FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de fin de grado del Grado en Ingeniería de Computadores

Cifrado de memoria en un procesador Leon-3 / Memory encryption in the Leon-3 processor

Autor:
Bogdan Gabriel Voicila

Directores:
Juan Carlos Fabero Jiménez
Hortensia Mecha López

Agradecimientos

Me gustaría agradecer a mi familia/amigos/compañeros de trabajo por el apoyo continuo en esta carrera de fondo que es “estudiar y trabajar”.

A Juan Carlos y a Hortensia por todo el tiempo dedicado, por todas las dudas resueltas, por esas preguntas que son el germen de muchas otras y que en su conjunto han fomentado en mí el seguir descubriendo y aprendiendo.

Las ciencias tienen las raíces amargas, pero muy dulces los frutos.

ARISTÓTELES

Índice

Índice de figuras

1. Resumen

Las memorias RAM resistivas, una tecnología de memoria actualmente en investigación, permiten, entre otras ventajas, conservar los datos sin necesidad de alimentación. Este hecho puede hacer posible un análisis postmortem de los datos almacenados cuando el sistema se apaga, lo que podría plantear problemas de privacidad en aplicaciones con datos críticos, como los datos médicos de un paciente.

Por otra parte, el Leon3 es un diseño hardware de un procesador de propósito general de código abierto en VHDL. Puede ser implementado tanto en ASIC como en FPGA.

Para evitar el intrusismo en los datos críticos, este proyecto cubre el diseño y desarrollo de un sistema de cifrado localizado en el procesador Leon3. Su principal función será ofuscar la información entre el controlador de memoria y el chip físico de memoria usando distintas técnicas.

Palabras Clave

Leon 3, vhdl, sparc v8, SDRAM, controlador memoria, ddr2spa, cifrado, Nexys 4 DDR, Grlib, Grmon, Vivado, bcc2.

2. Abstract

Resistive random-access memory, is a technology currently on development. It allows saving data on chips even if power supply is lost.

This fact makes possible a postmortem analysis of stored data when system is brought offline. This involves some privacy issues on mission critical applications like storage of patient's medical data.

On the other side, Leon3 is an open source processor hardware design for general purpose developed using VHDL. It may be implemented as ASIC or FPGA.

In order to avoid critical data breaches, this project will cover the design and development for a cryptographic system located in Leon3 processor. Its primary function will be data obfuscation between memory controller and physical RAM chip using different techniques.

Keywords

Leon 3, vhdl, sparc v8, SDRAM, memory controller, ddr2spa, encryption, Nexys 4 DDR, Grlib, Grmon, Vivado, bcc2.

3. Introducción

En 1945, John von Neumann y otros, presentaron lo que sería el primer borrador de un informe sobre el EDVAC. [1] Este informe sentaba las bases de lo que a día de hoy conocemos como arquitectura von Neumann, los comienzos de la informática moderna.

Ya por aquel entonces la memoria RAM formaba parte de la arquitectura como dispositivo de lectura/escritura para almacenar tanto datos como instrucciones.

A día de hoy existen multitud de arquitecturas donde la memoria RAM sigue desempeñando un papel fundamental en el funcionamiento del sistema.

Ordenadores personales, móviles, tablets, dispositivos IoT, servidores, pueden ser buenos ejemplos, cada uno de ellos con una tecnología desarrollada específicamente para los requerimientos de su arquitectura.

En 1985, Sun Microsystems, presenta el diseño SPARC (Scalable Processor ARChitecture), una arquitectura RISC big-endian, siendo esta la primera arquitectura RISC abierta, es decir, las especificaciones del diseño están publicadas, así otros fabricantes pueden evolucionar su propio diseño desde una base común.

LEON es el diseño de un microprocesador de 32 bits basado en la arquitectura SPARC con el juego de instrucciones de la versión v8. Inicialmente este proyecto fue diseñado dentro de la Agencia Espacial Europea, más tarde, por Gaisler Research.

Desarrollado en VHDL sintetizable y personalizable mediante el uso de generics nos permitirá abordar el problema estudiando su arquitectura realizando las modificaciones oportunas en sus componentes.

3.1. Antecedentes

En 1971 el ingeniero eléctrico Leon Chua define el memristor, un componente faltante que relaciona la carga eléctrica con el flujo magnético [2]. En 2008, este componente se relaciona con la operación de las memorias RRAM, también llamadas ReRAM (resistive random-access memory). [3]

Si bien hay ciertas controversias respecto a la implementación física de estos nuevos elementos [4], se establecen las bases para la investigación de una nueva generación de memorias que pretende reemplazar las memorias flash actuales con una velocidad de acceso más rápida, una densidad mucho más grande, así como un consumo más reducido [5] [6].

Ahora bien, imaginemos una nueva generación de dispositivos móviles integrando la tecnología de memorias resistivas. Si bien su uso estaría más que justificado por motivos de rendimiento y consumo, se nos presenta un nuevo problema de seguridad ante la persistencia de los datos en el chip de memoria y posibles ataques de ingeniería inversa sobre chips RRAM.

Los sistemas operativos modernos suelen contar con mecanismos para el cifrado del almacenamiento externo como los discos duros o las memorias USB.

Se abre la puerta a la investigación de mecanismos de cifrado de memoria que sean independientes a un sistema operativo y permitan frenar estos posibles ataques. Actualmente, tanto AMD (SME y SVE) como Intel (TME) incorporarán tecnologías para el cifrado de memoria en las próximas generaciones de procesadores. [7] [8]

3.2. Objetivos

Previo al comienzo del desarrollo del módulo de cifrado/ofuscador, se han tenido que superar varios objetivos:

- Preparación de un entorno de desarrollo, con una prueba de ejecución en placa volcando la implementación del procesador en la placa Nexys 4 DDR.
- Familiarización con la arquitectura del procesador Leon3.
- Familiarización con el desarrollo en VHDL del procesador Leon3.
- Familiarización con el funcionamiento de la memoria DDR2 SDRAM, estructura, comandos, inicialización, lecturas/escrituras, ráfagas etc...
- Conexión de una señal externa al procesador para determinar en tiempo real el último dato leído/escrito en memoria por el controlador de memoria de una manera amigable.
- Diseño del módulo de cifrado/ofuscador e integración en el diseño del Leon3.
- Publicación del código en un repositorio git público. [19]

3.3. Plan de trabajo

La organización de este trabajo ha consistido en reuniones periódicas en las que el tutor del proyecto y el alumno se han puesto al día con los avances. A grandes rasgos, el trabajo ha comprendido de los siguientes puntos:

- Documentación sobre las características generales de la placa Nexys 4 DDR
- Documentación sobre las características generales del procesador Leon3 usando GRLIB
- Documentación sobre el uso de GRMON para la interacción con la placa
- Documentación sobre distintas vías para la ofuscación de la memoria
- Diseño y desarrollo de un módulo ofuscador dentro del controlador de memoria
- Ejecución de pruebas sobre Nexys4 DDR y resultados
- Elaboración de la memoria

4. Introduction

In 1945, John von Neumann and others, described what was the First Draft of a Report on the EDVAC. [1]. This document layed the groundwork for the today known as von Neumann architecture, the beginning of modern computer science.

Already at that time, RAM memory was part of that architecture as a read/write device in order to save data an program instructions.

Today there are many architectures where RAM continues to play a fundamental role in the operation of the system.

Personal computers, mobiles, tablets, IoT devices, servers, can be good examples, each of them with a RAM technology developed specifically for the requirements of the architecture.

In 1985, Sun Microsystems, presented the SPARC (Scalable Processor ARChitecture) design, a big-endian RISC architecture, this being the first open RISC architecture, that is, the design specifications are published, so other manufacturers can evolve their own design from a common base.

LEON is a 32-bit microprocessor design based on the SPARC architecture using version v8 instruction set. Initially this project was designed within the European Space Agency, later, by Gaisler Research.

Developed in synthesizable and customizable VHDL through the use of generics, it will allow us to address the problem by studying its architecture and making the appropriate modifications on its components.

4.1. Background

In 1971 the electrical engineer Leon Chua defined the memristor, a missing component that relates the electrical charge to the magnetic flux [2]. In 2008, this component was related to the operation of RRAM memories, also called ReRAM (resistive random-access memory). [3]

Although there are certain controversies regarding the physical implementation of these new elements [4], the bases are established for investigations on a new generation of memories that intends to replace the current flash memories with a faster access speed, a much higher density. as well as lower consumption [5] [6].

Now, let's imagine a new generation of mobile devices integrating resistive memory technology. Although its use would be more than justified for performance and consumption reasons, we are presented with a new security problem due to the persistence of data on the memory chip and possible reverse engineering attacks on RRAM chips.

Modern operating systems often have mechanisms for encrypting external storage such as hard drives or USB sticks.

The door is opened to the investigation of memory encryption mechanisms that are independent of an operating system and allow to stop these possible attacks.

Currently, both AMD (SME and SVE) and Intel (TME) will incorporate technologies for memory encryption on next generations of processors. [7] [8]

4.2. Objectives

Before starting developing, next objectives had to be reached:

- Working development environment, running a Hello World program compiled for SPARC and downloaded to Nexys 4 DDR FPGA.
- Familiarization with Leon3 processor architecture
- Familiarization with Leon3 VHDL design
- Familiarization with DDR2 SDRAM memory operation, structure, commands, initialization, reads/writes, bursts etc.
- External signal connection to the processor in order to verify on real time the last word read/written by the memory controller on physical memory chip.
- Encryption/Obfuscator component design and integration on Leon3 VHDL design
- Publish the code to a public git repository [19]

4.3. Workplan

The organization of this work has consisted of periodic meetings in which the project tutor and the student have been updated with the progress. Broadly speaking, the work has included the following points:

- Nexys 4 DDR general characteristics
- Leon3 processor and GRLIB documentation
- Using GRMON in order to connect to Nexys 4 DDR FPGA
- Different approaches for memory obfuscation
- Design and development of a obfuscator inside the DDR2SPA controller
- Testing execution on Nexys4 DDR and results.
- Development of this document.

5. Estado del arte

5.1. Nexys 4 DDR

La placa Nexys 4 DDR es una FPGA de la familia Artix-7 comercializada por Xilinx siendo la evolución de la familia Spartan 6.

La placa se ha proporcionado para el desarrollo de este proyecto como recurso desde la Universidad Complutense de Madrid.

Incluye un módulo de memoria DDR2 de 128 MB con 8 bancos internos y una longitud de palabra de 16 bits funcionando a 140 MHz. Será este módulo el que utilizará el procesador LEON3 como dispositivo externo conectado al controlador de memoria DDR2SPA objeto del estudio.

El chip de memoria está fabricado por Micron siendo el modelo MT47H64M16HR-25E. [9]

Para el desarrollo de las pruebas se han utilizado otros dispositivos de la placa como el display de 7 segmentos o los leds para visualizar señales como la última palabra leída/escrita en memoria.

Para más información el manual de referencia está disponible para su consulta en la web. [10].

5.2. GRLIB

Para el estudio y desarrollo del diseño del procesador Leon3, Gobham Gaisler AB ofrece la librería GRLIB bajo licencia GPL que contiene todos los archivos fuente probados en distintas FPGAs del mercado soportando distintas configuraciones.

En nuestro caso, la placa Nexys 4 DDR es uno de los diseños soportados. En el fichero README.txt del diseño ubicado en la ruta "grlib/designs/leon3-digilent-nexys4ddr" se especifican los siguientes aspectos relacionados con la simulación y la síntesis del diseño:

> The design currently supports synthesis with Xilinx Vivado (tested with Vivado 2017.3).
> The design currently supports simulation with modelsim 10.5a and riviera 2017.2

Posicionados en la ruta arriba indicada, con el comando make podemos interactuar con el diseño mediante comandos predefinidos.

Algunos de los comandos más utilizados durante el proyecto:

```
#Proceso completo, generando el fichero bit desde la consola
make vivado

#Abre el proyecto con Vivado.
make vivado-launch

#Vuelca el fichero bit en la placa.
make vivado-prog-fpga

#Limpia los ficheros generados por Vivado
make vivado-clean
```

Otro comando importante sería **make xconfig**, una interfaz visual a través de la cual podemos configurar el diseño de nuestro procesador.

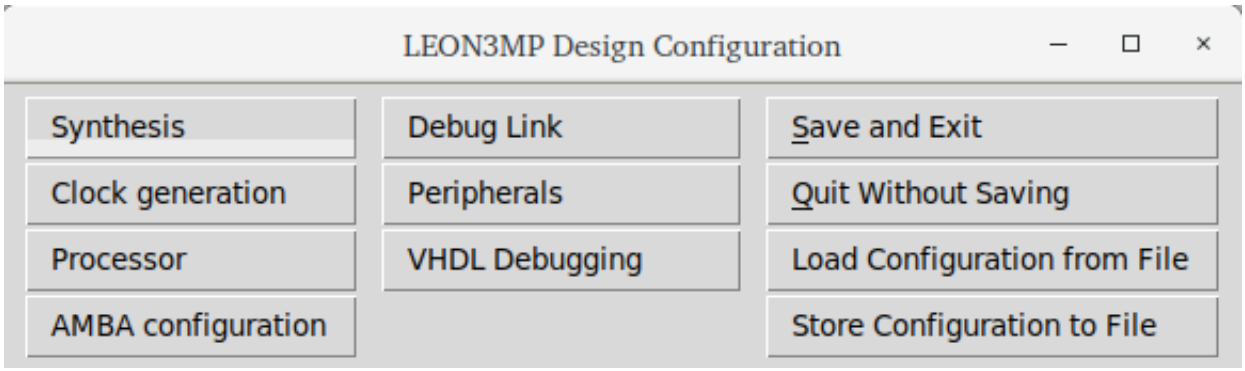

Figura 1: GUI xconfig

Esta configuración se guarda en el fichero **config.vhd** de la misma ruta. Para más información sobre el uso de GRLIB podemos consultar el manual de referencia. [11]

5.3. Vivado 2017.3

Como entorno de desarrollo y según lo indicado en el fichero readme.txt del diseño en GRLIB para Nexys4 DDR vamos a usar la versión 2017.3 disponible en la web de Xilinx.

