CONTROLADOR HW DE MEMORIAS SPI, I2C Y PARALELAS IMPLEMENTADO EN FPGA

SPI, I2C AND PARARELL MEMORY HARDWARE CONTROLLER IMPLEMENTED ON FPGA

TRABAJO FIN DE GRADO
CURSO 2023-2024

AUTOR
JAVIER FERNÁNDEZ GAMO

DIRECTORES
HORTENSIA MECHA LÓPEZ Y MOHAMMADREZA REZAEI

GRADO EN INGENIERÍA DE COMPUTADORES
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

AUTOR
JAVIER FERNÁNDEZ GAMO

DIRECTORES
HORTENSIA MECHA LÓPEZ, MOHAMMADREZA REZAEI
CONVOCATORIA: SEPTIEMBRE 2024

GRADO EN INGENIERÍA DE COMPUTADORES
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

DÍA 13 DE SEPTIEMBRE DE 2024

Agradecimientos

Quiero agradecer a todas las personas que han contribuido a la realización de este Trabajo de Fin de Grado.

En primer lugar, quiero agradecer a mis tutores Hortensia y Moh por todo el apoyo, guía y consejos que me han brindado durante toda la realización de este trabajo.

Agradezco también a todos los profesores de la Facultad de Informática por los conocimientos impartidos, que me han servido para la realización de este proyecto.

A mis compañeros, por el apoyo durante la elaboración de este trabajo y su disponibilidad para discutir ideas.

A mi familia, por su apoyo y comprensión durante los momentos de mayor estrés.

Finalmente, a todas las personas que han ayudado con sus conocimientos, tiempo y apoyo a la realización del TFG. A todos, muchas gracias.

Resumen

Controlador hw de memorias SPI, I2C y paralelas implementado en FPGA.

La idea de este proyecto surgió a partir del concepto de New Space, donde cada vez más centros de investigación, centros educativos y empresas están enviando al espacio numerosos satélites, en muchos casos con poco presupuesto, lo que implica que tengan que utilizar materiales de bajo coste. Por esta razón, se hace uso de dispositivos COTS (*Commercial Off-The-Shelf*), los cuales son dispositivos que no garantizan la protección frente a la radiación, ya que no han sido endurecidos, lo que permite que su coste sea más bajo. Para asegurar que estos dispositivos puedan usarse en los entornos a los que se van a enviar es necesario validar que los COTS van a cumplir unos requisitos, en este caso su resistencia a la radicación máxima que se prevé que vaya a recibir, es decir, medir su sección eficaz frente a distintas fuentes de radicación como electrones, protones, neutrones e iones pesados. Sabiendo las cantidades de radiación en las posibles órbitas de estos dispositivos, y conociendo su sección eficaz, se podrá concluir si son lo suficientemente seguros o no.

Para calcular la sección eficaz de un dispositivo es necesario probar estos dispositivos (como por ejemplo memorias COTS). Las pruebas consisten en radiar las memorias con distintos tipos de partículas y comprobar si pueden mantener los datos, o bien se producen alteraciones de los mismos, lo que se suele llamar SEU (*Single even upset*), que es la alteración producida por una única partícula. Estos mismos se dividen en SBU (*Sigle Bit Upset*) si solo altera un bit de memoria o MCU (*Multiple Cell Upset*) si altera el contenido de varias celdas. A partir del recuento de SBUs y MCUs se calcula la sección eficaz del dispositivo, que es la medida de su vulnerabilidad frente a radiación.

Debido a esto surge la necesidad de realizar un sistema capaz de leer y escribir en distintos tipos de memorias para poder llevar acabo estos experimentos.

Palabras clave

Controlador de memoria, FPGA, VHDL, I2C, SPI, QSPI, Paralelas.

Abstract

SPI, I2C and parallel Memory hardware controller implemented on FPGA.

The idea for this project came from the concept of New Space, where each day more research centers, educational institutions, and companies are sending a lot of low-cost satellites into space, using COTS (Commercial Off-The-Shelf) devices, which are not guaranteed to be radiation-protected, allowing their cost to be lower. To ensure that these devices can be used in the environments they will be sent to, it’s necessary to validate that the COTS will be resistant enough to the maximum radiation they are expected to receive. For this reason, it is necessary to test these devices (for example COTS memories). This is going to be done by irradiating the memories and see if they can keep the data, understand how radiation affects them and determinate how much they can resist the radiation effects. This involves measuring their effective cross-section against various radiation sources such as electrons, protons, neutrons and heavy ions. Knowing the radiation levels in the possible orbits of these devices and understanding their effective cross-section, we can determine if they are safety enough.

To calculate the effective cross-section of a device, it is necessary to test these devices (such as COTS memories). The tests involve irradiating the memories with different types of particles and verifying if they can maintain the data or not, which is commonly known as SEU (Single Event Upset), an alteration caused by a single particle. These can be further classified as SBU (Single Bit Upset) if only one memory bit is changed or MCU (Multiple Cell Upset) if the contents of multiple cells are affected. From the number of SBUs and MCUs, the effective cross-section of the device is calculated, which measures its vulnerability to radiation.

By this, there is a need to develop a system capable of reading and writing on different types of memories to carry out these experiments.

Keywords

Memory controller, FPGA, VHDL, I2C, SPI, QSPI, Parallel.

Índice de contenidos

Tabla de contenido

1. Objetivos 7
2. Desarrollo y estructura del proyecto 8
3. Interfaz de usuario 21
4. Estructura del controlador de memorias implementado en HW. 25
5. Protocolos Utilizados 29
5.1. SPI 29
5.1.1. Comunicación específica para la memoria SPI 30
5.1.2. Dificultades encontradas 31
5.1.3. Modulo VHDL SPI: 31
5.1.4 Resultados 33
5.2. QSPI 35
5.2.1. Comunicación específica para la memoria QSPI utilizada 35
5.2.2. Errores encontrados: 37
5.2.3. Modulo VHDL QSPI 37
5.2.4 Resultados 38
5.3. Protocolo paralelo 40
5.3.1. Comunicación específica para la memoria Paralela utilizada 41
5.3.2. Errores encontrados 41
5.3.3. Módulo VHDL protocolo paralelo 41
5.3.4 Resultados 42
5.4. I2C 44
5.4.1 Comunicación específica para la memoria I2C utilizada 44
5.4.2. Errores encontrados 45
5.4.3. Modulo VHDL I2C 45
5.4.4 Resultados 46
6. Conclusiones y trabajo futuro 47
7.Bibliografía 48

1. Objetivos

La finalidad de este TFG es implementar un controlador de memorias en una FPGA, concretamente la Basys 3 usando las herramientas de Xilinx, tanto Vivado como SDK. Las memorias estarán conectadas mediante cables a los pines I/O de la FPGA. Este controlador de memorias será capaz de escribir y leer el dato seleccionado en una sola posición de memoria determinada, previamente indicada o en todo su rango de posiciones.

Los protocolos de comunicación implementados serán SPI, QSPI, Paralela e I2C. Vamos a utilizar la memoria APS6404L-3SQR QSPI PSRAM ([11], s.f.), para los protocolos SPI y QSPI, la memoria CY14B101J ([13], s.f.), para el protocolo I2C y la memoria CY62167DV30 ([12], s.f.), para el protocolo paralelo, con un total de tres memorias distintas.

Para su uso se implementará una interfaz en C, que a través de un interfaz serie permitirá la entrada/salida de los datos y parámetros deseados, como el tipo de protocolo a usar, la velocidad, etc.

