiente, de insertar los módulos, estudiamos cuál debía ser la estructura que debía seguir nuestro driver. Tras varias reuniones con nuestro tutor, comprendimos que necesitábamos un *driver* capaz de realizar escrituras en un fichero especial (un chardev, son los ficheros que representan los dispositivos físicos, en los que se puede escribir y leer datos) para escribir/mandar datos (gestionados a través de la función *write* del módulo) y leer/recibir datos del fichero (gestionados a través de la función read del módulo).

El próximo paso fue conseguir el envío de datos a través de uno de los GPIO, de momento a un LED, para comprobar que, efectivamente cuando se escribía en el fichero, se llamaba a la función *write* y enviaba una señal al LED. Esto nos llevó a mostrar la representación de la cadena de entrada en su forma binaria (1´s y 0´s mediante el desplazamiento del carácter).

Era el momento de empezar con la segunda parte: El diseño para la FPGA. Tuvimos que plantear un escenario, por lo que fijamos cúal iba a ser el flujo de funcionamiento del programa. Elegimos un sensor de temperatura (DS18B20, hablamos más adelante de él, en el apartado 5.2.1) que se conectaría a uno de los pines de la Raspberry Pi, con el cual mediríamos la temperatura de la habitación. Le pediríamos al usuario que introdujera una temperatura que deseara alcanzar. La Raspberry Pi formatearía los datos y los enviaría de forma adecuada a la FPGA, quien calcularía la diferencia entre ambas temperaturas. Por último, la FPGA mandaría los datos de vuelta a la Raspberry Pi, quien mostraría los resultados al usuario. Todo esto se haría a través de un programa en espacio de usuario (test_driver.c), encargado de gestionar las escrituras o lecturas al fichero de caracteres. Se entiende que esta explicación corresponde a una prueba de concepto. En una aplicación real, la FPGA haría algo más que una resta.

La función de recepción de datos (función *read* del módulo) se dejó para más adelante, para cuando tuviésemos datos de vuelta. Por ello empezamos a implementar el diseño VHDL para la FPGA. Es un diseño sencillo, que cuenta con 2 procesos principales para gestionar la entrada y salida del flujo de datos. Haciendo el desplazamiento de, por ejemplo, el número 19 representado como una cadena de caracteres obtenemos:

00110001 00111001

mientras que si está representado como un entero obtenemos:

00010011

En la siguiente etapa comenzamos con la creación del clk y reconocimiento de flancos. Las pruebas se hicieron con los botones que vienen en la placa extendida adjunta a la FPGA Spartan 3. También hicimos la configuración de los pines a través del fichero pin.ucf. Tuvimos que desactivar la restricción de que la señal del clk fuese generada por la FPGA, ya que el entorno de desarrollo te avisa mediante un mensaje que el rendimiento obtenido con una señal de reloj externa, no generada por la FPGA puede ser muy inferior.

Luego creamos el programa en C en espacio de usuario para hacer pruebas de comunicación, enviando datos de la Raspberry a través del GPIO, conectado a un pin del SLOT1 de la placa extendida de la FPGA.

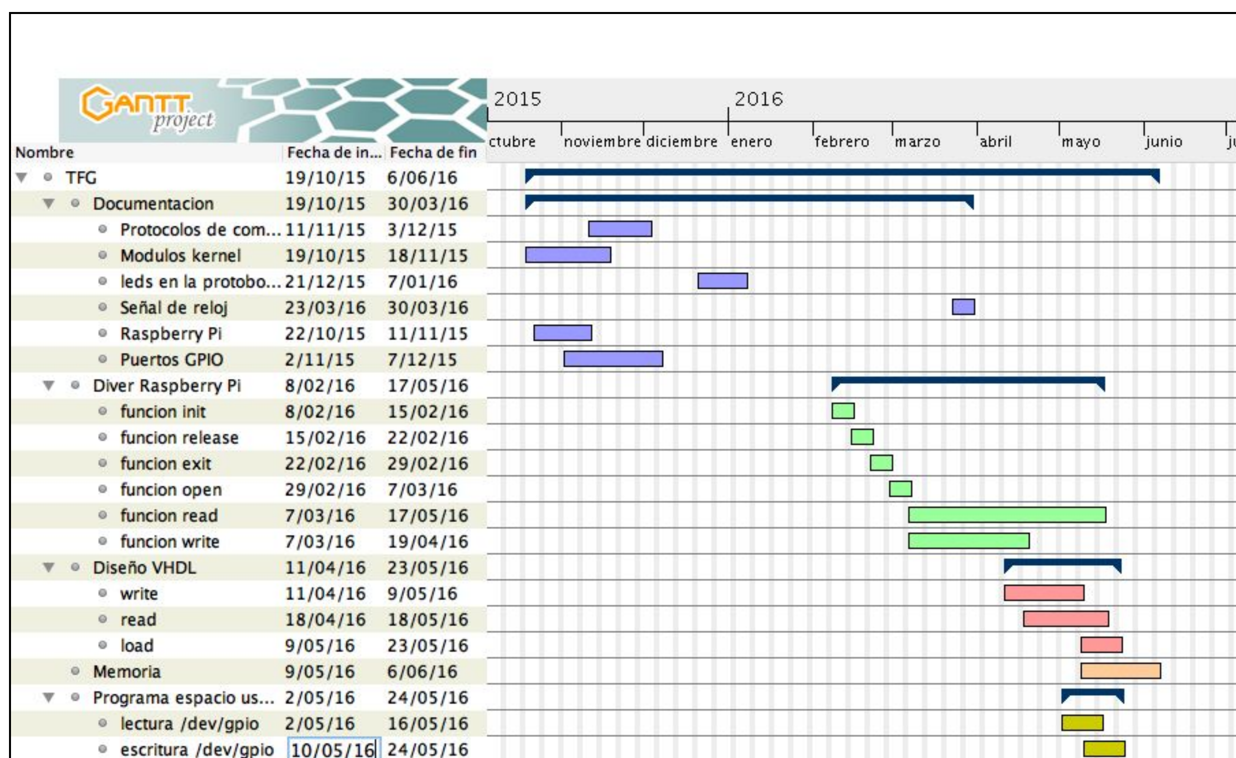
Una vez recibidos los 16 bits (dos números de 8 bits cada uno), procedemos a hacer la resta y a enviarlos de vuelta. Este proceso fue el más largo, ya que se juntaron varios errores y requirió de varias depuraciones. Se desarrolló el proceso de recepción de datos en la Raspberry Pi en paralelo. Por último se añadió el proceso de carga para realizar el envío de datos correctamente. Este proceso, mediante un flag de carga (que se activa cuando el contador ha alcanzado su tamaño máximo) dispone de un ciclo para cargar en paralelo todos los bits del registro donde se realiza la resta al registro de salida.

4.5. Toma de decisiones

Las decisiones se han llevado a cabo conjuntamente, siguiendo un proceso de meditación de ideas y/o soluciones. A los errores, por la falta de experiencia o desconocimiento en el campo, tomamos una postura unas veces reactiva, y en otras ocasiones de ensayo/error.

4.6. Planificación

A continuación en la figura 1 se muestra un diagrama de Gannt con el tiempo dedicado a las diferentes tareas llevadas a cabo a lo largo de nuestro desarrollo del proyecto.



Nombre	Fecha de in...	Fecha de fin
TFG	19/10/15	6/06/16
Documentacion	19/10/15	30/03/16
Protocolos de com...	11/11/15	3/12/15
Modulos kernel	19/10/15	18/11/15
leds en la protobo...	21/12/15	7/01/16
Señal de reloj	23/03/16	30/03/16
Raspberry Pi	22/10/15	11/11/15
Puertos GPIO	2/11/15	7/12/15
Diver Raspberry Pi	8/02/16	17/05/16
funcion init	8/02/16	15/02/16
funcion release	15/02/16	22/02/16
funcion exit	22/02/16	29/02/16
funcion open	29/02/16	7/03/16
funcion read	7/03/16	17/05/16
funcion write	7/03/16	19/04/16
Diseño VHDL	11/04/16	23/05/16
write	11/04/16	9/05/16
read	18/04/16	18/05/16
load	9/05/16	23/05/16
Memoria	9/05/16	6/06/16
Programa espacio us...	2/05/16	24/05/16
lectura /dev/gpio	2/05/16	16/05/16
escritura /dev/gpio	10/05/16	24/05/16

Figura 1- Planificación de trabajo

5. Diseño y requisitos.

5.1. Realización del módulo y funcionamiento

En la Raspberry hemos instalado el sistema operativo “Raspbian”, una distribución de “Debian” hecha a medida para este ordenador de bajo coste.