Descargamos el fichero "Vivado HLx 2017.3: WebPACK and Editions - Linux Self Extracting Web Installer (BIN - 100.61 MB)" en la sección "Vivado Design Suite - HLx Editions - 2017.3 Full Product Installation".

Esto nos descargará un fichero con extensión ".bin". Lo instalaremos añadiéndole permisos de ejecución y lanzándolo. La instalación se tiene que hacer con sudo. A continuación, se muestran algunas capturas del proceso de instalación:

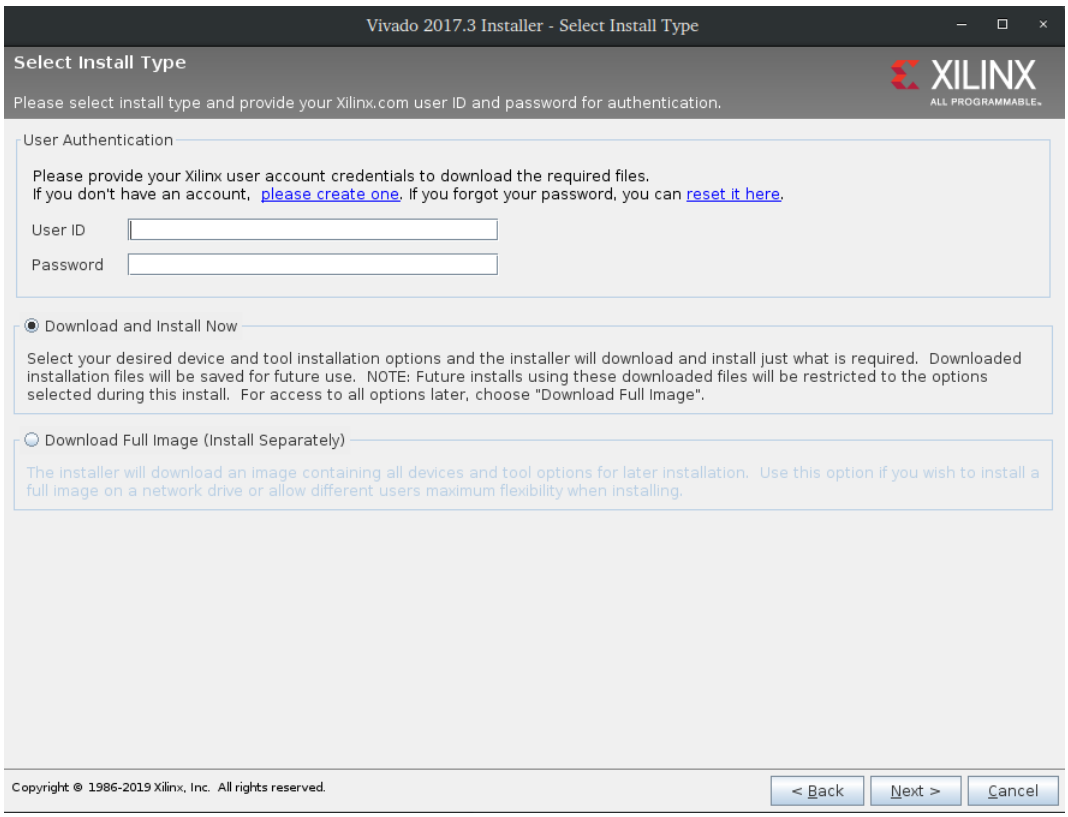

Figura 2: Instalación Vivado Paso 1

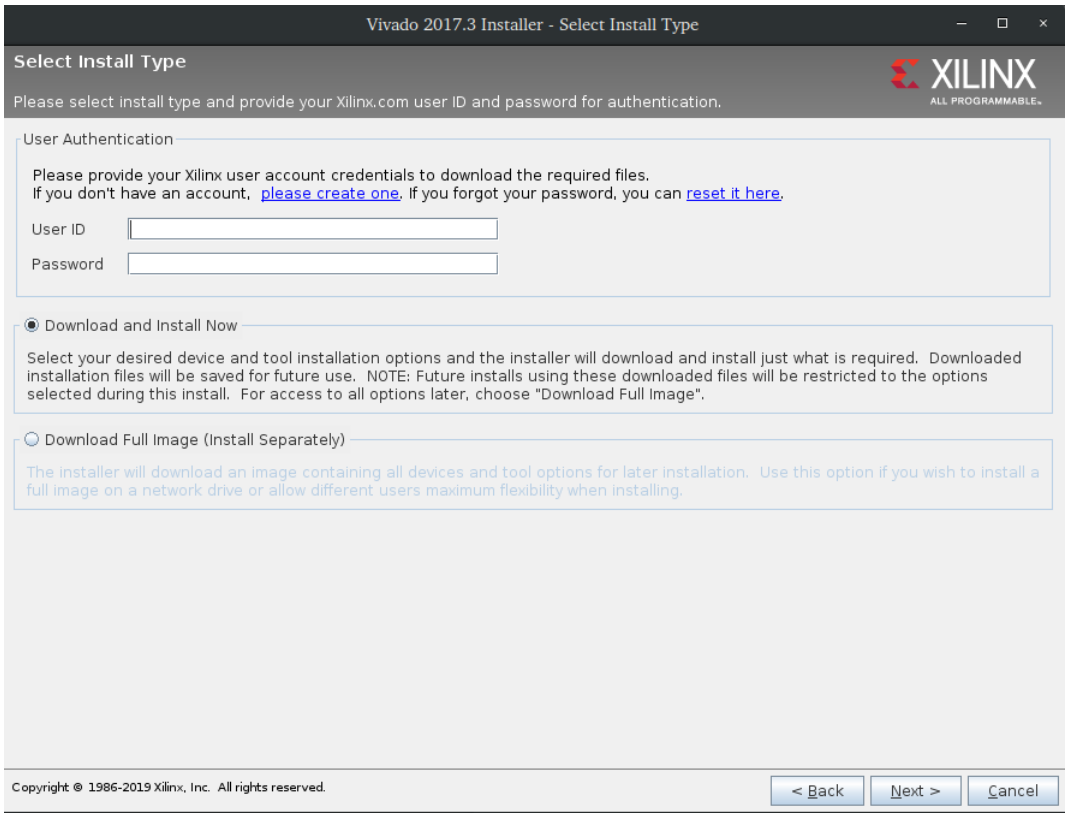

Figura 3: Instalación Vivado Paso 2

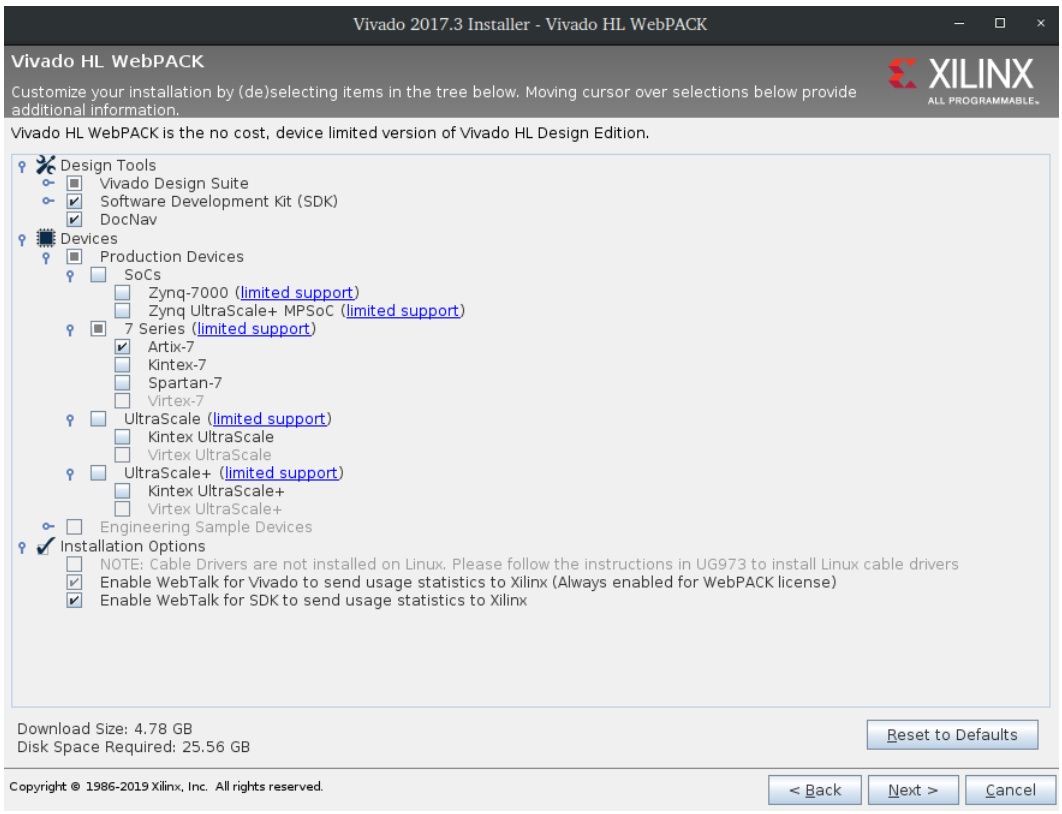

Figura 4: Instalación Vivado Paso 3

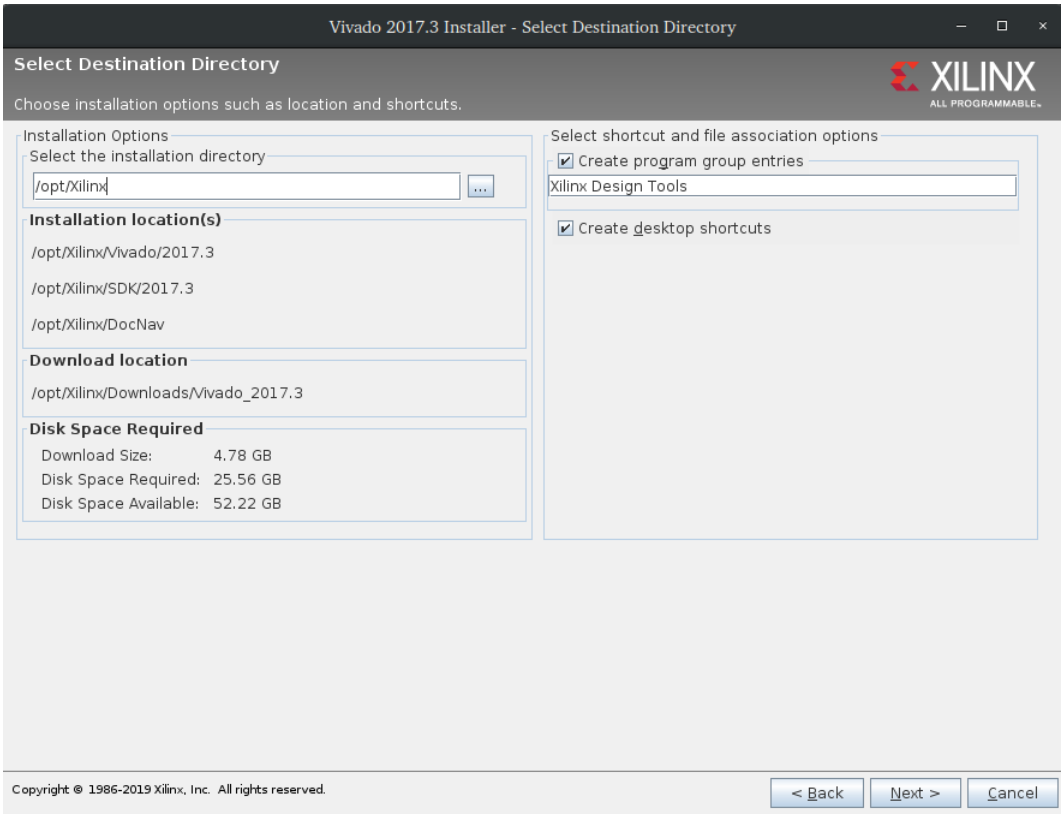

Figura 5: Instalación Vivado Paso 4

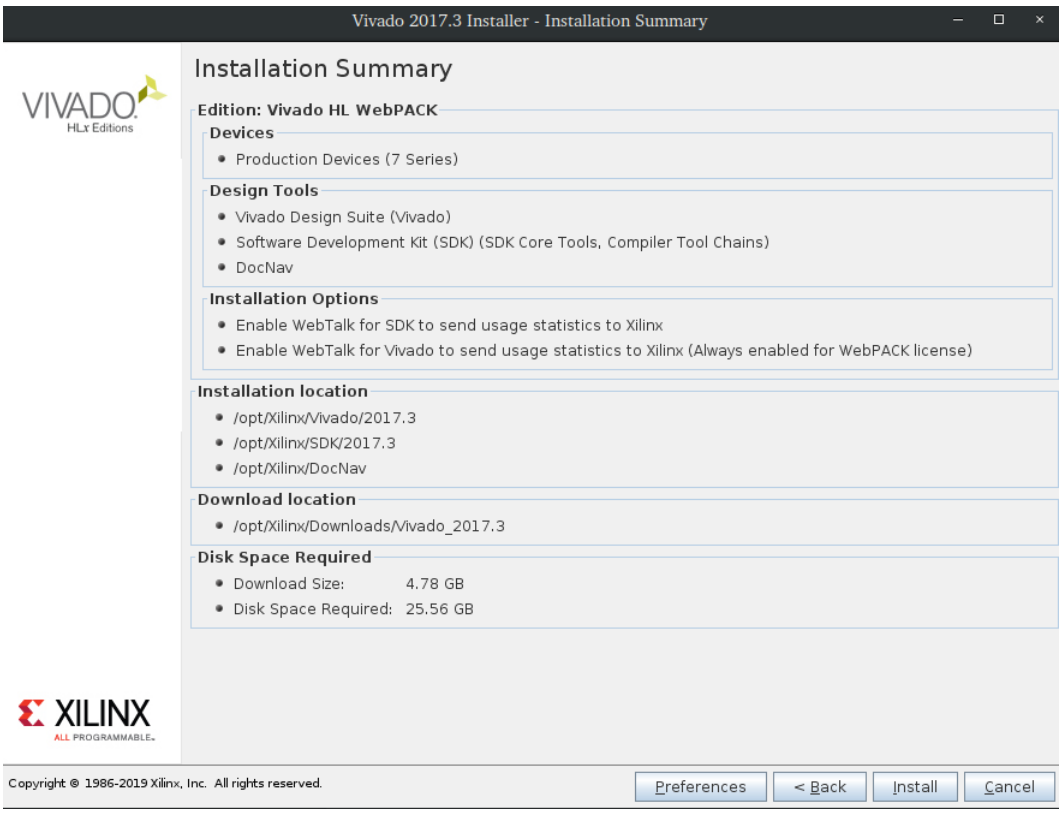

Figura 6: Instalación Vivado Paso 5

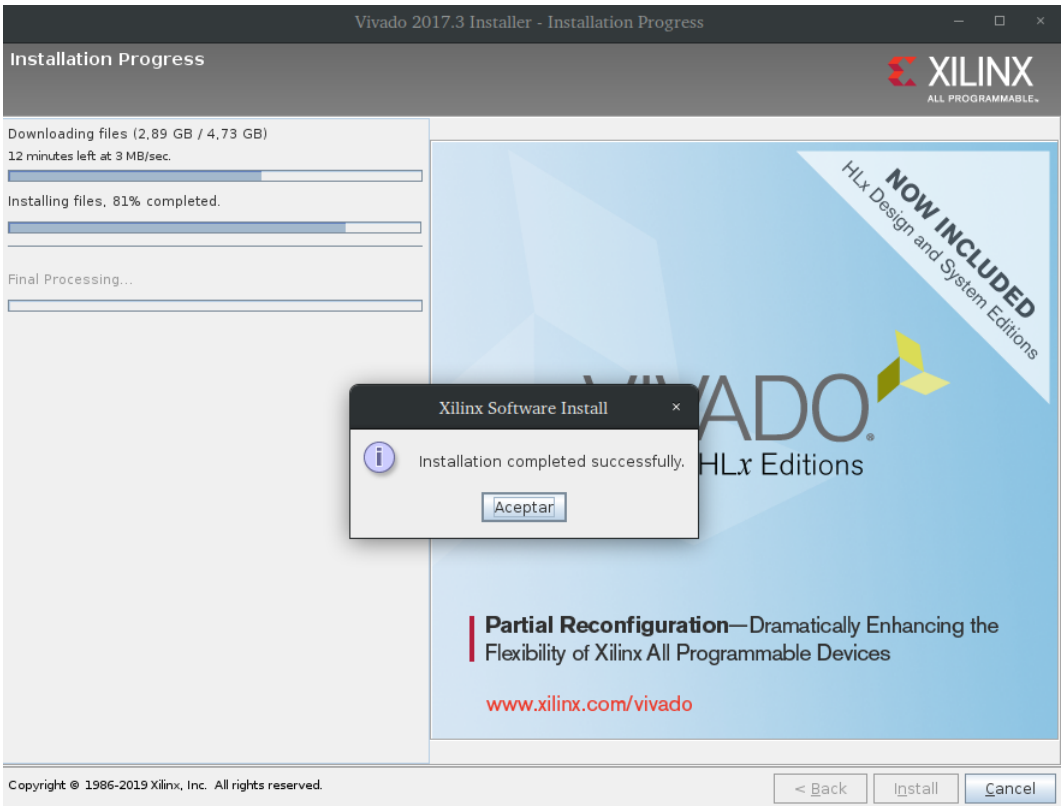

Figura 7: Instalación Vivado Paso 6

Para terminar la instalación añadimos la siguiente línea a nuestro fichero .bashrc:

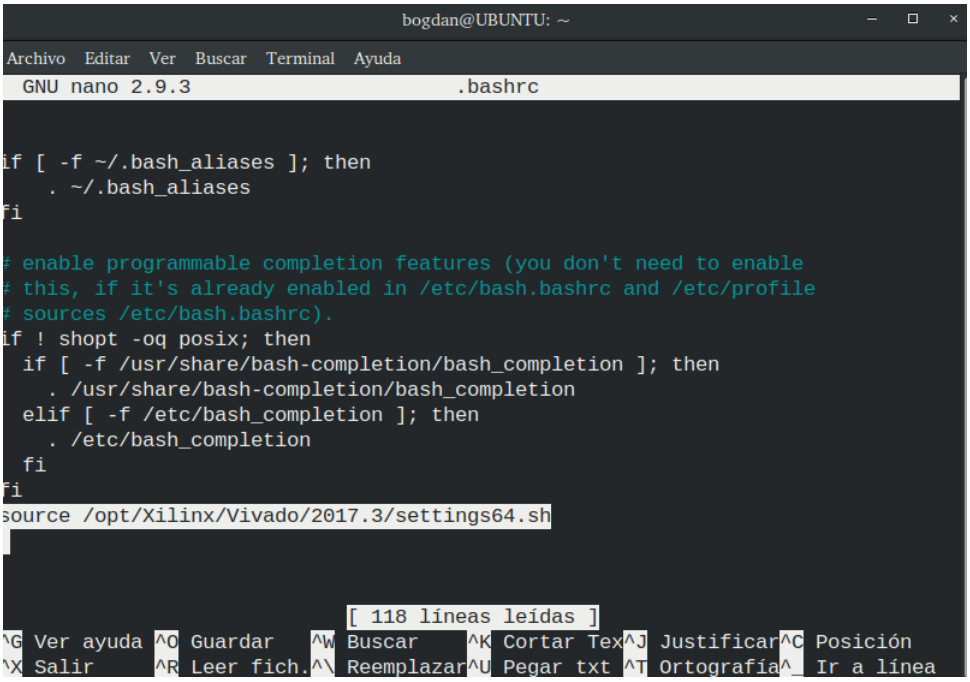

Figura 8: Configuración bashrc

Esto nos permitirá ejecutar vivado desde el terminal como un comando conocido más.

Podemos crear un acceso directo a la aplicación en la ruta `/usr/share/applications` llamado `Vivado_2017.3.desktop`

```
[Desktop Entry]
Encoding=UTF-8
Type=Application
Name=Vivado 2017.3
Comment=Vivado 2017.3
Icon=/opt/Xilinx/Vivado/2017.3/doc/images/vivado_logo.png
Exec=/opt/Xilinx/Vivado/2017.3/bin/vivado
```

Por último, para poder conectar nuestra placa tenemos que instalar los controladores USB que se facilitan con la instalación de Vivado:

Figura 9: Instalación drivers Digilent

5.4. Compilador cruzado - BCC

Para poder ejecutar un programa con el diseño del Leon3 en nuestra placa tenemos que compilarlo con un compilador cruzado, es decir, si bien nuestro desarrollo en C estará en una arquitectura linux, el fichero ejecutable se compilará para la arquitectura SPARC distinta a la x86 origen.

En la web oficial de Gobham Gaisler en el apartado de descargas nos provee el paquete llamado BCC en su versión 1 o 2 según la versión GCC utilizada. [14]