La conexión con la FPGA se realizará a través del puerto serie, Uart, de la placa y usando un *hyperterminal* para visualizarlo en nuestro ordenador.

2. Desarrollo y estructura del proyecto

El sistema de control de las memorias va a ser un sistema empotrado implementado íntegramente en una FPGA tipo Basys3, que se muestra en la imagen 1. Para la integración de todo el sistema hacemos uso del *softcore* MicroBlaze para que actúe como un procesador del sistema, donde ejecutaremos una interfaz de usuario que se comunique con nuestro hardware en VHDL.

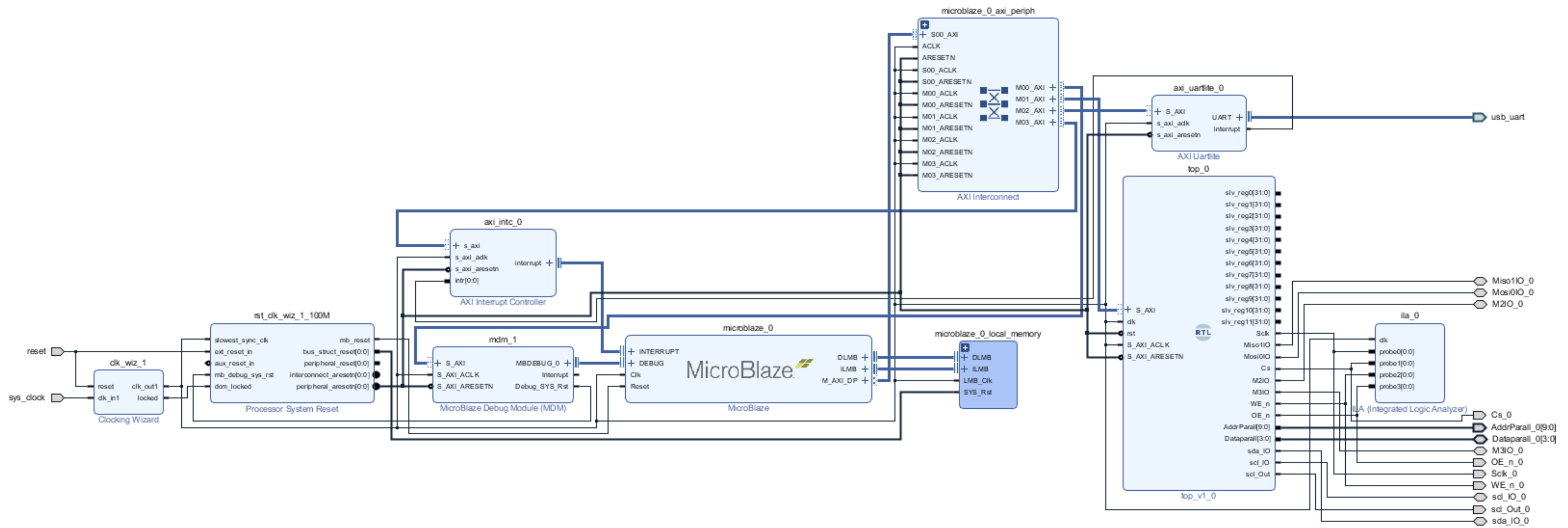

Imagen 1 sistema empotrado

Para la comunicación entre MicroBlaze y el VHDL se ha implementado el protocolo AXI Lite ([1], s.f.), que hace uso de doce registros. Así desde MicroBlaze podemos realizar escrituras y lecturas de los registros de forma rápida y sencilla usando los canales de escritura y lectura del AXI Lite cuando sean necesarios. Estos registros forman parte de nuestro controlador HW.

Como podemos ver en las imágenes 2 y 3 los canales de comunicación de AXI Lite son:

Canales de escritura:

AWADDR: Dirección del registro al que se desea escribir.

WDATA: Datos que se desean escribir.

WVALID/WREADY: Señales de validación y preparación para coordinar la escritura.

BRESP/BVALID/BREADY: Señales de respuesta de escritura.

Canales de lectura:

ARADDR: Dirección del registro del que se desea leer.

RDATA: Datos leídos del registro.

RVALID/RREADY: Señales de validación y preparación para coordinar la lectura.

El módulo que maneja las señales de AXI Lite que llegan desde MicroBlaze es myipreg_v1_0_S00_AXI, explicado más adelante.

Las señales *pslv_regX* contienen el valor del registro, lo que nos permite acceder a él en todo momento desde nuestro código VHDL.

Las ventajas de AXI Lite son:

Simplicidad: AXI Lite es más sencillo que AXI completo, lo que permite reducir la complejidad de la implementación.

Baja Latencia: Nos permite realizar accesos rápidos, asegurando una comunicación eficiente.

Fácil integración: La interfaz estándar facilita la integración entre el software en MicroBlaze y el hardware en VHDL.

```vhdl
entity myipreg_v1_0_S00_AXI is
    generic (
        -- Width of S_AXI data bus
        C_S_AXI_DATA_WIDTH  : integer   := 32;
        -- Width of S_AXI address bus
        C_S_AXI_ADDR_WIDTH  : integer   := 6
    );
    port (

        -- Global Clock Signal
        S_AXI_ACLK  : in std_logic;
        -- Global Reset Signal. This Signal is Active LOW
        S_AXI_ARESETN   : in std_logic;
        -- Write address (issued by master, acceped by Slave)
        S_AXI_AWADDR    : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        -- Write channel Protection type. This signal indicates the
            -- privilege and security level of the transaction, and whether
            -- the transaction is a data access or an instruction access.
        S_AXI_AWPROT    : in std_logic_vector(2 downto 0);
        -- Write address valid. This signal indicates that the master signaling
            -- valid write address and control information.
        S_AXI_AWVALID   : in std_logic;
        -- Write address ready. This signal indicates that the slave is ready
            -- to accept an address and associated control signals.
        S_AXI_AWREADY   : out std_logic;
        -- Write data (issued by master, acceped by Slave)
        S_AXI_WDATA : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        -- Write strobes. This signal indicates which byte lanes hold
            -- valid data. There is one write strobe bit for each eight
            -- bits of the write data bus.
        S_AXI_WSTRB : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto 0);
        -- Write valid. This signal indicates that valid write
            -- data and strobes are available.
        S_AXI_WVALID    : in std_logic;
        -- Write ready. This signal indicates that the slave
            -- can accept the write data.
        S_AXI_WREADY    : out std_logic;
        -- Write response. This signal indicates the status
            -- of the write transaction.
        S_AXI_BRESP : out std_logic_vector(1 downto 0);
        -- Write response valid. This signal indicates that the channel
            -- is signaling a valid write response.
```