Es en esta distribución donde hemos incluido nuestro módulo en el *kernel*, el cual se encarga tanto de controlar la información que vamos a mandar a la FPGA, como de recibir la misma una vez haya sido computada por ésta.

Los módulos aportan dos ventajas fundamentales. Por un lado, la posibilidad de carga y descarga bajo demanda que disminuye los requerimientos de memoria (sólo necesitamos cargar los módulos correspondientes al *hardware* conectado), y nos liberan de la necesidad de reiniciar el sistema cada vez que se añade un dispositivo. Por otro lado, evitan que tengamos que reconstruir un nuevo *kernel* monolítico cada vez que queremos añadir el soporte para un dispositivo nuevo.

La comunicación entre ambos dispositivos se realiza a través de los puertos GPIO de la Raspberry Pi, y el SLOT 1 de la FPGA.

Las distintas líneas de intercambio de información que hemos tenido en cuenta son las siguientes:
- Data_in: Para recibir los datos de la FPGA en la Raspberry.
- Data_out: Para mandar datos de la Raspberry a la FPGA.
- Direction: Se encarga de indicar en qué se sentido se está estableciendo la comunicación.
- Clk: La señal de reloj generada por la Raspberry.

5.2. Espacio de usuario

5.2.1. Sensor de temperatura

Descripción.

El termómetro digital DS18B20 proporciona medidas de temperatura en grados Celsius de entre 9 y 12 bits y tiene una función de alarma con puntos superior e inferior programables por el usuario. El DS18B20 comunica a través de un bus 1-Wire (One-Wire), que por definición requiere una sola línea de datos (y tierra) para la comunicación con un microprocesador central. Tiene un rango de temperatura operativa de -55 ° C a + 125 ° C y tiene una precisión de ± 0,5 ° C en el rango de -10 ° C a + 85 ° C. Además, el DS18B20 puede derivar energía directamente de la línea de datos ("parasit-power"), eliminando la necesidad de una fuente de alimentación externa. [8]

Protocolo 1-Wire

1-Wire es un protocolo de comunicaciones en serie diseñado por Dallas Semiconductor. Está basado en un bus, un maestro y varios esclavos de una sola línea de datos en la que se alimentan. Por supuesto, necesita una referencia a tierra común a todos los dispositivos.

Configuración del sensor de temperatura:

El protocolo 1-Wire está soportado por Raspbian a través del módulo w1-gpio, que configura el pin del GPIO, y el w1-therm, que permite recuperar los datos del sensor a través de */sys/bus/w1/devices*. Se cargan de la siguiente forma:

```
pi@raspberrypi:/boot $ sudo nano config.txt
```

Añadir la siguiente línea al final del fichero:

```
# OneWire support
dtoverlay=w1-gpio
```

```
pi@raspberrypi:~/TFG/Driver-Raspberry $ sudo modprobe w1-gpio
pi@raspberrypi:~/TFG/Driver-Raspberry $ sudo modprobe w1-therm

pi@raspberrypi:~/TFG/Driver-Raspberry $ cd /sys/bus/w1/devices
pi@raspberrypi:~/TFG/Driver-Raspberry $ ls
pi@raspberrypi:~/TFG/Driver-Raspberry $ cd 28-xxxx
pi@raspberrypi:~/TFG/Driver-Raspberry $ cat w1_slave
```

Descripción del funcionamiento:

El termómetro se encuentra anclado a una protoboard. La disposición que hemos seguido para realizar la conexión se muestra en la figura 2, en el que la conectamos al GPIO 4 de la Raspberry.

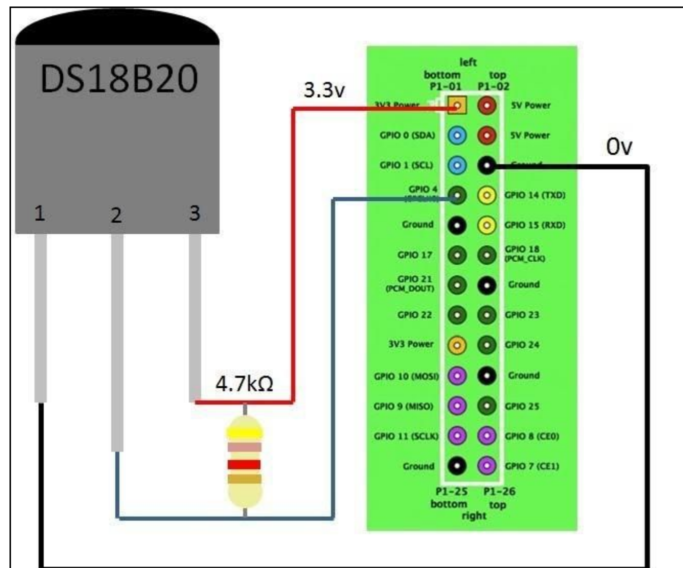


Figura 2: Conexión sensor temperatura

El sensor de temperatura devuelve lo siguiente:

87 01 4b 46 7f ff 09 10 48 : crc=48 YES
87 01 4b 46 7f ff 09 10 48 t=24437

En la función *thermometer()* nos quedamos con el valor “**24437**”, que es la temperatura actual (sin comas) que ha recogido el sensor. De estos 5 dígitos solo nos quedaremos con los 3 primeros. Los 2 primeros serán la temperatura, sin valor decimal, y el tercero (el primer decimal) lo usaremos para hacer más precisa la temperatura, redondeando a la siguiente unidad si el decimal es mayor o igual a 5.

5.2.2 Programa en espacio de usuario.

El método main de este programa se encarga de realizar los siguiente pasos:

1º - Invocar a la función *thermomether()* la cual, como hemos dicho anteriormente, se encarga de devolvernos la temperatura del ambiente.

2º - Abrir el fichero /dev/gpio, que es donde vamos a escribir y leer los datos de temperatura. En el caso de que se haya producido algún fallo devolverá un error que asignaremos a la variable *errno.*

3º - Mostrar por pantalla la temperatura actual.

4º - Solicitar la temperatura que queremos alcanzar.

5º - Leer la temperatura introducida en el rango [0-99]ºC.

6º - Concatenar temperatura_deseada + temperatura_ambiente

7º - Enviar los datos a la FPGA a través de una escritura al fichero de carácter (*write)*

8º - Realizar una lectura (*read*) una vez computados los datos por la FPGA.

9º- Mostrar por pantalla el resultado de la resta de ambas temperaturas (temperatura_deseada - temperatura_ambiente). Este resultado es fruto de los cálculos realizados por la FPGA.

En la figura 3 se puede apreciar cómo se lleva a cabo la ejecución del programa de espacio de usuario. Este programa debe ejecutarse con permisos de superusuario.