Para instalarlo en nuestro sistema operativo podemos descomprimirlo en la ruta `/opt/bcc` y añadiendo el siguiente path en nuestro fichero .profile (es necesario reiniciar la sesión para que los cambios surjan efecto):

```
export PATH=/opt/bcc/bin:$PATH
export PATH=/opt/bcc/man:$PATH
```

Después podemos compilar cualquier fichero .c con el comando:

```
sparc-gaisler-elf-gcc -O2 -g holamundo.c -o holamundo
```

5.5. GRMON

GRMON es una herramienta que se utiliza para realizar la depuración de programas en hardware sobre FPGA a través de una interfaz predefinida. Lo podemos descargar desde la página oficial [13] y utilizarlo en modo evaluación de manera temporal.

En nuestro caso vamos a utilizar el puerto USB junto a la UART del diseño de LEON3 para descargar los programas compilados para arquitectura SPARC y ejecutarlos.

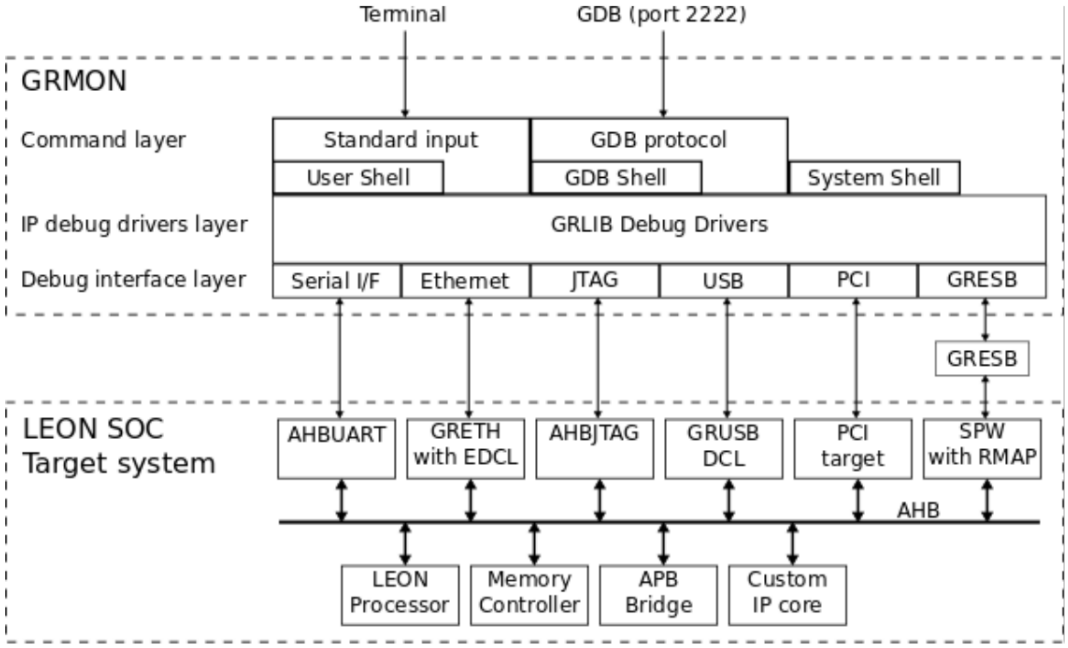

Figura 10: Arquitectura GRMON

Para más información acerca de los comandos soportados y los prerrequisitos hay documentación disponible en la web de Gobham Gaisler. [15]

5.6. 'Hello World' ejecutado por Leon3 en Nexys4DDR

En este apartado vamos a realizar el proceso completo desde la generación del código en C hasta la ejecución en la placa detallando cada paso.

5.6.1. El código

Compilaremos el siguiente código en c:

```
#include <stdio.h>
static const char msg[] = "Hello World";
int main()
{
        printf("Saved message in memory -> %s\n", msg);
}
```

La idea es que esta cadena se guarde en la memoria para posteriormente ser mostrada por la salida estándar.

5.6.2. Compilando el código

Una vez guardado el fichero con formato ".c" procedemos a compilarlo usando el compilador BCC.

```
bogdan@UBUNTU:~/Escritorio/tfg/6.Desarrollo$ sparc-gaisler-elf-gcc holamundo.c -o holamundo
bogdan@UBUNTU:~/Escritorio/tfg/6.Desarrollo$ ls -l hola*
-rwxrwxr-x 1 bogdan bogdan 501336 sep  6 15:49 holamundo
-rw-rw-r-- 1 bogdan bogdan    126 ago  6 19:40 holamundo.c
bogdan@UBUNTU:~/Escritorio/tfg/6.Desarrollo$
```

Figura 11: Compilando el código

Será este programa el que cargaremos en la placa usando GRMON.

5.6.3. Generación del fichero bit

Mediante la interfaz visual con el comando "make xconfig" comprobamos que el controlador DDR2 forma parte del diseño del procesador y tiene activada su inicialización.