Imagen 2 - VHDL de la entity myipreg_v1_0_S00_AXI 1

```
        S_AXI_BVALID    : out std_logic;
        -- Response ready. This signal indicates that the master
            -- can accept a write response.
        S_AXI_BREADY    : in std_logic;
        -- Read address (issued by master, acceped by Slave)
        S_AXI_ARADDR    : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        -- Protection type. This signal indicates the privilege
            -- and security level of the transaction, and whether the
            -- transaction is a data access or an instruction access.
        S_AXI_ARPROT    : in std_logic_vector(2 downto 0);
        -- Read address valid. This signal indicates that the channel
            -- is signaling valid read address and control information.
        S_AXI_ARVALID   : in std_logic;
        -- Read address ready. This signal indicates that the slave is
            -- ready to accept an address and associated control signals.
        S_AXI_ARREADY   : out std_logic;
        -- Read data (issued by slave)
        S_AXI_RDATA : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        -- Read response. This signal indicates the status of the
            -- read transfer.
        S_AXI_RRESP : out std_logic_vector(1 downto 0);
        -- Read valid. This signal indicates that the channel is
            -- signaling the required read data.
        S_AXI_RVALID    : out std_logic;
        -- Read ready. This signal indicates that the master can
            -- accept the read data and response information.
        S_AXI_RREADY    : in std_logic;

        pslv_reg0       : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg1       : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg2       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg3       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg4       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg5       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg6       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg7       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg8       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg9       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg10      : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        pslv_reg11      : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0)

    );
end myipreg_v1_0_S00_AXI;
```

Imagen 3 - VHDL de la entity myipreg_v1_0_S00_AXI 2

Los distintos módulos del sistema empotrado son:

MicroBlaze: Procesador *softcore*, que implementa una CPU en la FPGA y será el encargado de ejecutar el software. Imagen 4.

Imagen 4 – Modulo MicroBlaze

RTL: Es la parte donde está implementado el controlador de memoria, donde podemos observar las salidas que acabarán en el *pinout* de la placa y en el ILA (*Integrate Logic Analyzer*) como podemos ver en la imagen 5.

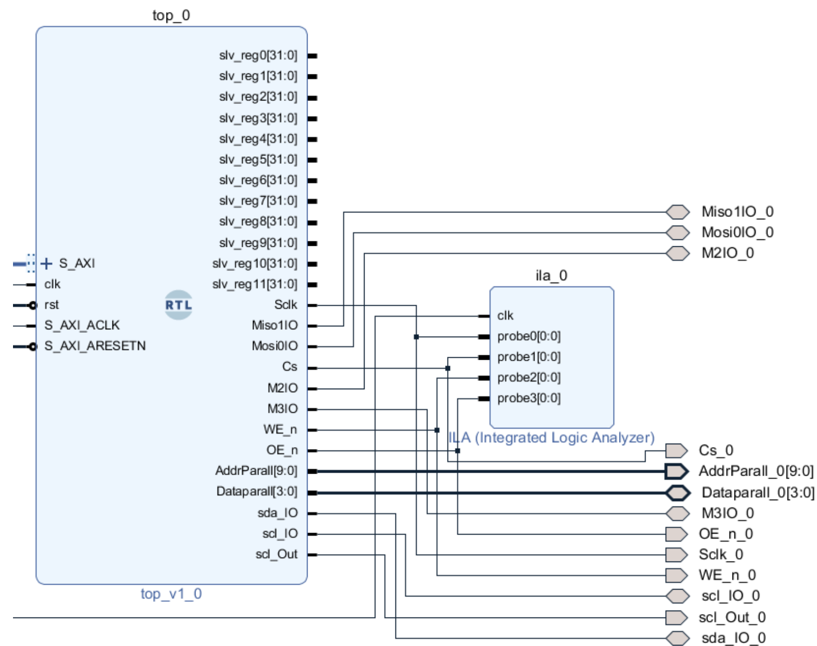

Imagen 5 – Modulo RTL e ILA

ILA: Para la depuración se hace uso de un ILA *(Integrate Logic Analizer*) que podemos ver en la imagen 5, que nos permite añadir *triggers* para ver el valor de las señales conectadas a este, en el momento indicado por el *trigger* y poder descubrir errores que ocurran durante la ejecución del sistema.

En la Imagen 6 podemos ver un ejemplo del funcionamiento del ILA, donde le pusimos como trigger el valor de *CS* a cero y podemos ver los ciclos de reloj enviados por la señal *SCLK* del protocolo SPI.

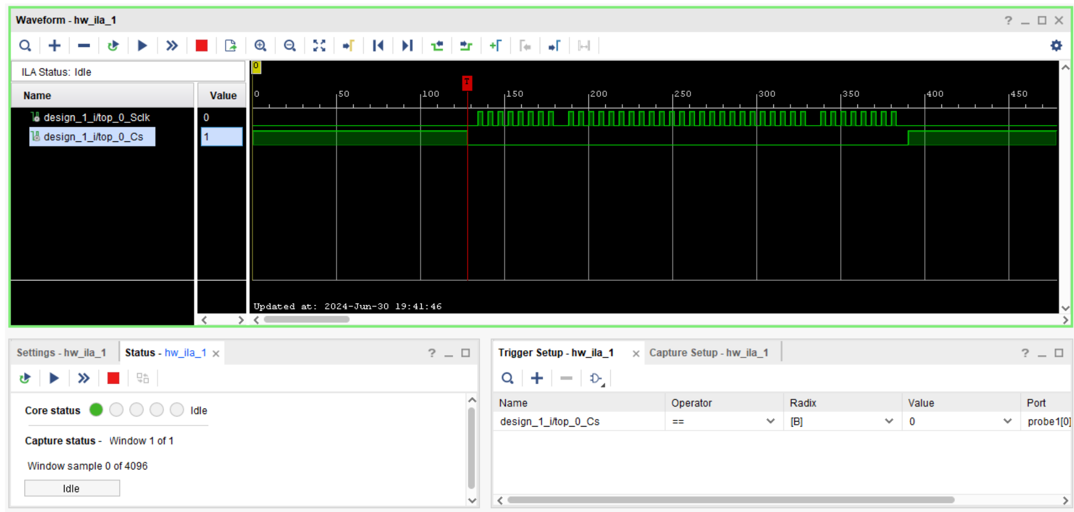

Imagen 6 - Uso de ILA

Clocking Wizard: Es el módulo encargado de generar las señales de reloj que usaremos en el sistema, necesarias para sincronizar todos los componentes. El interfaz de este módulo puede verse en la imagen 7.

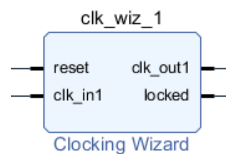

Imagen 7 – Clocking Wizard

Processor System Reset: Es el módulo encargado de generar las señales de *reset* que usaremos en el sistema necesarias para sincronizar todos los componentes. Su interfaz puede verse en la imagen 8.

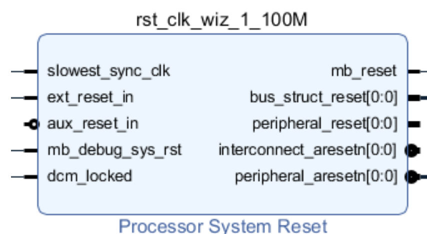

Imagen 8 – Processor System Reset

AXI Interrupt Controller: Controlador de interrupciones AXI, gestiona las interrupciones y se las envía a MicroBlaze. Su interfaz puede verse en la imagen 9.

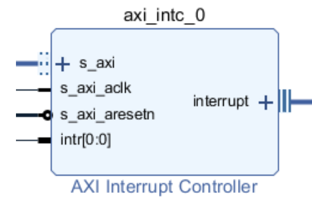

Imagen 9 – AXI Interrupt Controller

MicroBlaze Debug Module: Módulo que facilita la depuración del procesador permitiendo puntos de interrupción y ejecución paso a paso. Su interfaz puede verse en la imagen 10.