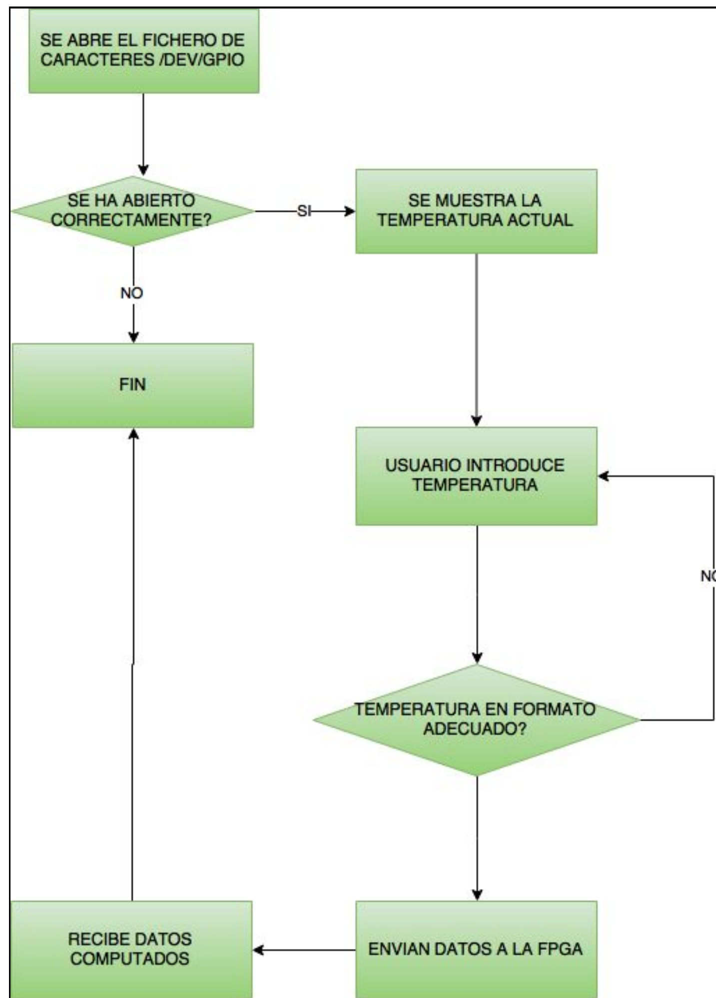


Figura 3 - Diagrama de flujo ejecucion espacio usuario.

5.3. Módulo en el kernel

Al principio definimos las direcciones de memoria correspondientes para acceder a los puertos GPIO según se establece en el manual BCM2835 ARM Peripherals [3]:

```
#define INP_GPIO_04(b)   *(volatile unsigned int *)(0x00000000+b) &=
(unsigned int)0xFFFF8FFF
```

(Definimos el GPIO 04 para utilizarlo como puerto de entrada).

Determinar las posiciones de memoria lógicas que corresponden a cada puerto GPIO implica llenar el código de macros para poder fijar una dirección de *input* o *output*, asignarle un valor, limpiar el pin... Para una mayor limpieza, organización y mantenimiento del código introdujimos una librería que facilita y optimiza el uso de los puertos GPIO. Esta librería es <linux/gpio.h>.

Cada puerto GPIO es una sola señal eléctrica que la CPU puede configurar a uno de dos valores - cero (0 V) o uno (3.3V), naturalmente.

Las funciones que nos facilitan su uso correcto son las siguientes:

- int *gpio_request (unsigned int gpio, const char *label)*; //asignación del gpio antes de su uso.
- void gpio_free (unsigned int gpio);
- int *gpio_direction_input (unsigned int gpio);* //definir GPIO de entrada.
- int *gpio_direction_output (unsigned int gpio, int value)*; //definir GPIO de salida.
- int *gpio_get_value (unsigned int gpio);* //devuelve el valor que tiene el gpio.
- void *gpio_set_value (unsigned int gpio, int value);* //asignas el valor "*value*" al gpio.

Hemos empleado un patrón de diseño de maestro-esclavo en el que el maestro es la Raspberry, generando de esta manera la señal de reloj para su correcta sincronización con su esclavo, la FPGA.

5.3.1. Init

Se encarga de crear el fichero de caracteres que se encuentra en la ruta */dev* y que le hemos asignado el nombre *gpio*. También inicializa y define cuáles son los puertos que van a ser utilizados de entrada y cuáles de salida.

Para crear el fichero de caracteres hemos seguido los siguientes pasos

1 - Invocamos a la función *alloc_chrdev_region()* para obtener un *major number* y un rango de *minor numbers.*

2 - Crea la clase device para nuestros dispositivos con la llamada a la función *class_create().*

3 - Invocamos a *cdev_init()* y *cdev_add()* para agregar el dispositivo de caracteres al sistema.

4 - Invocamos a la función device_create() que se encarga de crear un nodo de dispositivo.

En el caso de que se produzca un error en la creación del dispositivo entonces se mostrará un error en el fichero /var/log/kern.log.

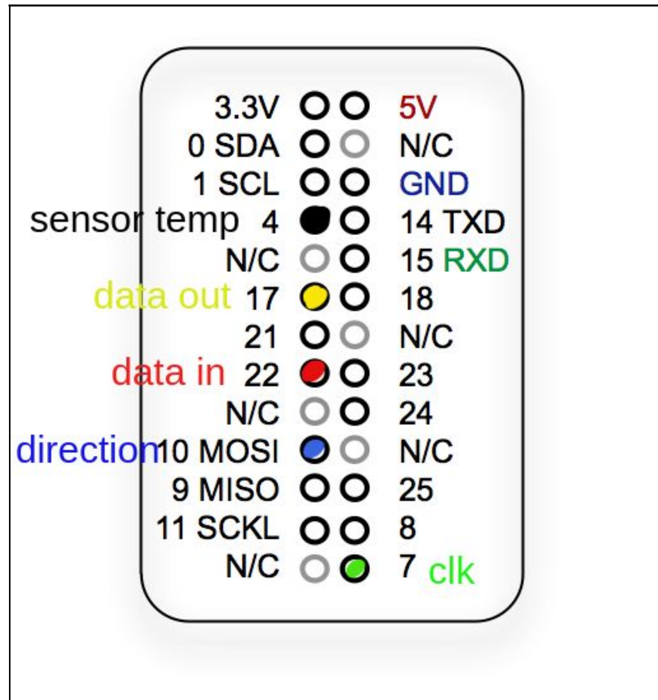


Figura 4: Puertos GPIO Raspberry

En la Figura 4 vemos los pines del GPIO que hemos usado, cada uno de ellos conectado al pin correspondiente del SLOT1 de la placa Spartan 3.

5.3.2. Read

Esta función *read* se encarga de recibir los datos de la FPGA una vez que hayan sido computados. Generamos un flanco de reloj para que se pueda reconocer la cantidad exacta de bits que se han recibido.

Se encarga de almacenar en un buffer los datos que va recibiendo. Dichos datos se reciben del más significativo a menos significativo, guardándolos en el array de forma creciente, es decir, el bit más significativo lo guardamos en la posición 0 del array y el menos significativo lo guardamos en la posición *n* del array.

El valor que recibe está contenido en 8 bits, de ahí a que el tamaño del *buffer* tenga un tamaño ya definido de 8. El bit más significativo se encarga de definir el signo de dicho número. Si este valor tiene el valor ‘1’ entonces el número recibido va a tener un valor negativo por lo que vamos a tener que realizar su complemento a dos.

5.3.3. ReceiveBit

Esta función se encarga de gestionar la recepción de datos de la FPGA una vez han sido computados los operandos. Los bits se reciben de más a menos significativo.

*num += pow(2,(7-i))*buffer[i];*

Devuelve el número *num* una vez se haya completado la recepción de bits.

5.3.4. Write

Esta función *write* se encarga de mandar el mensaje, que es un valor formado por la concatenación de la temperatura que hay en el ambiente junto con la temperatura que deseamos alcanzar. En el puntero *buf tenemos la temperatura deseada concatenada con la temperatura ambiente. La función se encarga de guardar en el operando op1 la temperatura deseada, y en el operando op2 la temperatura ambiente. Acto seguido, llama a la función *sendBit*, con los operandos op1 y op2 como parámetros.

5.3.5. SendBit

Esta función se encarga de generar un ciclo de reloj. Cada ciclo de reloj transmite un bit a través del GPIO correspondiente, siendo los 8 primeros bits el operando op2 y los 8 restantes el operando op1.

5.4. Diseño VHDL

En este diseño hemos tenido en cuenta los GPIO utilizados en la Raspberry Pi para declarar los puertos de la *entity.* Las líneas de dirección que hemos definido como salida en la Raspberry Pi las definimos de entrada, y de manera análoga con el resto de señales.