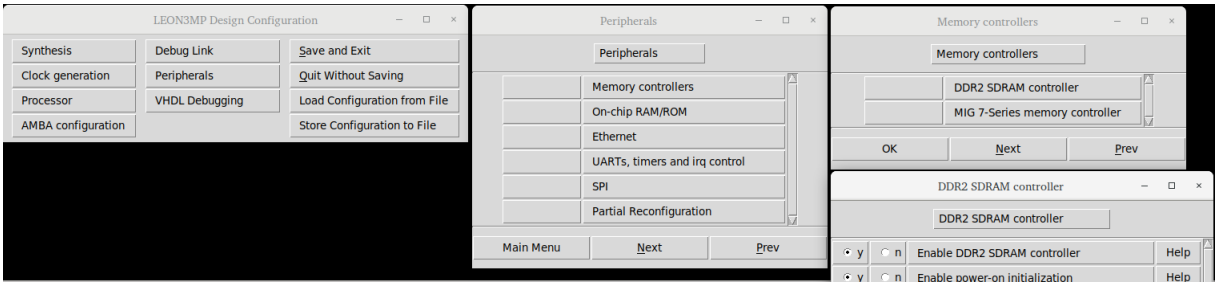

Figura 12: Configuracíon periféricos leon3

Procedemos por tanto a lanzar la síntesis, el place and route del diseño así como la generación del fichero .bit. La salida del comando es muy extensa, para simplificar se muestran las últimas líneas:

```
Loading route data...
Processing options...
Creating bitmap...
Creating bitstream...
Writing bitstream ./leon3mp.bit...
Creating mask data...
Creating bitstream...
Writing bitstream ./leon3mp.msk...
INFO: [Vivado 12-1842] Bitgen Completed Successfully.
INFO: [Project 1-120] WebTalk data collection is mandatory when using a WebPACK part without a full Vivado license. To see the specific WebTalk data colle
cted for your design, open the usage_statistics_webtalk.html or usage_statistics_webtalk.xml file in the implementation directory.
INFO: [Common 17-83] Releasing license: Implementation
100 Infos, 97 Warnings, 0 Critical Warnings and 0 Errors encountered.
write_bitstream completed successfully
write_bitstream: Time (s): cpu = 00:00:41 ; elapsed = 00:00:17 . Memory (MB): peak = 2748.434 ; gain = 205.750 ; free physical = 448 ; free virtual = 3685
INFO: [Common 17-206] Exiting Vivado at Sun Sep  6 16:04:33 2020...
[Sun Sep  6 16:04:34 2020] impl_1 finished
wait_on_run: Time (s): cpu = 00:05:29 ; elapsed = 00:02:45 . Memory (MB): peak = 1866.438 ; gain = 0.000 ; free physical = 1609 ; free virtual = 4895
INFO: [Common 17-206] Exiting Vivado at Sun Sep  6 16:04:34 2020...
bogdan@UBUNTU:~/Escritorio/tfg/3.Work/grlib-gpl-2018.1-b4217/designs/leon3-digilent-nexys4ddr$
```

Figura 13: Salida make vivado

Por último, volcamos el fichero ".bit" en la placa usando el comando "make vivado-prog-fpga" estando conectada esta vía USB.

```
Configuring Device 1 (xc7a100t) with Bitstream -- ./vivado/leon3-digilent-nexys4ddr/leon3-digilent-nexys4ddr.runs/impl_1/leon3mp.bit
.....10.....20.....30.....40.....50.....60....70.....80.....90.....Successfully downloaded bit file.

JTAG chain configuration
-----------------------------------------------------
Device   ID Code         IR Length    Part Name
 1       13631093            6       xc7a100t

bogdan@UBUNTU:~/Escritorio/tfg/3.Work/grlib-gpl-2018.1-b4217/designs/leon3-digilent-nexys4ddr$
```

Figura 14: Volcado del fichero bit a la placa

5.6.4. Conexión con GRMON, carga y ejecución

Nos conectamos a la placa usando el siguiente comando:

```
#-u Forward application console I/O to GRMON
#-digilent Digilent HS1/HS2/HS3 Adept JTAG connection

grmon -u -digilent
```

Si al lanzar GRMON nos aparece el error “libdabs.so: cannot open shared object file: No such file or directory” procedemos con la instalación del paquete `digilent.adept.runtime_2.19.2-amd64.deb`. [17]

```
bogdan@UBUNTU:~/Escritorio/tfg/5.GRMON/grmon-eval-3.2.3/linux/bin64$ ./grmon -digilent -u

  GRMON debug monitor v3.2.3 64-bit eval version

  Copyright (C) 2020 Cobham Gaisler - All rights reserved.
  For latest updates, go to http://www.gaisler.com/
  Comments or bug-reports to support@gaisler.com

  This eval version will expire on 26/11/2020

JTAG chain (1): xc7a100t
WARNING! Stack pointer not set
  GRLIB build version: 4217
  Detected frequency:  70.0 MHz

  Component                            Vendor
  LEON3 SPARC V8 Processor             Cobham Gaisler
  AHB Debug UART                       Cobham Gaisler
  JTAG Debug Link                      Cobham Gaisler
  GR Ethernet MAC                      Cobham Gaisler
  AHB/APB Bridge                       Cobham Gaisler
  LEON3 Debug Support Unit             Cobham Gaisler
  Single-port DDR2 controller          Cobham Gaisler
  SPI Memory Controller                Cobham Gaisler
  Generic UART                         Cobham Gaisler
  Multi-processor Interrupt Ctrl.      Cobham Gaisler
  Modular Timer Unit                   Cobham Gaisler
```

Figura 15: Conexión con GRMON

Antes de proceder con la carga del programa, comprobamos la información que tenemos sobre los distintos dispositivos conectados al bus AHB con el comando “info sys”.

```
grmon3> info sys
  cpu0      Cobham Gaisler  LEON3 SPARC V8 Processor
            AHB Master 0
  ahbuart0  Cobham Gaisler  AHB Debug UART
            AHB Master 1
            APB: 80000700 - 80000800
            Baudrate 115200, AHB frequency 70.00 MHz
  ahbjtag0  Cobham Gaisler  JTAG Debug Link
            AHB Master 2
  greth0    Cobham Gaisler  GR Ethernet MAC
            AHB Master 3
            APB: 80000f00 - 80001000
            IRQ: 12
            edcl ip 192.168.0.48, buffer 2 kbyte
  apbmst0   Cobham Gaisler  AHB/APB Bridge
            AHB: 80000000 - 80100000
  dsu0      Cobham Gaisler  LEON3 Debug Support Unit
            AHB: 90000000 - a0000000
            AHB trace: 64 lines, 32-bit bus
            CPU0:  win 8, itrace 64, V8 mul/div, lddel 1
                   stack pointer 0x00000000
                   icache 4 * 4 kB, 32 B/line, lrr
                   dcache 4 * 4 kB, 32 B/line, lrr
  ddr2spa0  Cobham Gaisler  Single-port DDR2 controller
            AHB: 40000000 - 48000000
            AHB: fff00100 - fff00200
            No SDRAM found
  spim0     Cobham Gaisler  SPI Memory Controller
            AHB: fff70000 - fff70100
            AHB: 00000000 - 01000000
            IRQ: 7
            SPI memory device read command: 0x0b
  uart0     Cobham Gaisler  Generic UART
```

Figura 16: GRMON info sys

Como muestra la captura anterior, no se detecta la memoria ram por lo que, tal y como se indica en el manual de GRMON, lanzamos el comando **ddr2delay scan** que ejecuta una rutina de calibración.

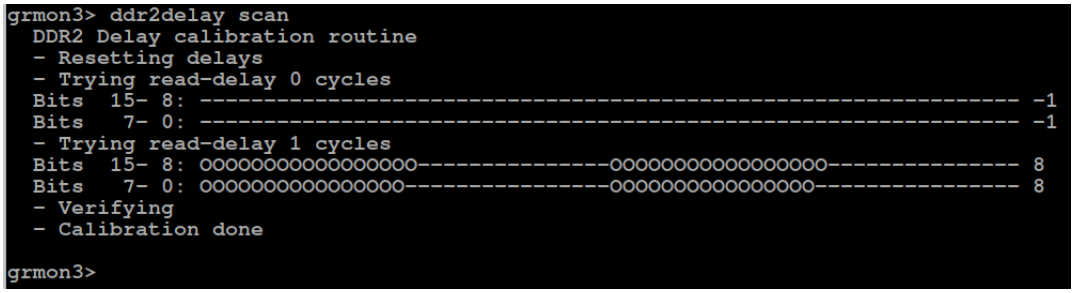

Figura 17: Calibración de memoria

Una vez ejecutada tenemos que salir y volver a conectar con grmon a la placa, lanzando el comando info sys ya podemos ver la información correcta.

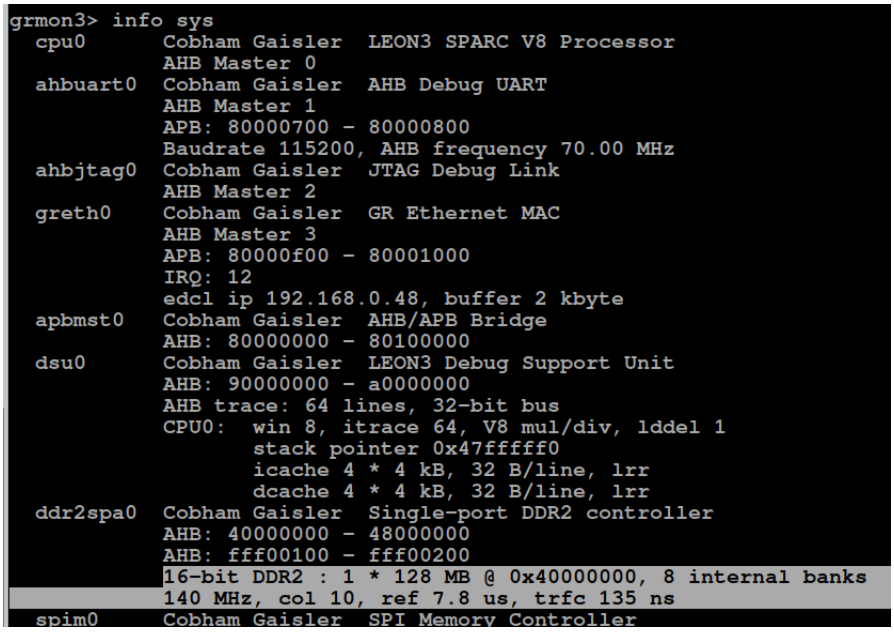

Figura 18: Controlador DDR2 identificado por GRMON

Procedemos con la carga del programa usando el comando load y el path completo del ejecutable que compilamos en un paso anterior.

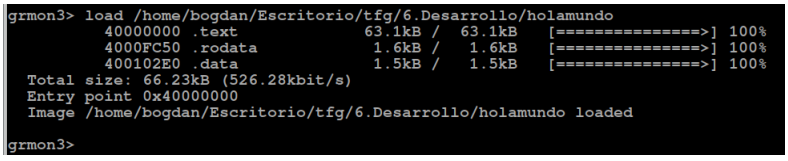

Figura 19: Carga del programa en memoria

La carga del programa en la placa copia los datos del ejecutable en las secciones correspondientes:

- .text – Instrucciones del programa
- .rodata – Datos de solo lectura no modificables, constantes
- .data – variables

En nuestro caso, en el código hemos declarado una constante de tipo **static const char** por lo que podemos examinar su contenido en la sección .rodata comenzando en la dirección **0x4000FC50**.

Utilizando el comando mem **0x4000FC50** podemos verificar efectivamente el contenido de esta posición en memoria:

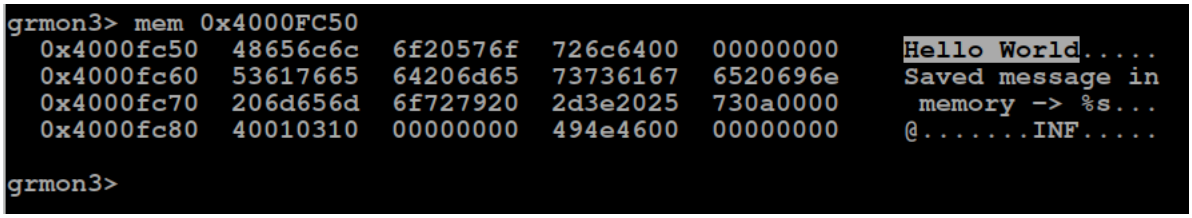

Figura 20: Acceso lectura a una posición de memoria

Otro punto importante es la definición del "entry point" es decir, el punto de entrada donde el procesador comenzará a ejecutar las instrucciones.

En este caso se ha establecido por el comando **load** a la dirección de memoria **0x40000000** que es el comienzo de la sección **.text**.

Se puede consultar también usando el comando **ep**:

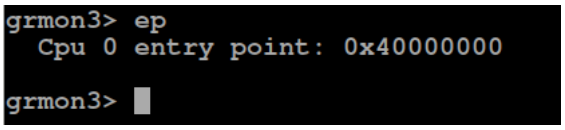

Figura 21: Punto de entrada al programa

Por último, lanzamos la ejecución del programa con el comando **run**:

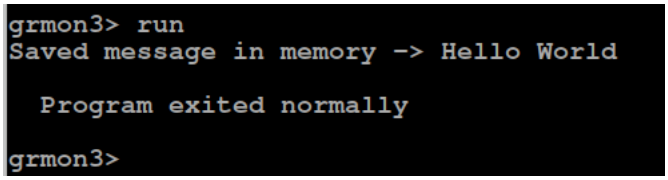

Figura 22: Ejecución del programa

Para volver a ejecutar el programa tenemos que volver a cargarlo con el comando **load**.

En este apartado hemos resumido los pasos principales para compilar un programa, volcarlo en la placa Nexys4 DDR y ejecutarlo usando las herramientas anteriormente descritas.

6. Planificación

6.1. Identificación del problema

6.1.1. Conexión con el módulo externo de memoria

El procesador Leon3 cuenta con varias señales externas que controlan la memoria externa localizada en un chip independiente (Micron).

Estas señales aparecen como puertos en la definición de la entidad leon3mp. También podemos ver su conexión a los pines definidos en el fichero leon3mp.xdc.

A continuación, enumeramos estas señales definiéndolas con un pequeño comentario.

```
--Bus de datos entrada/salida (16 bits)
ddr2_dq         : inout   std_logic_vector(15 downto 0);
--Dirección de la memoria
ddr2_addr       : out   std_logic_vector(12 downto 0);
--Selección del banco de memoria
ddr2_ba         : out   std_logic_vector(2 downto 0);
--Dirección de fila
ddr2_ras_n      : out   std_ulogic;
--Dirección de columna
ddr2_cas_n      : out   std_ulogic;
--Habilita la escritura en memoria
ddr2_we_n       : out   std_ulogic;
--Habilita el reloj
ddr2_cke        : out   std_logic_vector(0 downto 0);
--On Die Termination para mejorar la calidad de la señal
ddr2_odt        : out   std_logic_vector(0 downto 0);
--Selección de chip
ddr2_cs_n       : out   std_logic_vector(0 downto 0);
--Mascara para los datos
ddr2_dm         : out   std_logic_vector(1 downto 0);
--Reloj para la linea de datos entrada/salida según lectura/escritura
ddr2_dqs_p      : inout   std_logic_vector(1 downto 0);
--Reloj invertido para la linea de datos entrada/salida según lectura/escritura
ddr2_dqs_n      : inout   std_logic_vector(1 downto 0);
--Reloj positivo
ddr2_ck_p       : out   std_logic_vector(0 downto 0);
--Reloj negativo
ddr2_ck_n       : out   std_logic_vector(0 downto 0);
```

Teniendo en cuenta el chip de memoria y su tecnología DDR2, encontramos en la documentación [9] su diagrama de estados, estados que será encargado de gestionar el controlador de memoria DDR2SPA descrito en el siguiente apartado.