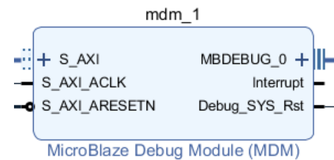

Imagen 10 – MicroBlaze Debug Module

MicroBlaze Local Memory: Modulo que proporciona almacenamiento RAM para MicroBlaze, imagen 11.

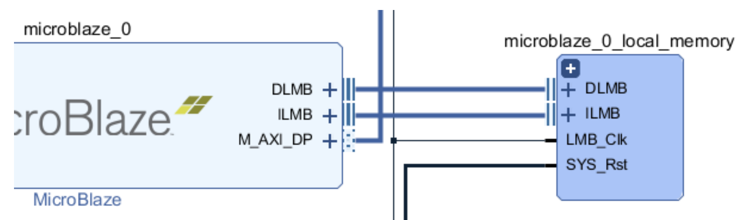

Imagen 11 – MicroBlaze Local Memory

Axi Interconnect: Modulo que se encarga de ser el enlace entre MicroBlaze y los periféricos AXI, imagen 12.

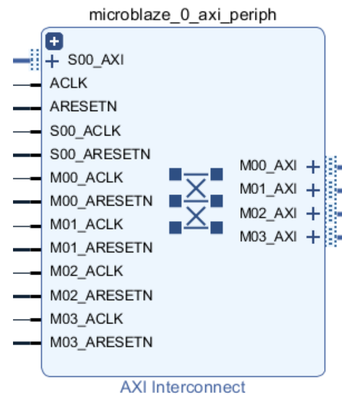

Imagen 12 – AXI Interconnect

AXI Uartlite: Módulo que proporciona una interfaz UART para la comunicación serie, imagen 13.

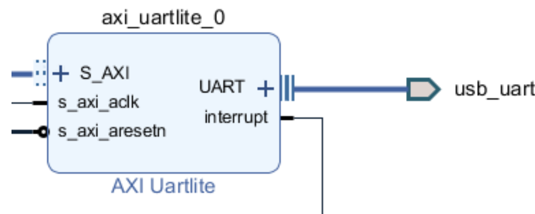

Imagen 13 – AXI Uartlite

Para la comunicación con estos registros, implementamos un programa en C que asigna una dirección en memoria a cada uno de ellos, previamente definida en nuestro proyecto de vivado. Se configuran de la siguiente manera:

Definimos la dirección donde se encuentra nuestro RTL.

#define AXI_REG_BASE_ADDR 0x44A00000

top_0	S_AXI	reg0	0x44A0_0000	64K	0x44A0_FFFF

Calculamos el offset de cada registro en este caso es de cuatro bytes.

#define OFFSET_REG0 0

#define OFFSET_REG1 4

#define OFFSET_REG2 8

#define OFFSET_REG3 12

```
#define OFFSET_REG4 16
#define OFFSET_REG5 20
#define OFFSET_REG6 24
#define OFFSET_REG7 28
#define OFFSET_REG8 32
#define OFFSET_REG9 36
#define OFFSET_REG10 40
#define OFFSET_REG11 44
```

Le asignamos a cada registro su dirección.

```
#define CERO_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG0)
#define UNO_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG1)
#define DOS_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG2)
#define TRES_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG3)
#define CUATRO_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG4)
#define CINCO_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG5)
#define SEIS_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG6)
#define SIETE_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG7)
#define OCHO_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG8)
#define NUEVE_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG9)
#define DIEZ_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG10)
#define ONCE_REG_ADDR (AXI_REG_BASE_ADDR + OFFSET_REG11)

uint32_t* ptaddrcero = CERO_REG_ADDR;
uint32_t* ptaddruno = UNO_REG_ADDR;
uint32_t* ptaddrdos = DOS_REG_ADDR;
uint32_t* ptaddrtres = TRES_REG_ADDR;
uint32_t* ptaddrcuatro = CUATRO_REG_ADDR;
uint32_t* ptaddrcinco = CINCO_REG_ADDR;
uint32_t* ptaddrseis = SEIS_REG_ADDR;
uint32_t* ptaddrsiete = SIETE_REG_ADDR;
uint32_t* ptaddrocho = OCHO_REG_ADDR;
uint32_t* ptaddrnueve = NUEVE_REG_ADDR;
```

```c
uint32_t* ptaddrdiez = DIEZ_REG_ADDR;
uint32_t* ptaddronce = ONCE_REG_ADDR;
```

Por último, escribimos el valor que necesitemos en ellos. El siguiente ejemplo correspondiente a la inicialización de nuestro programa en C, está asignando los valores 0 a los registros uno, dos, tres, cuatro y nueve, el valor uno a los registros ocho, diez y once y el valor dos al registro cinco.

```c
*ptaddruno = 0;
*ptaddrdos = 0;
*ptaddrtres = 0;
 *ptaddrcuatro = 0;
*ptaddrcinco = 2;
*ptaddrocho = 1;
*ptaddrnueve = 0;
*ptaddrdiez = 1;
*ptaddronce = 1;
```

La función de cada uno de los doce registros es la siguiente:

0- (*End*) Nuestro código en VHDL escribirá en este registro un "1" marcando el final del proceso de lectura o escritura. El programa en C esperará hasta que este valor sea uno para mostrar el dato leído o indicar que la escritura ha acabado.
1- (*Dato leído*) En este registro el código en VHDL escribirá el dato leído de la dirección de memoria proporcionada, de tal forma que cuando el código en C reciba el final de una lectura, leerá este registro para poder mostrar el valor leído.
2- (*Start*) Este registro estará inicialmente con un valor de 0 y MicroBlaze escribirá un 1 cuando deba empezarse la operación.
3- (*Dirección*) En este registro MicroBlaze escribirá la dirección en la que se desea hacer la operación.
4- (*Dato a escribir*) En este registro MicroBlaze introducirá el dato que se desea escribir en la memoria.
5- (*Velocidad*) En este registro se escribirá desde MicroBlaze los diferentes valores preconfigurados para elegir a qué velocidad queremos realizar las operaciones de la memoria.
6- (Sin uso)

7- (Sin uso)

8- (*Protocolo*) En este registro MicroBlaze escribirá el protocolo que se desea utilizar, indicando con un 1 el uso de SPI, un 2 el uso de QSPI, un 3 el uso de I2C y un 4 el uso de paralelo.

9- (*Dato Patrón*) En este registro cada vez que se escriba en modo patrón se guardará el dato que se va a escribir.

10- (*Modo*) En este registro se escribirá desde MicroBlaze "1" si se desea hacer la operación en modo solo una dirección o "2" en todas las posiciones de la memoria (limitadas a 255 en esta implementación).

11- (Operación) En este registro MicroBlaze escribirá "1" si se desea realizar la operación de escritura o "2" si se desea realizar la operación de lectura de las memorias.

Estos doce registros son de 32 bits debido a que la interfaz AXI Lite utiliza una arquitectura de 32 o 64 bits. Al tratarse de pocos registros no supone una sobrecarga excesiva para la implementación, permitiéndonos fácilmente ampliar el rango de direcciones utilizado de las memorias.

La implementación del modo patrón donde lee y escribe en las primeras 255 posiciones de la memoria, se ha implementado mediante un bucle *for* en software, ejecutando 255 veces la operación de escribir o leer cambiando la dirección.