Los procesos que hemos diseñado son análogos a los programados en el driver de la Raspberry Pi. El proceso *mem_write* diseñado en VHDL en la FPGA es el equivalente a la función *gpio_read* programada en C en la Raspberry Pi en cuanto a la funcionalidad. Por otra parte, el proceso *mem_read* es el equivalente a la función *gpio_write* programada en C en la Raspberry Pi en cuanto a funcionalidad se refiere.

5.4.1. Entity

Es la descripción de la interfaz entre un diseño y su ambiente externo.

Aquí es donde definimos tanto las señales externas que vamos a obtener de la Raspberry Pi, como las señales que vamos a enviar a la Raspberry Pi. En la figura 6 encontramos cuáles son los pines asignados a cada señal.

Hay que tener en cuenta que el *dataOut* de la FPGA es el *dataIn* de la Raspberry Pi. La señal de reloj es generada en todo momento por la Raspberry por lo que los procesos de la FPGA se llevarán a cabo cuando reciba la señal de reloj.

En la figura 5 se muestra la entity del diseño VHDL.

```
entity communication_protocol is
 port (
   clk_rbpi : in std_logic;
         direction : in std_logic;
         dataIn : in std_logic;
         dataOut : out std_logic);
end communication_protocol;
```

Figura 5: Entity

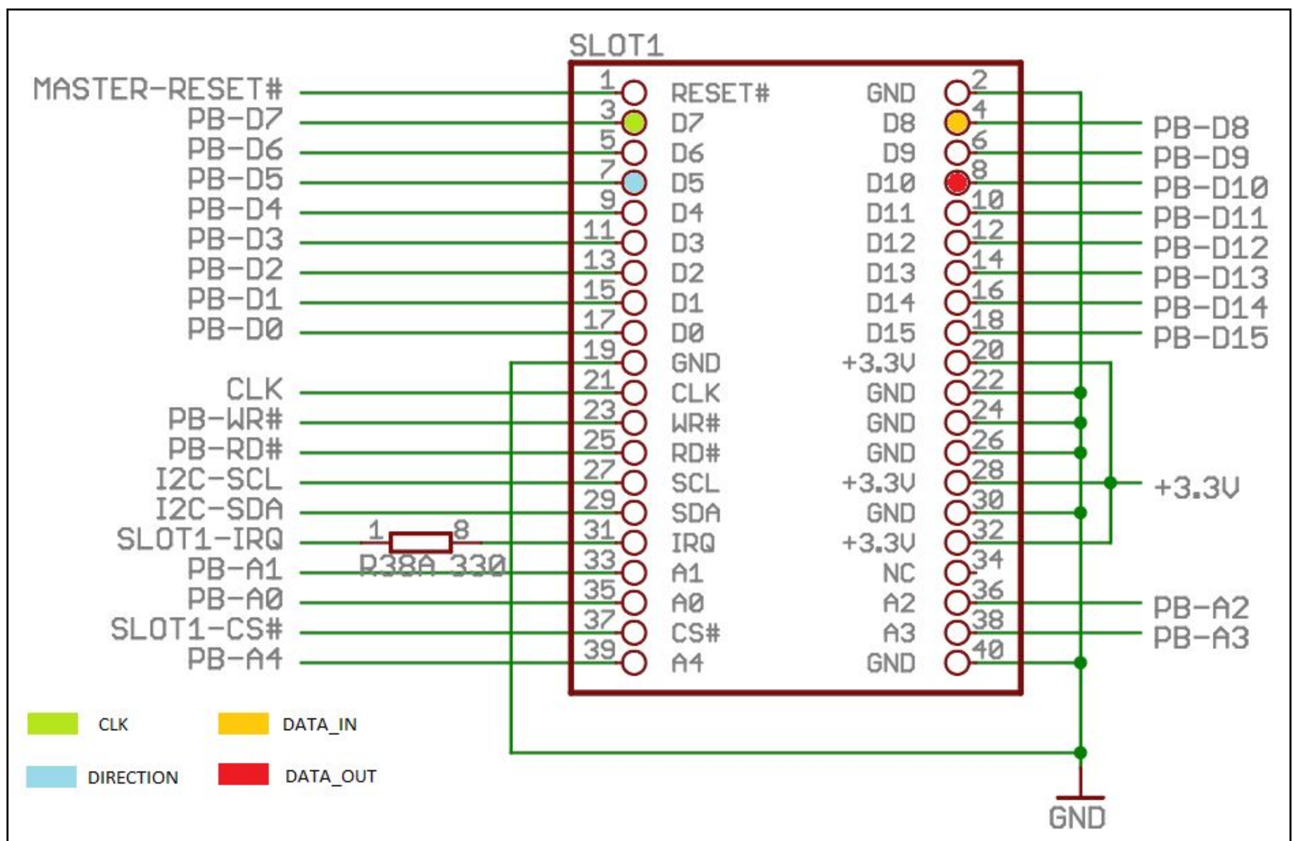


Figura 6 - Configuración de los pines de la spartan 3

5.4.2. Architecture Behavioral

Dentro del comportamiento de la arquitectura se van a declarar las señales y a realizar las operaciones combinacionales.

5.4.3. Mem_write

Este proceso se ejecutará cada vez que haya generado una señal de reloj la Raspberry.

En el caso que la señal *direction* esté activada, vamos almacenando en *buffer_data* los bits recibidos, desplazando a la derecha según vayan entrando datos. Este desplazamiento se irá haciendo hasta que se hayan recibido los 16 bits que forman el valor: *temperatura_deseada* + *temperatura_ambiente*.

En la figura 7 se puede apreciar cómo se lleva a cabo la ejecución del proceso.

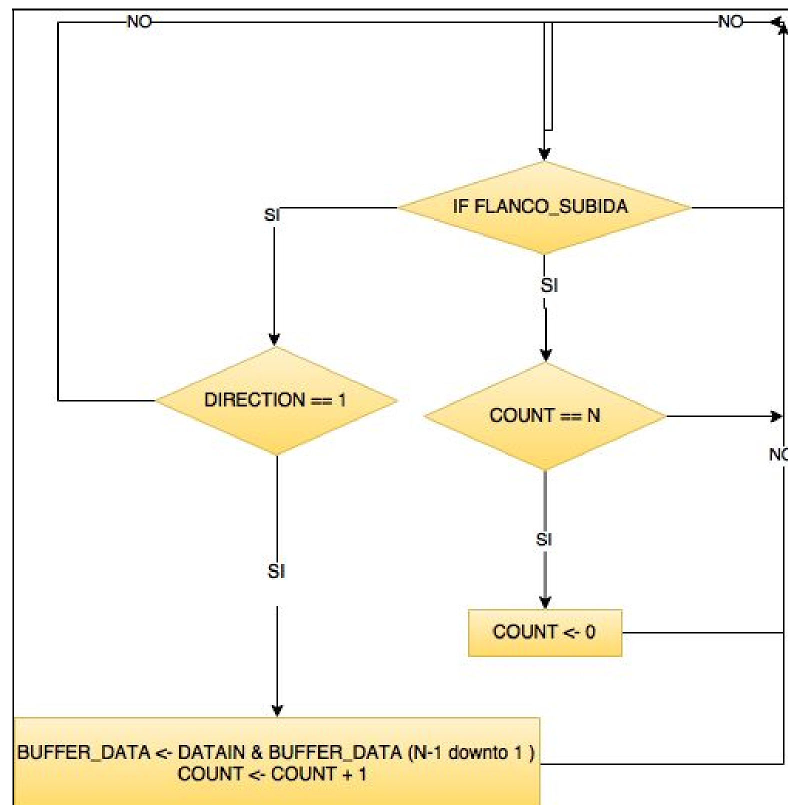


Figura 7 - Diagrama de flujo proceso write VHDL

5.4.4. Mem_load

El load <= ‘1’ activa durante un ciclo la señal de carga para que en el registro *resultado* se guarde la resta entre la temperatura deseada y temperatura ambiente.