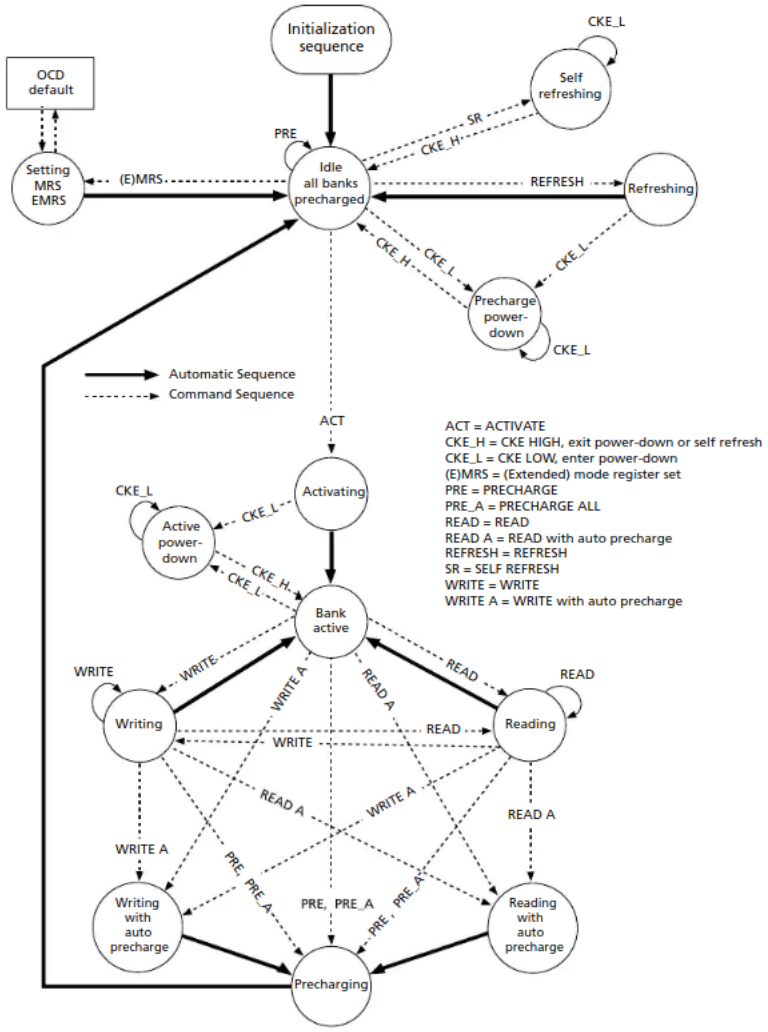

Figura 23: Diagrama de estados del modulo de memoria Micron MT47Hxx(x)M4/8/16

Por tanto, para el desarrollo del módulo ofuscador tenemos que tener en consideración distintos aspectos relativos a la estructura de la memoria ubicándolo en el nivel lógico correcto.

6.1.2. Estructura interna del controlador de memoria

El controlador de memoria DDR2SPA tiene dos componentes bien diferenciados, un controlador que va conectado en modo esclavo al bus AHB (ddrc) y un componente que implementa distintas tecnologías de memorias RAM seleccionables mediante el uso de generics `ddr_phy`.

En el manual "GRLIB IP Core User's Manual" [16], pagina 189, podemos encontrar un diagrama a más alto nivel con las señales incluidas.

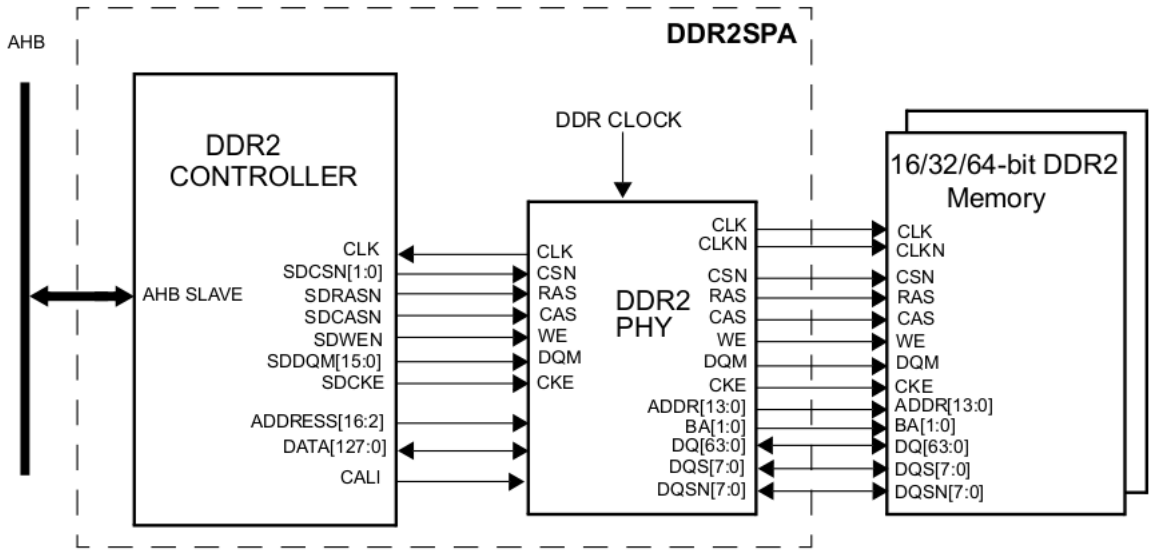

Figura 24: Estructura interna del controlador DDR2SPA

Este componente se genera en el fichero principal del procesador, `leon3mp.vhd` pasándole algunas constantes genéricas definidas en la configuración del procesador como `CFG_DDR2SP_DATAWIDTH`, fabtech o memtech (`ddr2spa.vhd`).

Dentro del fichero **ddr2spa.vhd**, encontramos los dos componentes antes mencionados, el controlador definido en el fichero **ddr2spax.vhd** y el módulo que genera la parte correspondiente en función de la tecnología especifica (`ddrphy_wrap.vhd` y `ddr2phy_wrap_cbd.vhd`) en nuestro hardware.

Estos dos módulos están interconectados mediante señales locales definidas en el fichero **ddr2spa.vhd**. Se utilizan también tipos de datos predefinidos (records) como son:

```
--señal con origen DDR2 PHY destino DDR2 CONTROLLER
sdi ddrctrl_in_type
--señal con origen DDR2 CONTROLLER a DDR2 PHY
sdo ddrctrl_out_type
--señal con origen bus AHB destino DDR2SPA
ahbsi ahb_slv_in_type
--señal con origen DD2SPA destino bus AHB
ahbso ahb_slv_out_type
```

Estos datos estructurados están declarados en el fichero **ddrpkg.vhd**.

Si nos adentramos en la estructura del controlador (**ddr2spax.vhd**) vemos que hay declarados dos buffers: **wbuf** (para escritura) y **rbuf** (para lecturas). Ambos buffers hacen referencia a la misma entidad definida en el fichero **ddr2buf.vhd**.

Aquí es donde radica la característica principal del tipo de memorias DDR2 SDRAM (**D**ouble **D**ata **R**ate type **two** **S**ynchronous **D**ynamic **R**andom-**A**ccess **M**emory).

En la documentación “GRLIB IP Core User’s Manual” [16] se indica el punto 22.7.3 las distintas configuraciones y su uso respectivo de bloques de memoria y tamaños del buffer:

DDR width	AHB width	Write FIFO block-RAM usage			Read-FIFO block-RAM usage			Total RAM count
		Count	Depth	Width	Count	Depth	Width	
16	32	1	16	32	1	8	32	2
16	64	2	8	32	2	4	32	4
16	128	4	4	32	4	2	32	8
16	256	8	2	32	8	1	32	16
32	32	2	8	32	1	4	64	3
32	64	2	8	32	1	4	64	3
32	128	4	4	32	2	2	64	6
32	256	8	2	32	4	1	64	12
64	32	4	4	32	1	2	128	5
64	64	4	4	32	1	2	128	5
64	128	4	4	32	1	2	128	5
64	256	8	2	32	2	1	128	10

Figura 25: Uso de bloques de memoria y tamaños buffer

Por tanto, las modificaciones ser harán sobre este buffer diferenciando entre las escrituras y las lecturas manteniendo la integridad de las dos FIFOs.

En el caso de nuestra placa contamos con un ancho de palabra de 16 bits (definido por el generic **ddrbits**), un ancho de bus AHB de 32 bits (definido por la variable `CFG_AHBDW` en el fichero `libgrlib/stdlib/config.vhd`) por tanto nos encontramos en el caso de la primera fila.

Contamos con un buffer de lectura y otro de escritura, ambos teniendo un ancho de 32 bits. En cuanto a la profundidad del buffer, para las lecturas contamos con 8 posiciones (bytes) mientras que para las escrituras con 16 (bytes).

Cada buffer usa una RAM `syncram_2p` con un puerto de lectura y otro para escritura.

6.2. Desarrollo de alternativas

A continuación se proponen 3 casos distintos para abordar el problema:

6.2.1. Mapa de bits

Un primer enfoque sería que el módulo ofuscador tuviera una tabla (mapa de bits), que indique para cada posición de memoria, si está ofuscada o no.

Si bien sería una solución relativamente fácil de implementar presenta un tamaño de la tabla excesivo.

En nuestro caso, sería un vector que hiciera referencia a cada palabra guardada en memoria y su estado, ofuscado o no.

6.2.2. Bit adicional en cada palabra de memoria

Bajando un nivel de abstracción, podemos identificar si una palabra en memoria está ofuscada cambiando la estructura de esta añadiendo un bit que indique su estado.

En el caso de las lecturas, se tendría en cuenta el valor de este bit mientras que, en las escrituras, la palabra se ofuscaría antes de volcar su contenido en la RAM y se fijaría este bit a 1.

Se trata de un enfoque rápido y versátil, pero supone un problema el cambiar la estructura de la memoria además de los accesos de tamaño inferior a la palabra.

6.2.3. Región de memoria ofuscada

Por ultimo, planteamos un tercer escenario en el que se define una region de memoria donde se aplicará la ofuscación.

En esta región se realizan los accesos siempre con ofuscación y se comprueba el contexto por los límites de esta, dirección de inicio y fin, así como la dirección de acceso.

No puede haber en ella variables inicializadas en tiempo de compilación, pues no se podrían leer en ejecución si no han sido previamente ofuscadas.

El principal inconveniente de esta tercera opción es que el compilador debe ser consciente de la región y también el tipo de variables que van a residir en esta zona, aunque por otro lado se trata de un enfoque relativamente fácil de implementar en hardware.

Probando con distintas directivas del compilador y los scripts del enlazador se puede conseguir ubicar las variables en la región adecuada.

6.3. Elección de la alternativa más convincente

Después de detallar la estructura interna del controlador de memoria en un apartado anterior y tomando en consideración las tres propuestas detalladas anteriormente llegamos a la siguiente conclusión:

- Teniendo en cuenta la dificultad de implementación en hardware ordenaríamos las tres propuestas de mayor a menor dificultad de esta manera:
    1. Bit adicional en cada palabra de memoria
    2. Región de memoria ofuscada
    3. Mapa de bits
- Identificando el tamaño de la memoria sobre el que actuaría el proceso de ofuscación de mayor a menor:
    1. Bit adicional en cada palabra de memoria (se podría ofuscar toda la memoria)
    2. Región de memoria ofuscada - Mapa de bits (solo una zona definida).

Por tanto, teniendo en cuenta ambos aspectos nos decantamos por la implementación de la zona de memoria por la versatilidad y la capacidad de poder configurar la que queramos con un tamaño también que predefinamos minimizando el impacto en la carga global de escrituras/lecturas desde el controlador a la memoria.

6.4. Ejecución del plan

Ofuscaremos el contenido de cada palabra utilizando la técnica del byte swapping, es decir intercambiando los bits byte por byte.

El ancho de palabra entrada-salida resultante del buffer ddr2buf.vhdl utilizado para las escrituras y las lecturas teniendo en cuenta los generics para la placa Nexys4ddr es de 32 bits.

Tomando como referencia un vector de 32 bits conteniendo la cadena "Hello Wo" y agrupando byte por byte su representación sería la siguiente:

DIGITO	HEX	PREOFUSCACION	POST OFUSCACION	HEX
H	48	01001000	00010010	12
e	65	01100101	10100110	A6
l	6c	01101100	00110110	36
l	6c	01101100	00110110	36
o	6f	01101111	11110110	F6
	20	00100000	00000100	04
W	57	01010111	11101010	EA
o	6f	01101111	11110110	F6

Utilizaremos este algoritmo ya que simplificamos la distinción entre lecturas y escrituras, es decir, se aplica el mismo algoritmo para ambas operaciones.

En un caso real se podría utilizar por ejemplo un algoritmo de clave simétrico con una clave generada de forma aleatoria en el arranque del sistema que solo el controlador de memoria conociese para cifrar las escrituras en el chip de memoria y descifrar las lecturas con la misma clave.

6.5. Toma de decisiones

Para encajar el desarrollo de la alternativa "Región de memoria ofuscada" hemos tenido en cuenta varios aspectos que detallamos a continuación.

6.5.1. Distinción señales AHB — RAM

Como se describe en la sección 'Estado del arte', para la ejecución de programas, partimos de un diseño ya implementado en formato .bit que se vuelca en la placa Nexys4DDR. Si bien se podría simular el diseño tal y como se indica en la documentación usando ModelSim por ejemplo, en este trabajo interactuamos con el procesador vía GRMON.

Esto permite lanzar directamente operaciones de lectura/escritura sobre la memoria usando distintos comandos. Teniendo en cuenta que estas operaciones se realizan en modo ráfaga usando dos buffers, para diagnóstico, vamos a extraer del diseño una señal de 32 bits que mediante un switch mostrará la salida de los dos buffers lectura-escritura.

La idea es que esta seña señal pueda ser visualizada y comparar con los datos mostrados en GRMON.

6.5.2. Visualización de los resultados

Para visualizar las dos señales antes mencionadas usaremos el display de 7 segmentos de la placa.