En la imagen 14, podemos ver la conexión de las memorias utilizadas en el proyecto y como han sido conectadas a los pines de la Basys3, haciendo uso de adaptadores y *protoboards*, donde la memoria paralela no ha sido conectada con todas sus señales por falta de pines en la FPGA. El cable micro USB que podemos ver conectado a la placa es el que usaremos para alimentar la, conectándolo a nuestro ordenador y también el puerto serie por el que nos comunicaremos con la placa.

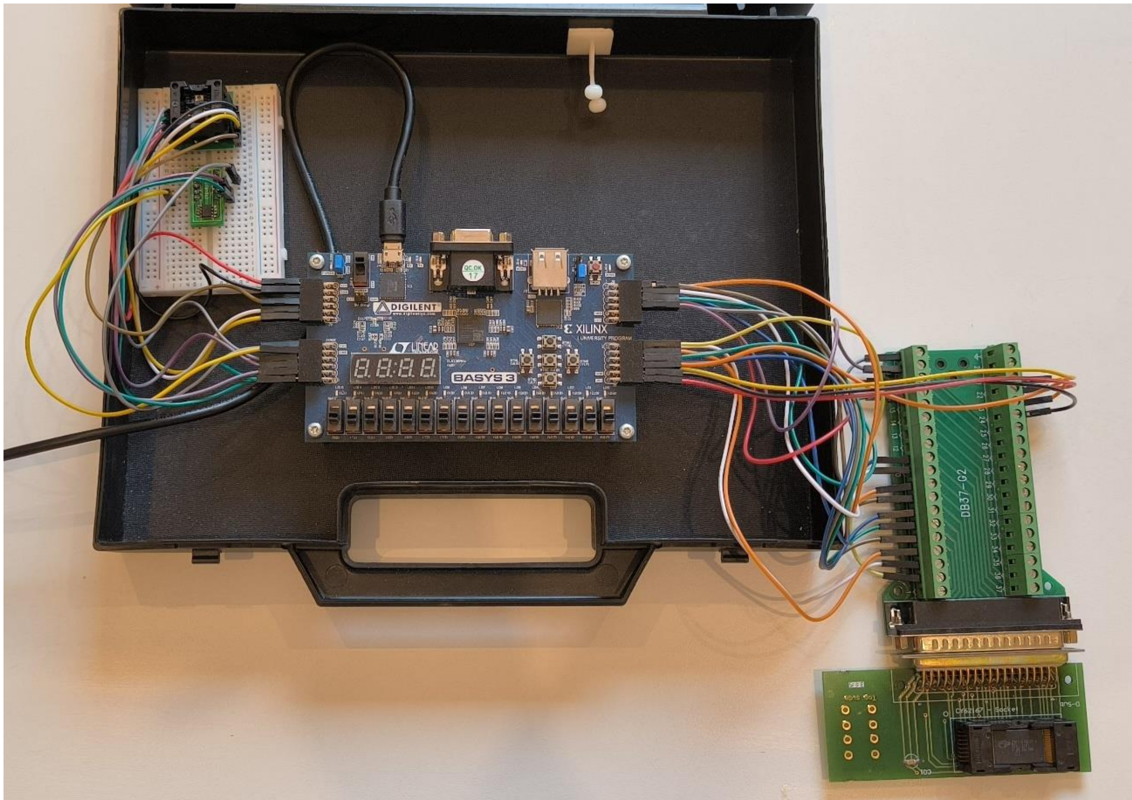

Imagen 14 – Basys 3 con pinout a las memorias

3. Interfaz de usuario

La implementación consta de un menú elaborado en C, el cual se controla a través del puerto serie y usando un *hyperterminal* para visualizarlo en nuestro ordenador, en el cual se permite seleccionar diferentes parámetros descritos a continuación, de la memoria que se desea controlar:

1- **Memoria:**
   - Se selecciona el tipo de protocolo de comunicación que se desea controlar (SPI, QSPI, I2C o paralelas), como podemos ver en la imagen 15.

2- **Lectura/escritura:**
   - Se selecciona si se desea escribir o leer en la memoria seleccionada.

3- **Modo:**
   - Se selecciona entre los modos de escribir/leer en una solo posición o patrón, es decir, escribir en todas las posiciones de la memoria

4- **Addr**:
   - Se escribe la dirección donde se desea escribir/leer. En caso de haber seleccionado el modo patrón, se ignorará el valor de este parámetro.

5- **Valor:**
   - Se escribe el valor que se quiere escribir o bien en la posición seleccionada o en el caso se patrón en todas las posiciones. Se ignora el valor si se selecciona el modo leer.

6- **Frecuencia:**
   - Con este parámetro, como podemos ver en la imagen 16, se puede elegir entre diferentes frecuencias a las que queremos que funcione nuestro sistema, para las memorias SPI, QSPI e I2C. Con el modo de paralelas esta opción no aparecerá, imagen 18. En algunos casos como en el modo SPI, no nos dejará ejecutar el programa con una velocidad superior a 30Mhz y saldrá un mensaje alertando, como podemos ver en la imagen 17.

7- **Dato leído:**
   - En este parámetro aparecerá el dato leído de la memoria.

a) Mode: Spi

b) Function: Read

c) One/Patron Format: Patron

d) Addr 0

e) Write data 0

f) Clk: 30MHz

Data Read: C

Introduce p to Execute the Operation:

a
-a Select the new Mode:

a) Spi

b) Qspi

c) I2C

d) Paralelas

Imagen 15 – Menú elección de protocolo donde seleccionaremos entre los diferentes protocolos

a) Mode: I2C

b) Function: Read

c) One/Patron Format: Patron

d) Addr 0

e) Write data 0

f) Clk: 30MHz

Data Read: C

Introduce p to Execute the Operation:

f
-f

Choose an option

a) 50 MHz

b) 30 MHz

c) 10 MHz

d) 1 MHz

e) 100 KHz

f) 10 KHz

g) 2 KHz

h) 1 KHz

Imagen 16 – Elección de frecuencia a la que queremos realizar la operación de lectura o escritura

Frequency not compatible with Spi Max 30MHz

a) Mode: Spi

b) Function: Write

c) One/Patron Format: Only one

d) Addr 0

e) Write data 0

f) Clk: 50MHz

Data Read: 0

Introduce p to Execute the Operation:

Imagen 17 – Error de frecuencia al intentar escribir a una frecuencia superior a 33MHZ con el protocolo SPI

a) Mode: Paralelas

b) Function: Write

c) One/Patron Format: Only one

d) Addr 0

e) Write data 0

Data Read: C

Introduce p to Execute the Operation:

Imagen 18 – Menú de memoria paralela, sin la opción de elegir la frecuencia

Una vez configurado todo, se introducirá la letra p para empezar la ejecución del controlador seleccionado con los parámetros introducidos.

```
a) Mode: Spi
b) Function: Write
c) One/Patron Format: Only one
d) Addr 0
e) Write data 0
f) Clk: 30MHz
Data Read: 0
Introduce p to Execute the Operation:
```

Imagen 19 - Interfaz de Usuario

En el caso de que hayamos seleccionado escribir en una posición específica, después de haber realizado la escritura volverá a aparecer el menú sin ningún indicativo adicional.