En la figura 8 se puede apreciar cómo se lleva a cabo la ejecución del proceso.

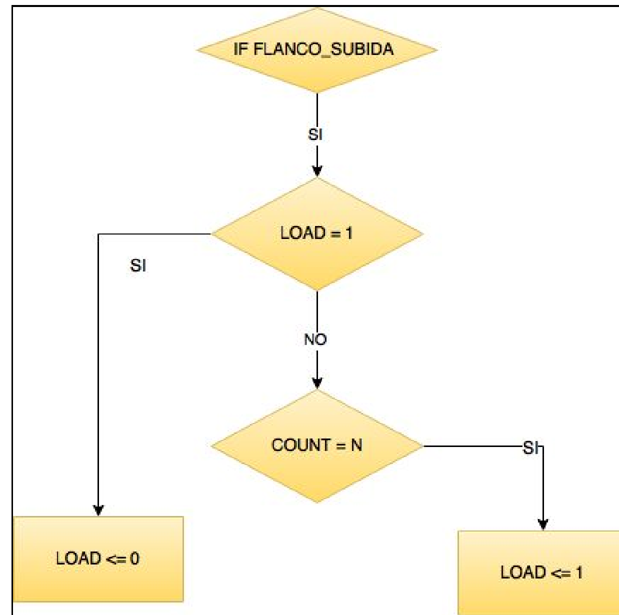


Figura 8 - Diagrama de flujo proceso load VHDL

5.4.5. Mem_READ

Desplaza a la izquierda los bits del vector resultado para que la siguiente instrucción combinacional envíe a través del *dataOut* los bits más significativos. En la figura 9 podemos ver de forma visual cómo se realiza este desplazamiento.

resultado <= resultado(6 downto 0)&'0';
dataOut <= resultado(7);

En la figura 10 se puede apreciar cómo se lleva a cabo la ejecución del proceso.

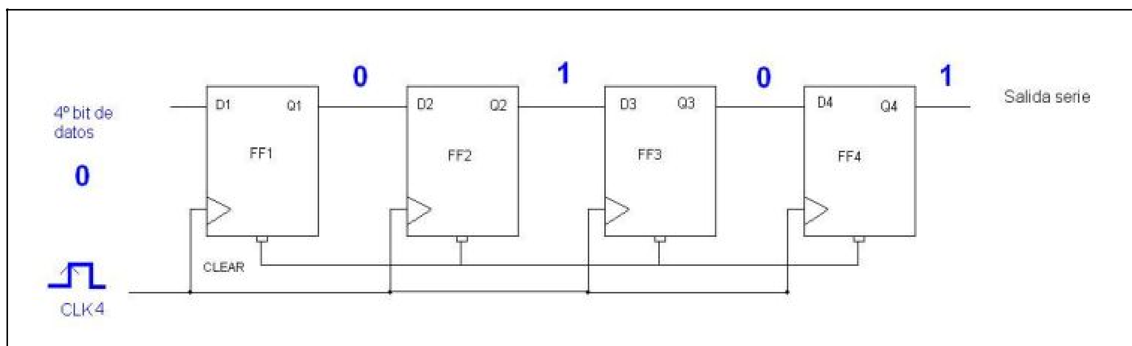


Figura 9: Registros desplazamiento

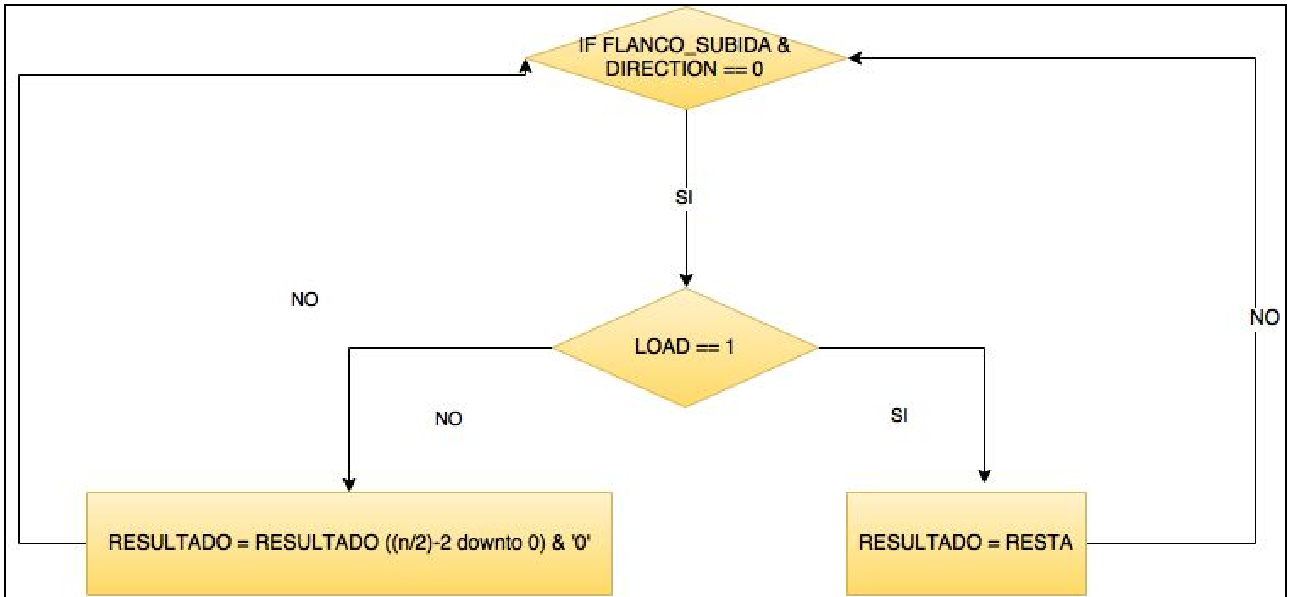


Figura 10 - Diagrama de flujo proceso read VHDL

6. Resultados, discusión crítica y conclusiones

6.1. Resultados

Una vez finalizado el protocolo de comunicación realizamos un banco de pruebas para poder observar la tasa de transferencia que alcanzamos.

Para poder observar esto, necesitamos ver un ejemplo “de verdad”, en el que el tamaño de datos a transmitir fuese mucho mayor de lo que habíamos hecho hasta ahora. De manera habitual, cada vez que enviamos datos a la FPGA invocando a la función *write* transferimos 16 bits (2 operandos de 8 bits). En este nuevo escenario, pusimos un bucle *for* que realizaba 100.000 iteraciones, es decir, 100.000 llamadas a la función *write.* También añadimos 2 variables de tipo clock (principio y final), que, una vez inicializadas de la forma *principio = clock(),* almacenan el número de ciclos de reloj que han pasado desde que el programa fue lanzado. De esta forma pudimos ver cuanto tardaba el bucle *for* en terminar, inicializando una variable al comienzo, y otra al final. Una vez finalizado el programa, obteníamos los siguientes resultados:

principio = 30.069
final=1.893.665

ciclos_totales = (final - principio) = 1.863.596 ciclos

Para interpretar el resultado de la llamada a clock(), es necesario conocer el valor de CLOCKS_PER_SEC (10^6 para POSIX). De esta forma sabemos el número de segundos que empleó nuestro protocolo en realizar las 100.000 iteraciones:

tiempo_segundos = ciclos_totales / ciclos_por_segundo = 1.863.596/1.000.000 = 1,86 segundos

Para terminar, como hemos dicho al principio, en cada escritura se enviaban 16 bits, por lo tanto, en 100.000 iteraciones se envían 1.600.000 bits. Teniendo el número de bits que hemos enviado, y el tiempo que hemos tardado en enviarlos, podemos mostrar nuestra tasa de transferencia:

*tasa de transferencia = (numero_bits * numero_iteraciones) / tiempo_segundos = 1.600.000/1,86 = 860.215*

Es decir, nuestra tasa de transferencia es de 860 kbps.

El otro estudio que hemos llevado a cabo, ha sido ver como afectaba el disponer de un solo gpio para enviar y recibir datos, con el correspondiente cambio de dirección de esta línea de señal. En el caso arriba descrito, con 2 líneas (1 para el envío y otra para la recepción) implica hacer una única fijación de la dirección al comienzo el programa.