Este display cuenta con 8 dígitos cada uno de ellos representable mediante 4 bits. Por tanto, si tuviéramos que ajustar un tamaño de entrada a mostrar en el display de 7 segmentos este tendría un ancho de 32 bits.

Teniendo en cuenta que el acceso de menor tamaño que podemos lanzar desde GRMON es el acceso vía bus AMBA de 1 byte (8 bits) tenemos en consideración también el funcionamiento de la memoria a modo ráfaga.

Es decir, si queremos leer un byte ubicado en la posición 0x44000000 lanzamos el comando "memb 0x44000000 1", esta operación desencadenaría una ráfaga de tamaño 8 determinado por el generic burstlen, es decir, se leerian 8 posiciones de 1 byte cada una, en total 64 bits. La visualización en el display de 7 segmentos se haría de estos últimos 32 bits, suficientes para identificar las palabras ofuscadas de las no ofuscadas.

6.5.3. Tamaño de la palabra

Tal y cómo se indica en la documentación de GRMON, hay varios tipos de comandos para lectura y escritura en memoria. Estos comando realizan peticiones vía bus AHB al controlador de memoria.

- memb – Acceso de lectura con un ancho de palabra de 8 bits
- memh – Acceso de lectura con un ancho de palabra de 16 bits
- mem – Acceso de lectura con un ancho de palabra de 32 bits
- memd – Acceso de lectura con un ancho de palabra de 64 bits

De la misma manera tenemos comandos para la escritura:

- wmemb – Acceso de escritura con un ancho de palabra de 8 bits
- wmemh – Acceso de escritura con un ancho de palabra de 16 bits
- wmem – Acceso de escritura con un ancho de palabra de 32 bits
- wmemd – Acceso de escritura con un ancho de palabra de 64 bits

Por tanto, la estrategia para la modificación de los buffers de lectura escritura sería realizarla a nivel de byte ofuscando cada 8 bits del vector.

6.5.4. Filtrado por dirección de memoria

Dado que el programa lo descargamos en la placa con GRMON, no podemos habilitar la ofuscación de memoria completa ya que por ejemplo las instrucciones del programa no estarían ofuscadas.

Es por ello que tenemos que indicarle al ofuscador sobre qué zona de memoria actuar. El compilador tiene que conocer esta zona de memoria configurando correctamente un script para el enlazador y de esta manera evitar colocar en esta zona variables inicializadas en tiempo de compilación.

6.6. Planificación

Expuestas todas las consideraciones el desarrollo y modificación del diseño vamos a abordarlo consiguiendo estos hitos:

1. Obfuscación de la señal de salida dentro del buffer utilizado para lectura/escritura

2. En un nivel superior, en el controlador ddr2spax se habilitan dos señales para poder seleccionar las salidas lectura/escritura y la dirección de acceso, así como su desplazamiento. También se conectará una señal que habilita la ofuscación, inicialmente manual.

3. Con la señal manual, habilitaremos la ofuscación una vez lanzada la rutina **ddr2delay scan** y finalizada la carga del programa. Esto nos permitirá validar que las lecturas/escrituras se realizan correctamente.

4. Para terminar el diseño, eliminaremos la señal de entrada y la generaremos internamente filtrando las direcciones de memoria sobre las que actúa la obfuscación mediante el uso de generics y teniendo en cuenta la definición de esta zona en el compilador. Dentro de este proceso se generarán tres nuevas constantes configurables con el comando **make xconfig**.

5. Por último, comprobaremos los resultados cargando un programa que actue sobre una región que previamente definiremos en el diseño del procesador.

7. Diseño y requisitos

7.1. Diseño inicial de los componentes

A continuación, se muestra un diagrama de los componentes del controlador DDR2SPA y sus conexiones iniciales.

El controlador de memoria se conecta al bus AHB mediante su instanciación en fichero leon3mp.vhd. Modificaremos su definición para admitir la ofuscación de una zona de memoria definida anteriormente.

Uno de los puertos de entrada a destacar sería la estructura ahbsi que contiene la dirección de memoria solicitada (haddr) así como el tamaño (hsize).

Internamente consta de dos componentes principales, un componente que implementa la tecnología especifica de la capa física (`ddr2phy_wrap_cbd`) y un controlador que coordina las operaciones (`ddr2spax`).

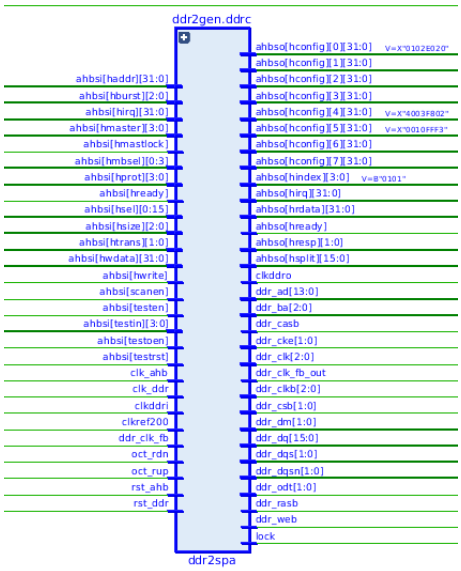

Figura 26: DDR2SPA Controlador de memoria conectado al bus AHB

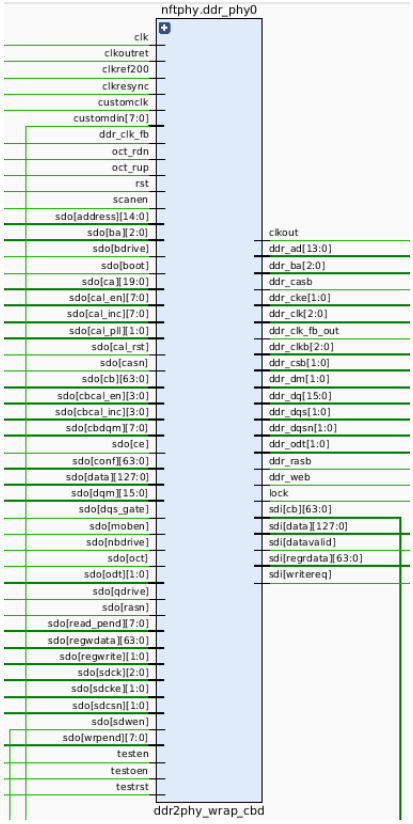

Figura 27: DDR2PHY

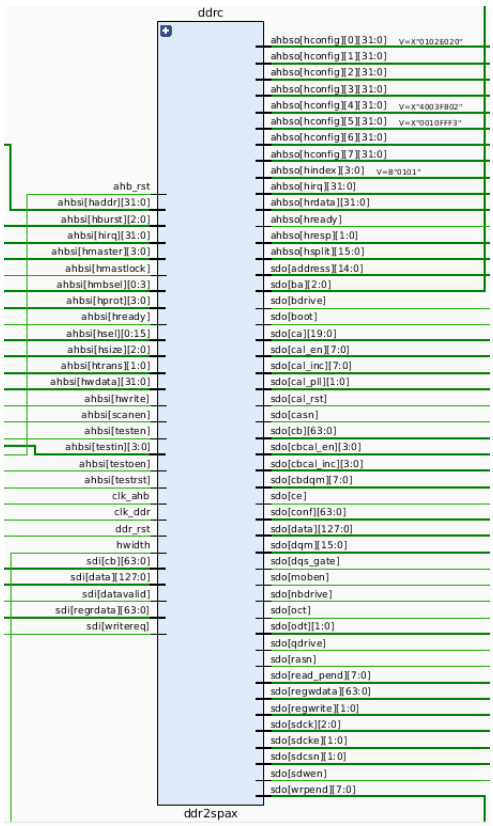

Figura 28: DDR2SPAX Controlador interno

Dentro del controlador **DDR2SPAX** tenemos cuatro componentes principales.

- wbuf Buffer de escritura (`ddr2buf.vhd`)
- rbuf Buffer de lectura (`ddr2buf.vhd`)
- ddrc Controlador interno para las coordinaciones de las operaciones de lectura escritura (`ddr2spax_ddr.vhd`)
- ahbc Controlador que hace de interfaz entre el controlador interno ddrc y el bus ahb externo (`ddr2spax_ahb.vhd`)

Nuestro propósito será habilitar la funcionalidad de la ofuscación en ambos buffers y dotar al controlador ddr2spax de capacidad para delimitar una zona de memoria sobre la que realizar la operación de ofuscación.

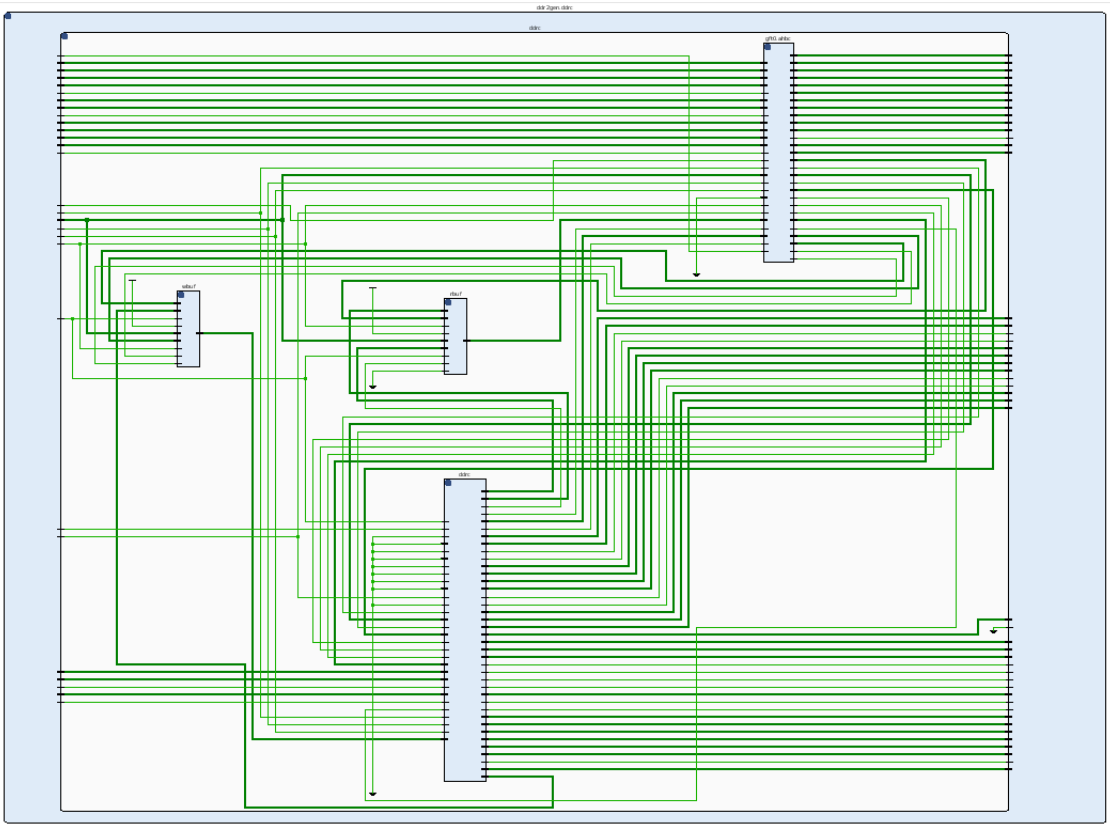

Figura 29: DDR2SPAX Estructura interna

7.2. Display de 7 segmentos

Tal y como se indica en la documentación de la placa Nexys4DDR [10], para mostrar un vector en el display de siete segmentos tenemos que mantener la señal de los anodos cierto tiempo activa y mostrando uno por uno cada dígito.

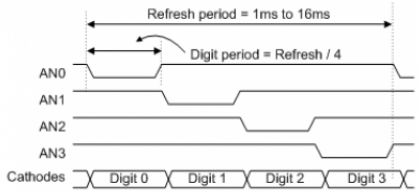

Figura 30: Timing display 7 segmentos

El código para dividir la frecuencia, asignar el valor por cada dígito, así como la conexión con la señal de 32 bits que nos llega desde el controlador de memoria es el siguiente:

```
--This process controls the counter that triggers the 7-segment
--to be incremented. It counts 100 and then resets.
timer_counter_process : process (sys_clk_i, clk_out)
begin
    if (rising_edge(sys_clk_i)) then
        cnt_sig <= cnt_sig + '1';
        if(cnt_sig = "1100100") then -- if = 100
             clk_out <= '1';
        else
             clk_out <= '0';
        end if;
    end if;
end process;

--This process increments the digit being displayed on the 7-segment
timer_inc_process : process (clk_out, contador)
variable tmp: std_logic_vector(7 downto 0);
begin
    if (rising_edge(clk_out)) then
        tmp := (others => '1');
        tmp(contador) := '0';
        an<=tmp;
        contador <= contador - 1;
        tmrVal<=ddr0_datafromphy((4*contador)+3 downto 4*contador);
    end if;
end process;

--This select statement encodes the value of tmrVal to the necessary
--cathode signals to display it on the 7-segment
with tmrVal select
    SSEG_CA <= "10000001" when "0000",
                  "11110011" when "0001",
                  "01001001" when "0010",
                  "01100001" when "0011",
                  "00110011" when "0100",
                  "00100101" when "0101",
                  "00000101" when "0110",
                  "11110001" when "0111",
                  "00000001" when "1000",
                  "00100001" when "1001",
                  "00010000" when "1010",
                  "00000110" when "1011",
                  "10001100" when "1100",
                  "01000010" when "1101",
                  "00001100" when "1110",
                  "00011100" when "1111",
                  "11111111" when others;
```

Se trata de una versión modificada del publicado en la web de Digilentic. [18]

7.3. DDR2BUF — Ofuscación de la entrada al buffer

Al trabajar con una memoria síncrona, los dos buffers de lectura/escritura tienen que ser capaces de poder ofuscar la señal de datos (32 bits).