Como podemos ver en la imagen 20, en el caso de haber seleccionado la lectura en una posición determinada, aparecerá nuevamente el menú después de realizar la operación y en el aparatado *Data Read* podremos visualizar el dato que se ha leído.

```
a) Mode: Spi
b) Function: Read
c) One/Patron Format: Only one
d) Addr 3
e) Write data C
f) Clk: 30MHz
Data Read: C
Introduce p to Execute the Operation:
```

Imagen 20 - Lectura Dato SPI

Como podemos ver en la imagen 21, en el caso de haber seleccionado la escritura en modo patrón, una vez haya acabado aparecerá el menú con el mensaje "*Write patron done*" en la parte superior.

```
Write patron done
a) Mode: Spi
b) Function: Write
c) One/Patron Format: Patron
d) Addr 0
e) Write data C
f) Clk: 30MHz
Data Read: C
Introduce p to Execute the Operation:
```

Imagen 21 - Escritura patrón SPI

Como podemos ver en las imágenes 22 y 23, en el caso que leamos de la memoria seleccionando el modo patrón, nuestro sistema habrá guardado el valor que se escribió por última vez usando el modo patrón y nos aparecerá en el terminal el mensaje “leído sin errores”, en caso de que todas las posiciones contengan el dato que se escribió la última vez, o nos aparecerá una lista con las posiciones que han fallado junto al valor que se ha leído de estas. Por último, aparecerá un mensaje indicando la introducción de cualquier tecla para continuar y que aparezca el menú nuevamente.

```
a) Mode: Spi
b) Function: Read
c) One/Patron Format: Patron
d) Addr 0
e) Write data C
f) Clk: 30MHz
Data Read: C
Introduce p to Execute the Operation:
p
-p
||||||||||||||||||||||||||
Save Data: C
Read without errors
||||||||||||||||||||||||||
Introduce any key
```

Imagen 22 - Lectura patrón

```
a) Mode: Spi
b) Function: Read
c) One/Patron Format: Patron
d) Addr 6
e) Write data B3
f) Clk: 30MHz
Data Read: C
Introduce p to Execute the Operation:
p
-p
||||||||||||||||||||||||||
Save Data: C
ERROR Pos:  6, Dato B3
||||||||||||||||||||||||||
Introduce any key
```

Imagen 23 - Lectura patrón con errores

4. Estructura del controlador de memorias implementado en HW.

En la imagen 24 podemos ver la jerarquía de los módulos en el controlador de memoria.

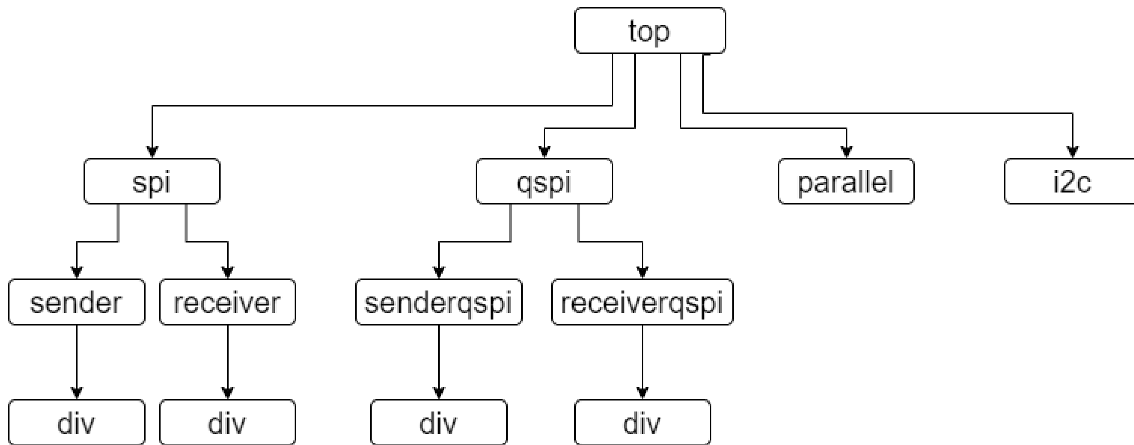

Imagen 24 - Jerarquía módulos VHDL

El top es el módulo principal del que dependen todos. Está compuesto de una máquina de estados para controlar los distintos tipos de memoria, que podemos ver en la imagen 25.

Inicialmente se mantiene en el estado cero “s0”, hasta que el usuario a través de la interfaz de usuario indica el comienzo de la operación, entonces se escribirá un 1 en el registro 0, eso es lo que le indicará al top la señal de inicio para que comience y pase al estado uno “s1”.

En el estado “s1” tenemos un “*if*” con diferentes opciones dependiendo el valor que contenga el registro ocho, indicando qué tipo de protocolo se va a usar para cambiar a un estado o a otro. Si el registro ocho vale uno, pasará al estado dos “s2”, para empezar la ejecución del protocolo SPI. Si el valor del registro ocho es dos pasará al estado tres “s3”, para iniciar el protocolo QSPI. Si el valor del registro ocho es tres, pasará al estado cuatro “s4”, para ejecutar el protocolo I2C y, por último, si el valor del registro ocho es cuatro, pasará al estado seis “s6”, para utilizar el protocolo paralelo.

En cada uno de los estados se marcará el inicio al módulo correspondiente para ejecutar la orden y se asignaran los valores de dato, dirección, velocidad y modo de lectura o escritura. Los estados esperarán la señal de terminación de su modulo correspondiente. Una vez han recibido esta señal de finalización pasarán al estado siete “s7”, donde a través del registro cero indicará a la interfaz de usurario que la operación ha terminado y volverá al estado cero esperando un nuevo comienzo.

A continuación, en la imagen 25 podemos ver el diagrama de los estados utilizados en el top. Comenzando y terminando en S0.