Con una sola vía tendremos que fijar la dirección del *gpio* en cada iteración del ejemplo arriba descrito. Tras hacer los pertinentes cambios, obtenemos estos nuevos valores:

principio = 39.173
final = 2.193.941

ciclos_totales = (final - principio) = 2.154.768 ciclos
tiempo_segundos = ciclos_totales / ciclos_por_segundo = 2,15 segundos

Observamos que usando 2 vías obtenemos una mejora cercana al 16%, debido a que solo es necesario fijar la dirección del cable una única vez. Si teniendo un único cable se hiciese una única escritura, apenas apreciaríamos esta mejora, pues la dirección del cable solo se fijaría una vez.

6.2. Conclusiones

En este apartado describimos las motivaciones iniciales del proyecto y las conclusiones a las que hemos llegado.

Este proyecto nos ha permitido investigar las diferentes formas de comunicación entre dispositivos, la velocidad a la que estos transmiten sus datos y cuales son los protocolos para el intercambio de información más eficientes actualmente.

Como bien se ha descrito a lo largo del documento, las placas elegidas para el proyecto han sido una *Raspberry Pi*, modelo A y una FPGA *Spartan 3*. La elección del modelo de Raspberry se debe a dos factores: el primero, que ya disponíamos de esta placa. Y el segundo, que dentro de los distintos modelos que hay en el mercado, el modelo A es el segundo que menos energía consume, haciendo este modelo ideal para proyectos en los que la placa se alimente a través de una batería.

Para crear un driver para el *Kernel* de Linux; que en este proyecto lo hemos hecho a través de la *Raspberry Pi* con su sistema Operativo Raspbian, basado en Debian; son necesarios conocimientos sobre sistemas UNIX, los cuales son bastante básicos por nuestra parte en cuanto a su kernel se refiere. Nunca nos habíamos enfrentado a un problema de inserción de módulos en el núcleo del sistema y esto nos causaba interés.

Empezamos investigando sobre las principales ventajas de crear un programa en el núcleo del sistema frente a crearlo en el espacio de usuario. Nos encontramos con que la principal ventaja que nos ofrece esta primera opción es su versatilidad y acoplamiento en otros Sistemas Operativos. Estos módulos deben llevar una licencia GPL para que cualquier error que sea experimentado pueda ser corregido por aquellos que deseen realizar su mantenimiento.

Nos encontramos con que a partir de la versión 2.2 del *Kernel* se integraron las macros _init y _exit. Estas funciones dentro del espacio de kernel son las correspondientes a las del espacio de usuario, insmod y rmmod.

El manejo de los puertos GPIO de la Raspberry Pi nos abre un horizonte de posibilidades en cuanto a la comunicación con otros dispositivos. Su manejo es tan simple como el activar o desactivar la señal eléctrica en la posición de memoria reservada al puerto correspondiente.

El hecho de comunicar ambas placas sacando el mejor rendimiento de cada una de ellas también era una idea que nos creaba curiosidad. Aunque en este proyecto lo que nos centramos principalmente es en el intercambio de información entre la Raspberry Pi y la placa Spartan 3, de la familia Xilinx, lo realmente interesante es que la Spartan 3 pueda realizar operaciones de cálculo mucho más complejas en un menor tiempo de lo que lo pueda realizar la Raspberry Pi. En nuestro proyecto lo que hemos realizado es una resta entre dos operandos que recibe de la Raspberry Pi.

La idea principal era obtener información del medio ambiente a través de un sensor para enviar datos reales a la FPGA. Tras investigar y estudiar sobre el tema, decidimos utilizar un sensor que midiese la temperatura ambiente donde se encuentra la Raspberry Pi. Este tipo de sensor nos pareció práctico y visible. En la figura 11 se muestra la conexión entre la Raspberry Pi y la Spartan 3.

Por último, mostramos nuestra tasa de transferencia, alcanzando los 860 kbps, superando así las tasas del protocolo I^2C tanto en su modo estándar (100 kbps) como en su modo rápido (400 kbps), que era una de las motivaciones iniciales.

El propósito de nuestro proyecto, que es la transferencia de información entre dos placas, se puede llevar a cabo de diferentes maneras, por lo que damos la posibilidad de modificarlo incluyendo el código y esta memoria en un repositorio de Github, accesible a través del siguiente enlace: https://github.com/saeN13/TFG_Protocolo_Comunicacion

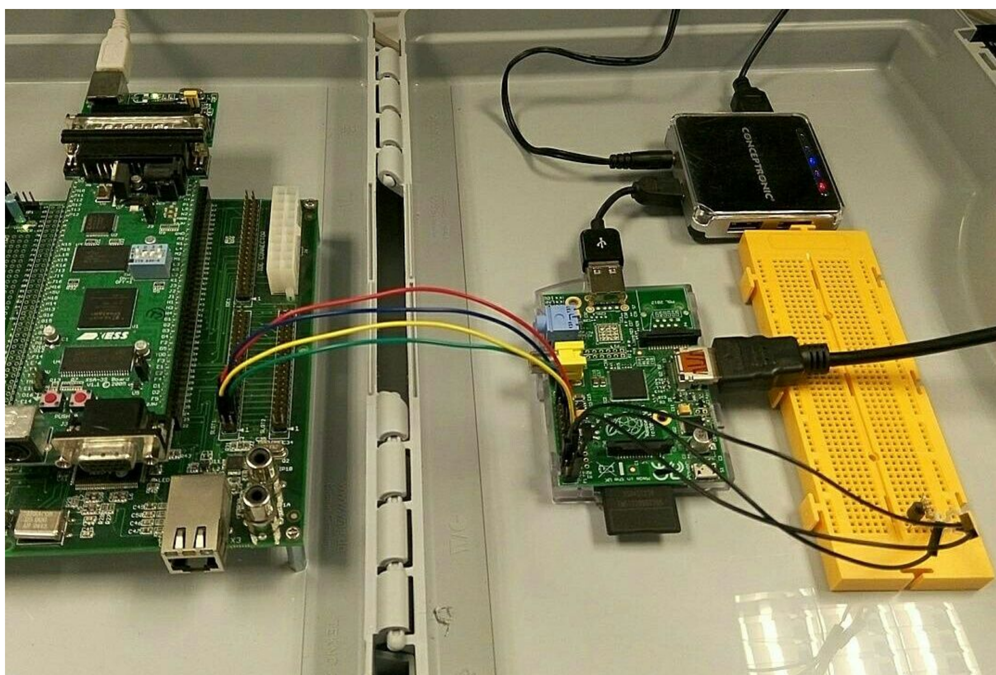


Figura 11: Conexión entre Raspberry Pi y Spartan 3

6.3. Futuras líneas de investigación

En vista del resultado obtenido al finalizar el desarrollo del proyecto, cabe replantearse la posibilidad de futuras iteraciones que añadan nuevas características. Dejamos abierta la posibilidad de una futura investigación más amplia.

En el caso de poder continuar con su desarrollo, sugerimos posibles modificaciones y ampliaciones.

Comunicación en paralelo. Actualmente nuestro proyecto transmite información a través de un único puerto GPIO, siendo esta en serie. Una posible mejora sería transmitir dicha información en paralelo, por ejemplo de byte en byte utilizando 8 puertos GPIO.

Comunicación full duplex entre la Raspberry Pi y la Spartan 3. En el proyecto actual la que se encarga tanto de iniciar como establecer la comunicación es la Raspberry Pi, generando la señal de reloj y activando o desactivando el GPIO asociado a la dirección. Una de las mejoras o ampliaciones de este proyecto sería que la Spartan 3 pudiese recibir los datos a la vez que está enviando el resultado de la operación anterior.

Envío y recepción de datos de tamaño n. Actualmente solo podemos enviar números que estén en un intervalo de valores entre 0 y 99, restringiendo temperaturas que superen los 3 dígitos. Una futura mejora sería eliminar esta restricción, dejando libertad al usuario de introducir la temperatura que desee.