Dado que internamente estos datos los guarda haciendo uso de un componente llamado `syncram_2p` que mantiene el valor en el tiempo a través de los distintos ciclos de reloj, vamos a seguir la estrategia de ofuscar la entrada y guardarlos ya ofuscados.
La instanciación de este componente se realiza con mediante la sentencia generate.

```
ramgen: for x in 0 to nrams-1 generate
    r: syncram_2p generic map (tech,abits,dbits,sepclk,wrfst,testen)
      port map (rclk => rclk,renable => renable,
                raddress => raddress((rabits-1) downto (rabits-abits)),
                dataout => do(x),wclk => wclk,write => we(x),
                waddress => waddress((wabits-1) downto (wabits-abits)),
                datain => di(x), testin => testin);
  end generate;
```

La señal que tenemos que ofuscar, di(x) se genera a partir del vector de entrada.

```
-- Generate vdi from datain
    for x in 0 to nrams-1 loop
      vdi(x) := datain(wdbits-(x mod wdratio)*dbits-1 downto wdbits-(x mod wdratio)*dbits-dbits);
    end loop;

di <= vdi;
```

Mediante dos nuevas señales internas, ofuscaremos la señal.

Por otra parte, para controlar en qué momento ofuscamos, habilitamos también una señal de entrada que controle la conexión de la salida ofuscada/no ofuscada.

```
vdio := (others => (others => '0'));
vdi_nonobfuscated:= (others => (others => '0'));

-- Generate vdi from datain, obfuscating each vector one by one
    for x in 0 to nrams-1 loop
      vdi_nonobfuscated(x) := datain(wdbits-(x mod wdratio)*dbits-1 downto wdbits-(x mod wdratio)*dbits
      for y in 0 to ((dbits-1)/8) loop
        for i in 0 to 7 loop
          vdio(x)(8*y+i) := vdi_nonobfuscated(x)(8*y+(7-i));
        end loop;
      end loop;
    end loop;

-- Connect syncram_2p di with vdi/vdio
   if obfuscate = '1' then
        di <= vdio;
    else
        di <= vdi;
    end if;
```

7.4. DDR2SPAX — Nuevas señales de control y visualización en el display de 7 segmentos

Teniendo la parte del buffer definida, subimos un nivel en la capa de hardware, el controlador ddr2spax.

En este, se instancian los dos buffers:

```
wbuf: ddr2buf
  generic map (tech => memtech, wabits => wbuf_wabits, wdbits => wbuf_wdbits,
               rabits => wbuf_rabits_r, rdbits => wbuf_rdbits,
               sepclk => 1, wrfst => ramwt, testen => scantest)
  port map ( rclk => clk_ddr, renable => vcc, raddress => wbraddr(wbuf_rabits_r-1 downto 0),
             dataout => wbrdata, wclk => clk_ahb, write => wbwrite,
             writebig => wbwritebig, waddress => wbwaddr, datain => wbwdata,
             obfuscate=> ofuscatedaddress, testin => ahbsi.testin);

rbuf: ddr2buf
  generic map (tech => memtech, wabits => rbuf_wabits, wdbits => rbuf_wdbits,
               rabits => rbuf_rabits, rdbits => rbuf_rdbits,
               sepclk => 1, wrfst => ramwt, testen => scantest)
  port map ( rclk => clk_ahb, renable => vcc, raddress => rbraddr,
             dataout => rbrdata,
             wclk => clk_ddr, write => rbwrite,
             writebig => '0', waddress => rbwaddr, datain => rbwdata,
             obfuscate=> ofuscatedaddress,  testin => ahbsi.testin);
```

Parte para ver la salida de ambos buffers, habilitaremos una señal de switch y sacaremos una señal de salida de 32 bits que muestre la salida del buffer de lectura o escritura según el valor de entrada del switch 0.

Para ver la dirección de memoria inicio y su desplazamiento solicitada por el controlador `ddr2spax_ahb` al controlador `ddr2spax_ddr` habilitamos el switch 15. A su vez, el switch 0 controlará si vemos la dirección inicio o dirección fin.

Exponemos el valor de la salida de 32 bits según el valor de los switches 0 y 15.

SW(15)	SW(0)	OUTPUT
0	0	request.endaddr (desplazamiento)
0	1	request.startaddr (direccion de memoria)
1	0	rbrdata (output buffer lectura)
1	1	wbrdata (output buffer escritura)

Por tanto, nuestro proceso combinacional dentro del controlador DDR2SPAX queda de la siguiente manera:

```
dataout_sel: process(ddr_dataout_sel, ddr_dataout_type)
    variable endaddress: std_logic_vector(31 downto 0);
    begin
      -- Buffer outputs
      if ddr_dataout_type = '1' then
        if ddr_dataout_sel = '1'  then
          --Write Buffer output
          ddr_dataout (31 downto 0) <= wbrdata (31 downto 0);
        else
          --Read Buffer output
```

```
          ddr_dataout (31 downto 0) <= rbrdata (31 downto 0);
        end if;
      else
      -- Addressing outpus
        -- Start address request
        if ddr_dataout_sel = '1'  then
          ddr_dataout <= request.startaddr;
        else
        -- End address request
          endaddress := (others => '0');
          endaddress(9 downto 0) := request.endaddr;
          ddr_dataout <= endaddress;
        end if;
      end if;
    end process;
```

7.5. Configurando la zona de memoria a ofuscar

7.5.1. Generación del script del enlazador

Para probar la ofuscación en la memoria de forma selectiva en una zona delimitada, vamos a generar un nuevo programa que guarda y modifica el valor de una cadena de texto en una zona específica de memoria.

Para ello construimos un fichero con extensión ld con el siguiente contenido:

```
SECTIONS
{
  .obfuscated 0x44000000 : {KEEP(*(.obfuscatedSection))}
}
```

7.5.2. Ejecutable que utiliza la zona de memoria definida

A continuación creamos un programa en c que utiliza la sección definida en el paso anterior:

```
#include <stdio.h>
#include <string.h>

char msg2[] __attribute__((section(".obfuscatedSection"))) = "1234567887654321";

int main()
{
        printf("Initial value -> %s\n", msg2);
        strncpy(msg2,"8765432112345678",16);
        printf("Updated value -> %s\n", msg2);
}
```

Compilamos el programa añadiendo el script del enlazador:

```
sparc-gaisler-elf-gcc holamundo.ld holamundo.c -o holamundo
```

Podemos verificar listando los símbolos desde los ficheros objeto generados:

```
sparc-gaisler-elf-nm holamundo | grep msg2
```

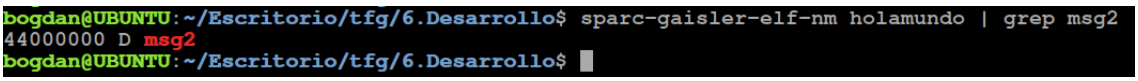

Figura 31: Listado de simbolos

Una vez compilado el programa, si lo volcamos en la placa podemos verificar la definición de esta nueva zona:

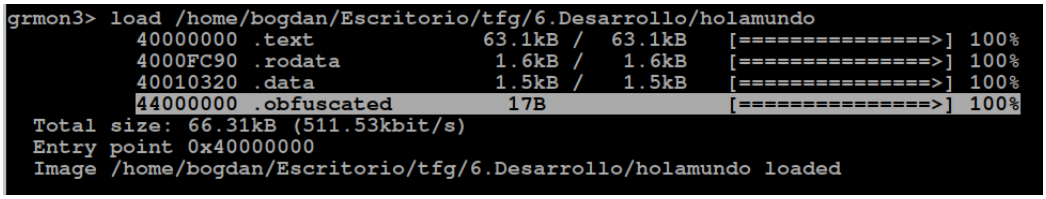

Figura 32: Load variable ofuscada

7.5.3. XCONFIG Generación de constantes para delimitar la zona de memoria ofuscada

Para que el hardware distinga las direcciones de memoria sobre las que se aplica la ofuscación, tenemos que definir las direcciones de inicio y fin de esta zona.

Haremos esto extendiendo la interfaz gráfica de **xconfig** para nuestra placa como se indica en el apartado 8.7 del manual GRLIB IP Library User's Manual. [11]

En la ruta `/lib/gaisler/ddr` editamos el fichero ddr2sp.in para incluir las nuevas opciones del menu.

```
bool 'Enable Memory Obfuscation' CONFIG_DDR2SP_OBFUSCATION
      if [ "$CONFIG_DDR2SP_OBFUSCATION" = "y"]; then
        hex 'Obfuscation start address' CONFIG_DDR2SP_OBFUSCATION_START 4000000
        hex 'Obfuscation end address' CONFIG_DDR2SP_OBFUSCATION_END 5000000
      fi
```

Con esta modificación al lanzar el comando **make xconfig** en el apartado del controlador de memoria mostramos tres nuevas opciones.

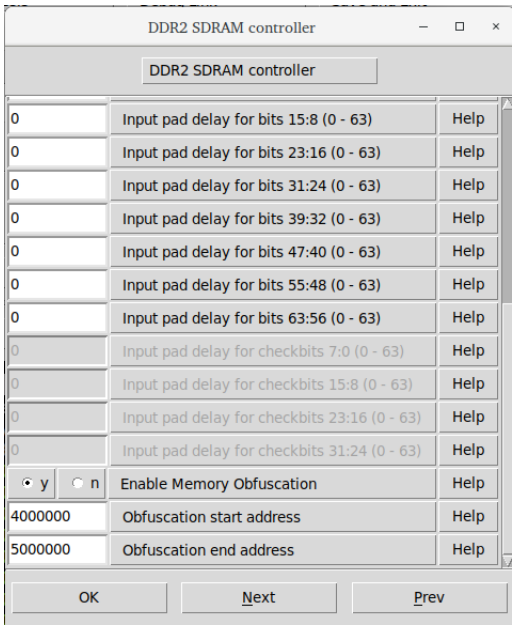

Figura 33: Opciones extendidas

Configuramos en cada apartado de ayuda el mensaje correspondiente. Esto lo hacemos en el fichero **ddr2sp.in.help** de la misma ruta añadiendo lo siguiente:

```
Enable Obfuscation
CONFIG_DDR2SP_OBFUSCATION
  Say Y here to enable obfuscation on DDR2 controller. This will obfuscate
  RAM data inside ddr2buf.vhd file reading/writing obfuscated data to memory
  for a defined zone.

Start Obfuscation Addr
CONFIG_DDR2SP_OBFUSCATION_START
  Start address for obfuscated data in memory

End Obfuscation Addr
CONFIG_DDR2SP_OBFUSCATION_END
  End address for obfuscated data in memory
```

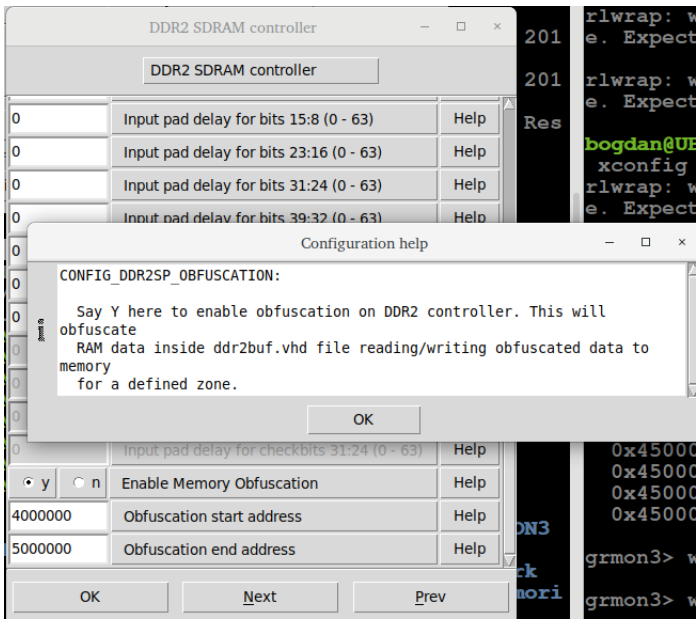

Figura 34: Ayuda definida para las nuevas opciones

Añadimos las nuevas constantes al fichero **ddr2sp.in.vhd** como sigue:

```
constant CFG_DDR2SP_OBFUSCATION : integer := CONFIG_DDR2SP_OBFUSCATION;
constant CFG_DDR2SP_OBFUSCATION_START : integer := 16#CONFIG_DDR2SP_OBFUSCATION_START#;
constant CFG_DDR2SP_OBFUSCATION_END : integer := 16#CONFIG_DDR2SP_OBFUSCATION_END#;
```

Por último en el fichero **ddr2sp.in.h** establecemos valores por defecto para estas constantes:

```
#ifndef CONFIG_DDR2SP_OBFUSCATION
#define CONFIG_DDR2SP_OBFUSCATION 0
#endif

#ifndef CONFIG_DDR2SP_OBFUSCATION_START
#define CONFIG_DDR2SP_OBFUSCATION_START 4000000
#endif

#ifndef CONFIG_DDR2SP_OBFUSCATION_END
#define CONFIG_DDR2SP_OBFUSCATION_END 5000000
#endif
```

Esto nos permitirá utilizar las tres constantes como generics en el código vhdl conectándolas hasta el nivel del controlador ddr2spax donde tenemos los dos buffers.

7.5.4. DDR2SPAX Generación de una nueva señal de entrada para el buffer

Teniendo las tres constantes en la instanciación del componente DDR2SPAX, creamos un nuevo proceso que controlará la señal **obfuscateaddress** de entrada a los buffers.