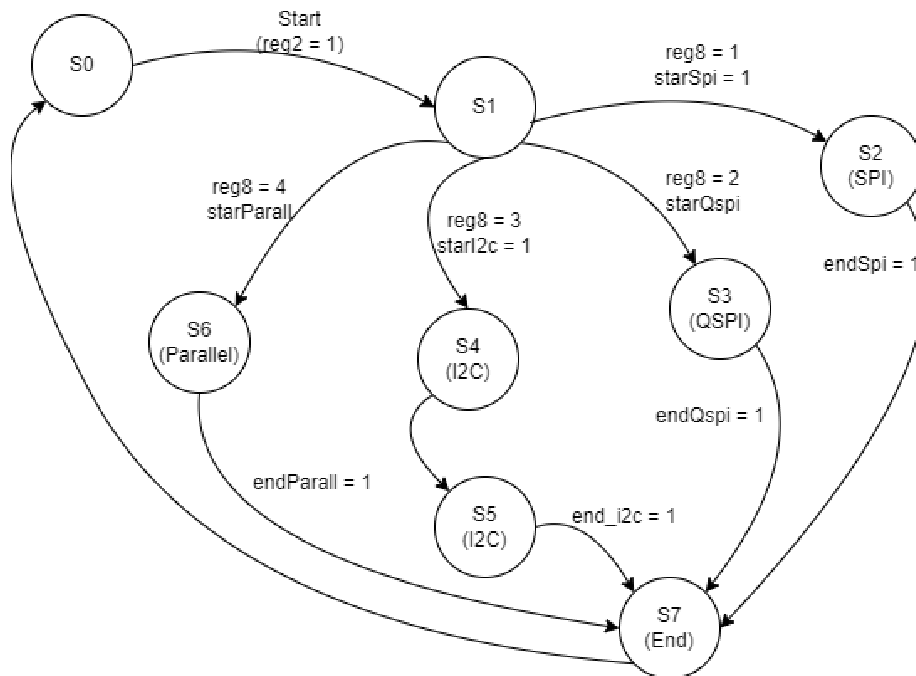

Imagen 25 - Maquina de estados del Top

En las imágenes 26 y 27 podemos ver respectivamente las condiciones para cambiar del estado inicial y el estado final indicando al *sw* el final de la ejecución.

```vhdl
process (clk)
begin

  if rising_edge(clk) then
    if rst = '0' then
      pslv_reg0 <= (others => '0');
      current_state <= s0;
      starI2c <= '0';
      starSpi <= '0';
      starQspi <= '0';
      starParall <= '0';

    else
       case current_state is
           when s0 =>

               pslv_reg0 <= "00000000000000000000000000000000";
               if pslv_reg2 = "00000000000000000000000000000001" then
                   current_state <= s1;
               else
                   current_state <= s0;
               end if;
           when s1 =>
               if pslv_reg8 = "00000000000000000000000000000001" then --Spi
                   current_state <= s2;
                   starSpi <= '1';
               elsif pslv_reg8 = "00000000000000000000000000000010" then --Qspi
                   current_state <= s3;
                   starQspi <= '1';
               elsif pslv_reg8 = "00000000000000000000000000000011" then --I2c
                   current_state <= s4;
                   starI2c <= '1';
                   ena <= '1';
                   addr <= pslv_reg3(7 downto 0);
                   rw <= pslv_reg11(1);
                   data_wr <= pslv_reg4(7 downto 0);

               elsif pslv_reg8 = "00000000000000000000000000000100" then --Parallel
                   current_state <= s6;
                   starParall <= '1';
               end if;
```

Imagen 26 - Código VHDL del estado inicial s0

```vhdl
    when s7 =>
        ena <= '0';
        qestado <= "00100";
        etate <= "00000000000000000000000000010000";
        pslv_reg0 <= "00000000000000000000000000000001";
        if (pslv_reg2 = "00000000000000000000000000000010") then
            current_state <= s0;
        else
            current_state <= s7;
        end if;
    when others =>
            current_state <= s0;
end case;
```

Imagen 27 - Código VHDL del estado final s7

En este módulo también se hace uso de la primitiva IOBUF para transformar las señales I y O a una señal bidireccional, junto a la señal de control T. Esta señal bidireccional la sacamos a los pines de nuestra FPGA para enviar y recibir los datos por la misma línea. Cuando la señal de control T está en alta, la señal bidireccional actúa como entrada y cuando la señal T está en baja actúa como salida. ([2], s.f.)

```
IOBUF_m0 : IOBUF
    generic map (
       DRIVE => 16,
       IOSTANDARD => "LVCMOS33",
       SLEW => "FAST")
    port map (
       O => m0_I,      -- Buffer output
       IO => Mosi0IO,    -- Buffer inout port (connect directly to top-level port)
       I => m0_O,      -- Buffer input
       T => m0_T       -- 3-state enable input, high=input, low=output
    );
```

Imagen 28 - Código VHDL de la primitiva IOBUF

Un módulo muy usado durante la ejecución es el módulo *div*, el cual nos permite generar las diferentes frecuencias de reloj seleccionada desde el software. Este módulo recibe el dato escrito por la interfaz de usuario en el registro cinco y genera una señal de reloj con la frecuencia escogida.

```
entity divlh is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           clk_out: out std_logic;
           slv_regclk       : in std_logic_vector(32-1 downto 0)
           );
end divlh;
```

Imagen 29 - Código VHDL de la entity div

5. Protocolos Utilizados

5.1. SPI

(Serial Peripheral Interface) ([3], s.f.) ([4], s.f.)

Es un protocolo de comunicación serie, síncrono, con una arquitectura maestro esclavo, donde el maestro es el encargado de enviar el *clk* al esclavo para una correcta sincronización. Este protocolo dispone de cuatro líneas para transmitir información entre un microcontrolador y uno o varios periféricos, como puede ser el caso de un sensor de temperatura, donde el sensor actuando como esclavo envíe la temperatura al maestro.

Las líneas de comunicación son las siguientes:

SCLK: Por esta señal el maestro envía el reloj al esclavo con la frecuencia requerida.

MOSI: Señal que sale del maestro (la FPGA) al esclavo (la memoria) con la información, cambiando el dato en el flanco de bajada, como podemos ver en la imagen 30 y tomando el valor en el flanco de subida de la señal *SCLK*.

MISO: Señal que sale de la memoria hacia la FPGA con la información, cambiando el dato al igual que *MOSI* en el flanco de bajada.

CS: Señal que se pone a 0 para indicar que se está transmitiendo por el canal, se mantiene a 0 hasta la finalización de la transmisión.

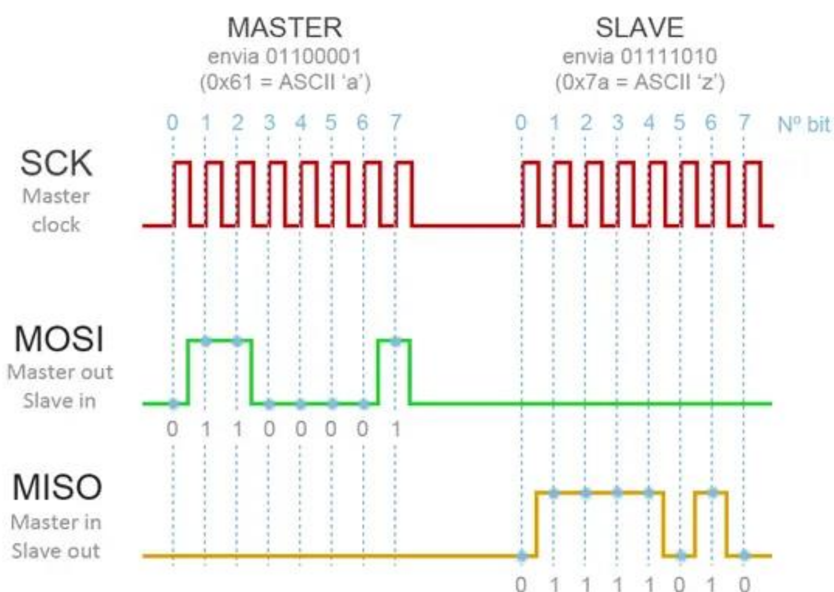

Imagen 30 - Funcionamiento SPI ([5], s.f.)

5.1.1. Comunicación específica para la memoria SPI

Para el protocolo SPI y QSPI se ha utilizado la memoria APS6404L-3SQR QSPI PSRAM, ([11], s.f.).

Los siguientes códigos que se enviarán al principio a la memoria, como podemos ver en la imagen 22, para indicar qué operación se va a realizar:

Código 0x03 para indicar que es una lectura en SPI.

Código 0x02 para indicar que es una escritura en SPI.

Código 0x0B para indicar que es una lectura en SPI a una mayor frecuencia. Este modo no ha sido implementado en el sistema.

Para la lectura de SPI, transmitiremos a través de la señal *MOSI* 8 bits correspondiente al código "0x03" para indicarle a la memoria que queremos leer.

A continuación, enviaremos veinticuatro bits correspondientes a la dirección de memoria que queremos leer y por último esperaremos a recibir ocho bits referentes al dato almacenado en esa posición de memoria a través de la señal *MISO*. En todos los envíos y recepciones, se comienza por enviar el bit más significativo y por último se envía el bit en la posición cero. Este proceso lo podremos hacer con un *SCLK* máximo de 33MHz.

En la imagen 31 podemos ver una trama de lectura extraída del *datasheet* de la memoria.