Interfaz gráfica. Las interacciones que tiene que realizar el usuario para manejar dicho programa son a través de la línea de comandos, dando solamente la posibilidad de utilizar nuestro programa a usuarios con un mínimo conocimiento de informática. En el caso de mejorar dicha aplicación, se crearía una interfaz gráfica para facilitar su usabilidad y ofrecer un mejor aspecto.

7. Results, critical discussion and conclusions

7.1. Results

Once the communication protocol was completed, we made a test to observe the transfer rate we achieve.

To observe this, we need a "real" example, in which the size of data to be transmitted was much larger than we had done so far. Regular basis, every time we send data to the FPGA invoking the write function we transfer 16 bits (two 8-bit operands). In this new scenario, we put a for loop, that performed 100,000 iterations, 100,000 calls to the write function. We also added two variables of type clock (principio and final), that once initialized the form principio = clock (), store the number of clock cycles that have passed since the program was launched. In this way we could see how long it took to complete the loop, initializing a variable at the beginning, and at the end. Once the program is completed, we obtained the following results:

principio = 30.069
final=1.893.665

ciclos_totales = (final - principio) = 1.863.596 ciclos

Being one million the number of cycles per second we got on the Raspberry Pi, we know the number of seconds that our protocol has used to perform the 100,000 iterations:

tiempo_segundos = ciclos_totales / ciclos_por_segundo = 1,86 segundos

Finally, as we said at the beginning, each write sent 16 bits, therefore, 1,600,000 bits were sent in 100,000 iterations. Given the number of bits that we have sent, and the time we took to send them, we can show our transfer rate:

*tasa de transferencia = (numero_bits * numero_iteraciones) / tiempo_segundos*
= 1.600.000/1,86 = 860.215

That is, our transfer rate is 860 kbps.

The other study that we have conducted has been to see how affected having just one line for sending and receiving data. In the case described above, with 2 lines (one

for sending and one for receiving) involves setting the line just one time, at the beginning of the program. With one line we have to set the direction of gpio in each iteration of the example described above. After making the necessary changes, we get these new values:

principio = 39.173
final = 2.193.941

ciclos_totales = (final - principio) = 2.154.768 ciclos
tiempo_segundos = ciclos_totales / ciclos_por_segundo = 2,15 segundos

We note that using 2-way provides an improvement close to 16 %, noting that it is experienced in cases where a large number of messages transmitted (such as the 100,000 we put into the test case) but short ones (like our messages 16 bits).

7.2. Conclusion

In this section we describe the initial motivations of the project and the conclusions we have reached.

This project has allowed us to investigate different forms of communication between devices, the speed at which they transmit their data and what are the protocols for the exchange of information more efficient.

As has been described throughout this document, the chosen boards for the project have been a Raspberry Pi, model and FPGA Spartan 3. The choice of Raspberry model is due to two factors: first, that we already had this board. And the second, that within the different models on the market, is the second model that consumes less power, making this model ideal for projects where the plate is fed through a battery.

To create a driver for the Linux kernel; that in this project we have done through the Raspberry Pi with its operating system Raspbian, based on Debian; They are necessary knowledge of UNIX systems, which are pretty basic on our part as to its kernel is concerned. We had never faced a problem of insertion of modules in the system kernel and this caused us concern.

We started researching the main advantages of creating a program in the kernel of the system instead at the user space. We find that the main advantage offered by this first option is its versatility and engaging in other OSes. These modules should bear a GPL license for any errors that can be corrected is experienced by those who wish to maintain it.
We find that from kernel version 2.2 macros and _exit _init were integrated. These

functions within the kernel space are relevant to user space, insmod and rmmod.

The management of GPIO ports in Raspberry Pi opens a wide variety of possibilities for communication with other devices. Its handling is as simple as turn on or off the electrical signal in the memory location reserved for the corresponding port.

Although this project is focused on the exchange of information between the Raspberry Pi and the Spartan 3 board, what is really interesting is that the Spartan 3 can perform much more complex operations in less time of what Raspberry Pi could do. In our project we have done a subtraction between two floating point operands received from the Raspberry Pi.

Other idea idea was getting environmental information via a sensor for sending actual data to the FPGA. After research and study on the subject, we decided to use a sensor that would measure the ambient temperature. We found this type of sensor practical and visible.

Finally, we show our transfer rate, reaching 860 kbps, surpassing the I2C protocol rates both in standard mode (100 kbps) and its fast mode (400 kbps), which was one of the initial motivations.

The purpose of our project, which is the transfer of information between two boards, can be carried out in different ways, so we give the possibility to modify it including this code and this memory in a Github repository, accessible through the following link: https://github.com/saeN13/TFG_Protocolo_Comunicacion

7.3. Future work

In view of the result obtained at the end of the project, it should reconsider the possibility of future iterations that add new features. We leave open the possibility of future wider investigation.

In the case of continuing its development, we suggest possible modifications and extensions.

Parallel communication. Currently our project transmits information through a single port GPIO, being this serial. One possible improvement would transmit this information in parallel, for example byte by byte, using 8 GPIO ports.

Full duplex communication between the Raspberry Pi and Spartan 3. In the current project which handles both start and establish communication is the Raspberry Pi, generating the clock signal and activating or deactivating the associated GPIO direction. One of the improvements or extensions of this project would be the Spartan 3 could receive data while being sent the result of the previous operation.

Sending and receiving data of size n. Currently we can only send numbers within a

range of values between 0 and 99, restricting temperatures exceeding 3 digits. A further improvement would be to eliminate this restriction, leaving freedom to the user to enter the desired temperature.

Graphic interface. The interactions that the user has to do to operate the program are through the command line, giving only the possibility of using our software users with minimal computer knowledge. In the case of improving the application, a graphical interface would be created to facilitate usability and offer a better look.

8. Trabajo individual

En este capítulo se expone la aportación de cada miembro del grupo al proyecto.

8.1. Roberto de la Cruz Martínez

Comenzamos el proyecto a principios del mes de octubre teniendo una reunión con nuestro tutor Juan Carlos Fabero en la que nos indicó cuáles eran las pautas que deberíamos seguir y la información que teníamos que buscar al respecto .
Lo primero que hice fue investigar sobre cómo gestionar los puertos GPIO de la Raspberry Pi. Creé un pequeño programa en espacio de usuario que consistía en hacer parpadear un led "x" veces. Esto me sirvió para entender que el valor que adquiere un puerto está albergado en un directorio `"/sys/class/gpio/gpio" + gpionum + "/value"`.

A partir de este punto me centré sobre todo en investigar sobre los diversos módulos que existían en el kernel de Linux y cuál era su principal cometido. En el momento de crear nuestro propio módulo en la Raspberry, me encontré con el problema de que en la Raspberry por defecto no están incluidas las librerías, por lo tanto mi siguiente tarea fue investigar cómo enlazarlas.

Juan Carlos nos facilitó una plantilla para que pudiésemos empezar a implementar el módulo. Nunca había realizado un programa que se ejecutase en el núcleo del sistema y por lo tanto desconocía las librerías que se utilizaban y las funciones que estas incluían. Al igual que con la Raspberry Pi, mi primer paso fue crear un módulo en el kernel que mostrase un mensaje a través de `/var/log/kern.log` al insertar el módulo y otro mensaje cuando lo eliminásemos.

Una vez entendido el funcionamiento básico, conecté un botón a la protoboard y a su vez a un puerto GPIO de la Raspberry Pi. Me creé un programa que manejase la interrupción tras haber pulsado el botón y cambiase el valor del GPIO al que estaba conectado. Esto me sirvió para entender cómo se escribía y se leía del fichero de caracteres.

El siguiente paso fue analizar cómo se enviaban datos a través del comando `echo`. Entre mi compañero Juan y yo creamos una función sendBit que envía a través del puerto GPIO lo que escribes por consola. Ejemplo:

```
pi@raspberrypi: $ echo  "hola mundo" > /dev/gpio
```

El GPIO que tenemos asignado al envío de la información estaba conectado a un led, y la señal de reloj que generaba la Raspberry Pi a otro. Cada vez que transmitimos un bit dejábamos un sleep de 1 segundo por lo que éramos capaces de observar si la información enviada era la correcta.