Para ello creamos un nuevo proceso que activará esta señal cuando la dirección de memoria motivo de la solicitud del bus ahb esté en la zona ofuscada.

```
evaluate_ofuscation: process(request.startaddr)
    variable tmp : integer;
    begin
    tmp := to_integer(unsigned(request.startaddr(31 downto 0)));
    if obfuscation=1 then
```

```
        if tmp >= obfuscationstartaddr and tmp<obfuscationendaddr then
          ofuscation_enabled <= '1';
          ofuscatedaddress <= '1';
        else
          ofuscation_enabled <= '0';
          ofuscatedaddress <= '0';
        end if;
    else
        ofuscatedaddress <= '0';
        ofuscation_enabled <= '0';
    end if;
    end process;
```

7.6. Validación de la ofuscación

7.6.1. Ejecución de un programa que utiliza la zona de memoria ofuscada

Para validar la ofuscación descargamos el programa en la placa y al ejecutarlo comprobamos la coherencia de los datos.

```
grmon3> load /home/bogdan/Escritorio/tfg/6.Desarrollo/holamundo
          40000000 .text             63.1kB /  63.1kB   [===============>] 100%
          4000FC90 .rodata            1.6kB /   1.6kB   [===============>] 100%
          40010320 .data              1.5kB /   1.5kB   [===============>] 100%
          44000000 .obfuscated         17B              [===============>] 100%
  Total size: 66.31kB (520.84kbit/s)
  Entry point 0x40000000
  Image /home/bogdan/Escritorio/tfg/6.Desarrollo/holamundo loaded

grmon3> run
Initial value -> 1234567887654321
Updated value -> 8765432112345678

  Program exited normally

grmon3> mem 0x44000000
  0x44000000  38373635  34333231  31323334  35363738    8765432112345678
  0x44000010  00633064  75900000  ffff0000  ffff0000    .c0du...........
  0x44000020  ffff0000  ffff0000  ffff0000  ffff0000    ................
  0x44000030  ffff0000  ffff0000  ffff0000  ffff0000    ................

grmon3>
```

Figura 35: Ejecución del programa dentro de la zona ofuscada

7.6.2. Escritura en memoria

Otra manera de validar que la ofuscación se produce en la memoria es realizar escrituras dentro y fuera de la zona comprobando la salida del buffer de escritura.

Usamos dos escrituras de ejemplo:

1. **wmem 0x44fffffc 0x696e** (escribe 'in' al final de la zona de memoria ofuscada)
2. **wmem 0x45000000 0x6f757400** (escribe 'out' en la posición 0x45000000, fuera de la zona de memoria ofuscada)

La validación consiste en activar lase señales sw(0) y sw(15) para ver la salida del buffer de escritura, es decir, los datos que se graban en la memoria.

En el primer caso, se observa el valor 0x00009676 que corresponde al valor ofuscado de 0x0000696e que escribimos.

```
grmon3> wmem 0x44fffffc 0x696e

grmon3> wmem 0x45000000 0x6f757400

grmon3> mem 0x44fffff0
  0x44fffff0  00000000  00000000  00000000  0000696e    ..............in
  0x45000000  6f757400  00000000  00000000  00000000    out.............
  0x45000010  00000000  00000000  ffff0000  ffff0000    ................
  0x45000020  ffff0000  ffff0000  ffff0000  ffff0000    ................

grmon3>
```

Figura 36: Escrituras dentro y fuera de la zona ofuscada

En el segundo caso, se observa el valor 0x6f757400, es decir, no se produce la ofuscación.

Se comprueba también el valor del led(7) estando encendido en el primer caso mientras que en el segundo se encuentra apagado.

7.7. DDR2SPA Diseño final

A continuación mostramos un diagrama del controlador ddr2spax a nivel interno haciendo énfasis en las señales creadas.

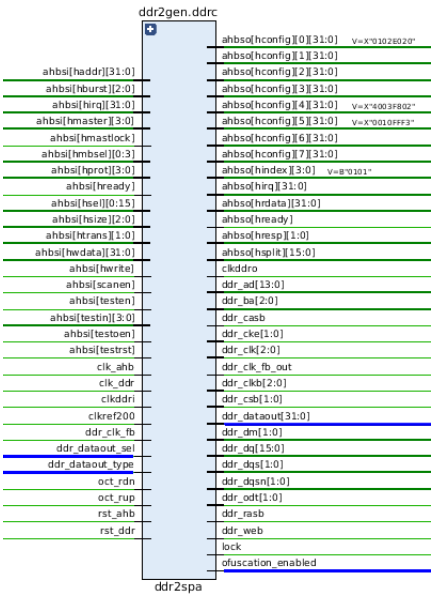

Figura 37: Diseño final DDR2SPA

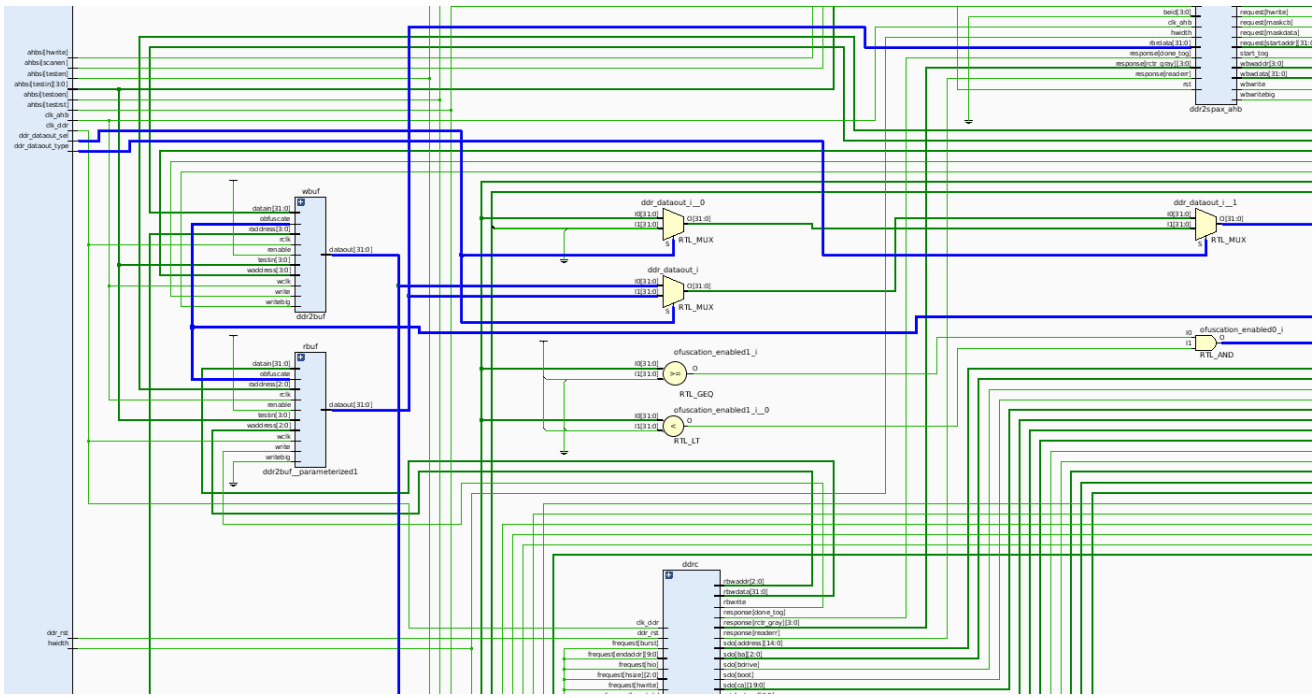

Figura 38: Diseño interno DDR2SPAX final

8. Resultados, discusión crítica y conclusiones

8.1. Resultados

Una vez finalizada la modificación de los distintos componentes del controlador DDR2SPA se ha procedido a realizar una serie de pruebas para confirmar la validación del proceso de ofuscación.

Inicialmente se ha desarrollado un programa que actuara sobre un área dentro de la zona de memoria definida como ofuscada. Volcándolo en la placa, comprobamos su correcta ejecución.

Posteriormente se han realizado distintas pruebas desde GRMON usando varios tipos de acceso a distintas direcciones de memoria verificando la coherencia de la información.

Una parte importante de la validación ha sido la utilización del display de siete segmentos para verificar la salida hacia la memoria RAM.

8.2. Conclusiones

Para llevar a cabo este trabajo se han tenido que superar varios hitos:

1. **Familiarización con el entorno de desarrollo.** Se han instalado varios componentes software detallando su configuración en el apartado **estado del arte**. Podemos enumerar la librería grlib de Cobham Gaisler, suite de Vivado, GRMON, compilador cruzado BCC y definición de los scripts para el enlazador entre otros.

2. **Familiarización con la arquitectura del procesador.** Para conseguir desarrollar la solución de ofuscación se han consultado varios documentos relacionados con la arquitectura de LEON3. El diseño del procesador, el bus AHB, el funcionamiento del

controlador DDR2SPA y su estructura interna han permitido obtener una trazabilidad de las señales involucradas y comprender el ciclo de vida de un programa desarrollado en c desde su compilación hasta su ejecución.

3. **Depuración y visualización de señales externas.** La utilización de los switches, leds y display de 7 segmentos externos han permitido visualizar los resultados en cada fase, así como la realización de las pruebas que pudieran demostrar que la ofuscación se estaba realizando correctamente.

4. **Documentación.** Finalmente resaltar la importancia de la documentación en cualquier proceso, desarrollo, trabajo. Latex es un sistema de composición de textos muy versátil, un universo descubierto a lo largo del desarrollo de este trabajo y que ha permitido elaborar esta memoria teniendo en cuenta los distintos elementos que la componen.

8.3. Futuras líneas de investigación

Dentro del buffer usamos un algoritmo sencillo para ofuscar la información. A nivel interno se traduce en un sistema combinacional (intercambio de bits a nivel de byte).

En un caso real, una memoria resistiva con una zona de memoria ofuscada, esta información seria fácilmente recuperable con un análisis criptográfico sencillo.

Por ello, una posible continuidad de este trabajo podría ser la aplicación de un algoritmo de cifrado simétrico con una clave generada de forma aleatoria en la inicialización del controlador DDR2SPA.

Para conseguir mayor aleatoriedad en la generación de la clave, se podrían usar sensores externos, en nuestro caso la placa Nexys4DDR cuenta con accelerometro o sensor de temperatura.

Al ser un algoritmo de cifrado en varias etapas se trataría de un sistema secuencial, funcionando con un reloj distinto, con una frecuencia mayor a la del controlador de memoria.

Teniendo en cuenta que el diseño del LEON3 el reloj interno del controlador DDR2SPA es distinto al reloj del bus AHB habría que tener en cuenta la penalización del cifrado en operaciones sobre zonas de memoria ofuscadas controlando el riesgo.

Ejemplos de algoritmos de cifrado simétricos podrían ser TWOFISH o CAST.

9. Results, critical discusion and conclusions

9.1. Results

Once the modification of the different components of the DDR2SPA controller is completed, has proceeded to perform a series of tests to confirm the validation of the obfuscation.

Initially, a program has been developed that will act on an area within the zone of memory defined as obfuscated. Turning it onto the plate, we check its correct execution.

Subsequently, different tests have been carried out from GRMON using various types of access to different memory addresses verifying the coherence of the information.

An important part of the validation has been the use of the seven segment display to verify the output to RAM.

9.2. Conclusion

To carry out this work, several milestones had to be passed:

1. **Familiarization with the development environment.** Multiple components have been installed software detailing its configuration in the state of the art section. We can list Cobham Gaisler's grlib library, Vivado suite, GRMON, compiler cross BCC and definition of the scripts for the linker among others.

2. **Familiarization with the processor architecture.** In order to develop the obfuscation solution, several documents related to the LEON3 architecture. The design of the processor, the AHB bus, the operation of the DDR2SPA controller and its internal structure have allowed for traceability of the signals involved and understand the life cycle of a developed program in c from compilation to execution.

3. **Debugging and visualization of external signals.** The use of switches, LEDs and external 7-segment display have allowed the results to be viewed in each phase, as well as the performance of tests that could demonstrate that obfuscation is was performing correctly.

4. **Documentation.** Finally, highlight the importance of documentation in any process, development, work. Latex is a very versatile text composition system, a universe discovered throughout the development of this work and that has allowed prepare this report taking into account the different elements that compose it.

9.3. Future Work

Inside the buffer we use a simple algorithm to blur the information. Internally, it translates into a combinational system (exchange of bits at the byte level).

In a real case, a resistive memory with an obfuscated memory area, this information would be easily retrievable with a simple cryptographic analysis.

Therefore, a possible continuity of this work could be the application of a symmetric encryption algorithm with a randomly generated key upon initialization of the DDR2SPA driver.

To achieve greater randomness in the generation of the key, external sensors could be used, in our case the Nexys4DDR board has an accelerometer or temperature sensor.

Being a multi-stage encryption algorithm, it would be a sequential system, operating with a different clock, with a higher frequency than the memory controller.

Taking into account that the design of the LEON3, the internal clock of the DDR2SPA controller is different from the clock of the AHB bus, it would be necessary to take into

account the encryption penalty in operations on obfuscated memory areas, controlling the risk. Examples of symmetric encryption algorithms could be TWOFISH or CAST.

Bibliografia

Referencias

[1] *First Draft of a Report on the EDVAC*, Michael D. Godfrey - webarchive.org

[2] *Memristor The missing circuit element*, Leon O. Chua - ieeexplore.ieee.org

[3] *The missing memristor found*, Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart and R. Stanley Williams - nature.com

[4] *Fundamental Issues and Problems in the Realization of Memristors*, Paul Meuffels, Rohit Soni - arxiv.org

[5] *Nanotechnology-based next generation memory nears mass production*, Jim Lewis - foresight.org

[6] *ReRAM Advantages* - crossbar-inc.com

[7] *White Paper AMD Memory Encryption*, David Kaplan, Jeremy Powell, Tom Woller - amd.com

[8] *Intel Architecture Memory Encryption Technologies Specification*, Ref. 336907-002US - intel.com

[9] *MT47Hxx(x)M4/8/16 Datasheet* - digikey.es

[10] *Nexys 4 DDR Reference Manual* - digilentinc.com

[11] *GRLIB IP Library User's Manual* - gaisler.com

[12] *Vivado Archive* - xilinx.com

[13] *Download GRMON* - gaisler.com

[14] *Download Cross Compiler System* - gaisler.com

[15] *GRMON Manual* - gaisler.com

[16] *GRLIB IP Cores Manual* - gaisler.com

[17] *Adept 2* - digilentinc.com

[18] *Nexys 4 DDR GPIO Demo* - digilentinc.com

[19] *GRLIB code used for this project* - github.com