Una vez nos aseguramos que el envío de información se estaba llevando a cabo de la manera adecuada, nos centramos en obtener la temperatura ambiente a través del sensor de temperatura. Leyendo la ficha técnica del sensor *DS18B20,* que es el que hemos empleado, vi que utilizaba un protocolo de comunicación 1-wire.
Encontre un programa en Python que realizaba lo que estábamos buscando, que era quedarnos únicamente con la temperatura en grados centígrados. Entonces basándome en ese código realicé un programa similar en lenguaje C.

Sobre el programa en espacio de usuario no tuve que buscar mucha información puesto que lo domino un poco más. Simplemente buscar el manual de cada función que utilizamos y ver cuáles son los argumentos que tiene cada una de ellas.

Donde más dificultades encontré fué en el diseño VHDL y en su configuración de pines. Hacía tiempo que no me enfrentaba a un problema de diseño, por lo que me costó varios días de repaso el ver cómo diseñar un buen circuito en VHDL.
Me descargué el manual de la placa extendida XStend Board V3.0 [2] para analizar cuáles eran los pines que podíamos utilizar como entrada y cuales de salida, así como su codificación para añadirla al archivo UCF.

En la realización de la memoria concretamente he redactado e ilustrado los siguientes apartados:

- Módulos del kernel
- Tipos de módulos en el kernel
- Major / minor number
- Diferencias entre espacio de usuario y kernel
- Descripción VHDL
- Planificación
- Realización del módulo y funcionamiento
- Sensor de temperatura
- Programa principal
- Módulo en el kernel
- Init
- Read
- ReceiveBit
- Write
- SendBit
- VHDL Program
- Entity
- Architecture_behavioral
- Men write
- Mem load
- Men read
- Conclusiones
- Futuras líneas de investigación.

8.2. Juan Samper González

Al comenzar el proyecto, nuestro tutor nos dio las primeras pautas sobre lo que deberíamos empezar mirando y en qué protocolos fijarnos. Mi trabajo al principio del proyecto fue leerme toda esta documentación y aprender todo lo referente a los distintos protocolos de comunicación. Como mis conocimientos sobre la materia no eran extensos y no había trabajado mucho en este campo, tuve que buscar bastante información acerca de esto, ya que tener una base sólida para el resto de proyecto nos ayudaría mucho.

En esta primera fase de preparación previa decidí aprender a usar git para llevar a cabo el control de versiones del proyecto. Es cierto que tanto mi compañero como yo teníamos nociones básicas sobre git, pero hasta ahora siempre habíamos realizado el control de versiones de distintos proyectos a través de carpetas compartidas en la nube. La motivación de llevar a cabo esta tarea vino impulsada tanto por la idea de mantener una buena gestión del código como por mi entrada al mundo laboral, donde vi las ventajas que suponía mantener el código a través de un control de versiones.
Mi trabajo en ese momento también fue la de buscar información acerca de github, y sobre todo, aprender a gestionar los repositorios que ahí se creaban. También ayudé a mi compañero a usar git y github mediante ejemplos, y preparé un repositorio para alojar el proyecto y llevar ahí el desarrollo del mismo.

Antes de comenzar con el diseño e implementación de nuestro protocolo de comunicación, decidí acudir a una tienda de componentes electrónicos, donde compré el material adecuado para llevar a cabo diferentes prácticas que conectaban a la Raspberry Pi con diversos componentes electrónicos. En su mayoría se realizaban usando los pines de 3.3V y de GND, sin estar muy presente el control de los pines. Aunque bien es cierto que en alguna de ellas sí que se configuraban los pines, activándolos o desactivándolos, se hacía a través de funciones de librerías, siempre en programas en espacio de usuario. Aunque no estuviera directamente relacionado con lo que queríamos desarrollar en nuestro trabajo, si que me sirvió para obtener una visión de cómo se llevaban a cabo las comunicaciones entre la Raspberry Pi y distintos componentes electrónicos, adquirir soltura con el manejo de éstos y descubrir cómo se comportaban las funciones de control de los pines del GPIO en los distintos lenguajes.

Durante la siguiente fase, mi compañero y yo comenzamos a investigar sobre los tipos de módulos que hay para el *kernel.* Yo realicé distintos ejemplos para familiarizarme con las etapas de creación del módulo: compilación a través del comando *make* y el fichero *makefile*, inserción y borrado del módulo, y escritura y lectura, gestionada por el módulo, a un fichero de caracteres (*/dev/gpio*). Era el momento de comenzar con el desarrollo en la Raspberry Pi.

Una vez tuve un primer ejemplo en la Raspberry Pi no fui capaz de compilarlo con los pasos que había seguido previamente en mi ordenador con Ubuntu. Esto es

debido a que la última actualización del sistema operativo Raspbian no traía las librerías para poder compilar módulos para el kernel. A través de un proyecto de software libre, llamado *RPi-Source* y alojado en Github, fui capaz de compilar el módulo, ejecutando los pasos que explican en la wiki del proyecto (https://github.com/notro/rpi-source).

Mi siguiente tarea consistió en la búsqueda de las direcciones de memoria de los pines que íbamos a usar para la comunicación en el manual de periféricos de la Raspberry Pi (BCM2835 ARM Peripherals [3]). Este manejo fue sustituido por el uso de las funciones de la librería <linux/gpio.h>, ya que decidí buscar sobre una mejor accesibilidad a los pines.

Tras el diseño de ambas partes por separado y la comprobación del correcto funcionamiento de cada una, quedaba unir las dos placas. En esta etapa surgieron problemas, ya que no conseguíamos ver dónde estaba el fallo. Aquí mi trabajo consistió en depurar los procesos de envío de datos por parte de la Raspberry Pi y carga de datos en el buffer de la FPGA a través de diodos LED conectados a la protoboard. Una vez podía asegurar que el envío se hacía de manera correcta, realicé la segunda etapa, de la FPGA a la Raspberry Pi de manera análoga.

Después de comprobar que la comunicación era real, quedaba ir haciendo pruebas del sistema e ir arreglando los errores que veíamos. En paralelo a esta tarea, estuve escribiendo esta memoria. Concretamente he redactado e ilustrado los siguientes apartados:

- El resumen inicial, como su traducción al inglés.
- La introducción completa, junto con su traducción al inglés.
- El punto 3.1 y todos sus apartados, y el punto 3.2 del capítulo 3 - Estado del arte.
- El capítulo 4 - Planificación y todos sus apartados.
- El capítulo 6 - el apartado de Resultados así como la traducción del capítulo 6 completa, mostrada en el capítulo 7.

Bibliografía

[1] Xess corporation. (2001-2005). XSA-3S1000 Board V1.0, User Manual. How to install, test, and use your new XSA-3S1000 Board. 48.

[2] Xess corporation. (1998-2008). XStend Board V3.0 Manual How to install and use your new XStend Board.36.

[3] Broadcom Corp.. (2012, February 06). *BCM2835 ARM Peripherals*, 205. 2016.

[4] Raspberry Pi Foundation, disponible en https://www.raspberrypi.org/

[5] Xilinx Corp. disponible en http://www.xilinx.com/

[6] Jiménez, A., Domínguez, M., Corezuelo, E., Paz, R., Jiménez, G., Linares, A. & Villar, J.I. (2015). *Práctica de desarrollo de interfaces hardware/software para el manejo de redes de sensores inalámbricos.* mayo 20, 2016, de Universidad Sevilla Sitio web: http://upcommons.upc.edu/bitstream/handle/2099/11925/a04.pdf

[7] Notro. (2015). rpi-source. 29-05-2016, de notro.
Sitio web: https://github.com/notro/rpi-source

[8] Dallas Semiconductor Corp.. (2015). *DS18B20 Programmable Resolution 1-Wire Digital Thermometer*, 20. 2016.