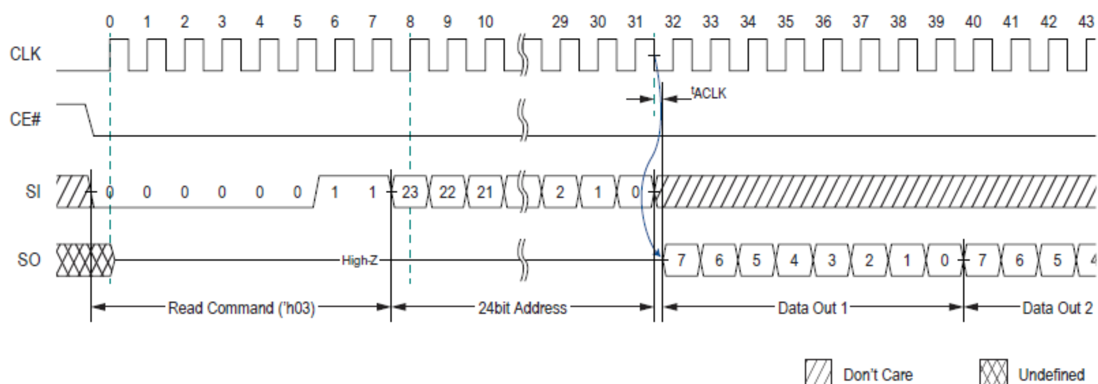

Imagen 31 - memoria SPI Read ([11], s.f.)

Para la escritura de SPI, transmitiremos a través de la señal *MOSI* ocho bits con el código, en este caso "0x02" para indicarle a la memoria que queremos escribir. A continuación, enviaremos veinticuatro bits correspondientes a la dirección de memoria que queremos escribir y por último enviaremos ocho bits con el dato que queremos escribir en esa

posición de memoria. En todos los envíos se comienza por enviar el bit más significativo y por último se envía el bit en la posición cero. Este proceso lo podremos hacer con un *SCLK* como máximo de 84MHz.

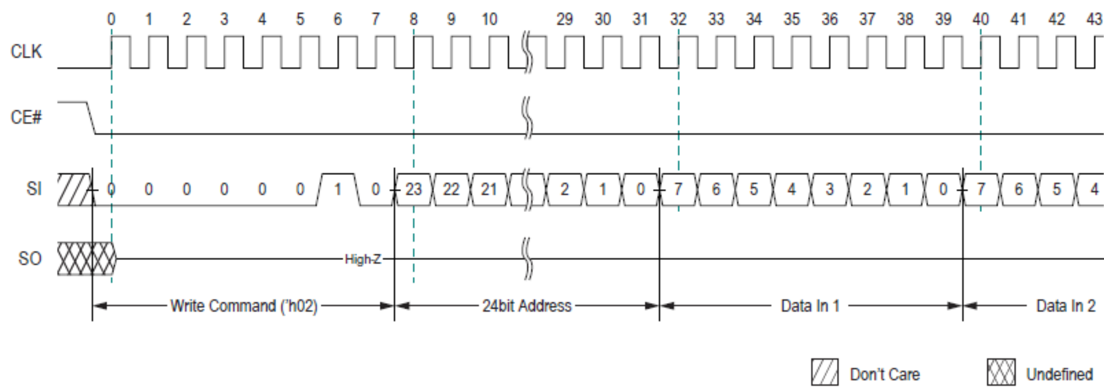

Imagen 32 - memoria SPI Write ([11], s.f.)

5.1.2. Dificultades encontradas:

Difícil sincronización de las señales tomando el valor en flanco de subida y cambiándolo en flanco de bajada, haciendo necesario un *process* que se modificase con flanco de subida del reloj y otro con flanco de bajada.

5.1.3. Modulo VHDL SPI:

Es el módulo encargado del protocolo SPI. Al igual que el top está compuesto por una máquina de estados que inicialmente se encuentra en el estado cero y espera el *start* del módulo Top.

Las señales del módulo SPI son las entradas y salidas correspondientes al protocolo SPI, los registros con todos los parámetros de la operación, señales necesarias como el *start* proveniente del top, para indicar el inicio, el *end* para señalizar el final de la operación el *reset* y el *clk* del sistema.

```
entity spi is
    Port ( clk :  in STD_LOGIC;
           rst :  in STD_LOGIC;
           Start: in std_logic;
           Ends:  out std_logic;

           Sclk : out std_logic;
           Miso:  in std_logic;
           Mosi:  out std_logic;
           Cs:    out std_logic;

           slv_reg1       : out std_logic_vector(32-1 downto 0);
           slv_reg3       : in  std_logic_vector(32-1 downto 0);
           slv_reg4       : in  std_logic_vector(32-1 downto 0);
           slv_reg5       : in  std_logic_vector(32-1 downto 0);
           slv_reg6       : in  std_logic_vector(32-1 downto 0);
           slv_reg7       : in  std_logic_vector(32-1 downto 0);
           slv_reg8       : in  std_logic_vector(32-1 downto 0);
           slv_reg9       : in  std_logic_vector(32-1 downto 0);
           slv_reg10      : in  std_logic_vector(32-1 downto 0);
           slv_reg11      : in  std_logic_vector(32-1 downto 0)
           );
end spi;
```

Imagen 33 - Código VHDL de entity SPI

Hace uso de dos módulos extra, el módulo *sender* y *reciver*.

Las principales señales del *sender* son las siguientes:

Numbits: número de bits de data que se van a enviar.

Data: dato que se va a enviar.

Clkout: señal *SCLK* del protocolo SPI.

Dataout: señal MOSI del protocolo SPI.

Slv_regclk: Dato referente a la frecuencia de SCLK.

El módulo *sender* se usa para enviar los ocho bits de comando, para que distinga entre una operación de escritura o de lectura, para enviar los veinticuatro bits de dirección y por último enviar ocho bits de datos en caso de escritura.

```
entity sender is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;

           numbits : in integer;
           data: in std_logic_vector(31 downto 0);
           start: in std_logic;
           psend : out std_logic;
           clkout: out std_logic;
           dataout: out std_logic;
           stdone: out std_logic;
           slv_regclk: in std_logic_vector(32-1 downto 0)
           );
end sender;
```

Imagen 34 - Código VHDL de la entity sender

Las principales señales del receiver son las siguientes:

Numbits, **Clkout** y **Slv_regclk** tiene la misma función que en el *sender*.

Data: señal donde almacena el dato recibido.

Datain: señal MISO del protocolo SPI.

El módulo receiver se usa para recibir los ocho bits del dato leído en caso de lectura.

```
entity receiver is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;

           numbits : in integer;
           data: out std_logic_vector(31 downto 0);
           start: in std_logic;
           psend : out std_logic;
           clkout: out std_logic;
           datain: in std_logic;
           stdone: out std_logic;
           slv_regclk: in std_logic_vector(32-1 downto 0)
           );
end receiver;
```

Imagen 35 - Código VHDL de la entity receiver

El módulo de SPI multiplexará las salidas dependiendo de en qué estado se encuentre.

5.1.4 Resultados

Las imágenes 36 y 37 están extraída de un logic analyzer conectado a la memoria SPI para mostrar el correcto funcionamiento del sistema empotrado

La imagen 36 es una escritura en SPI, enviando inicialmente el código de escritura 0x02, la dirección 0x41 y el dato a escribir 0x56

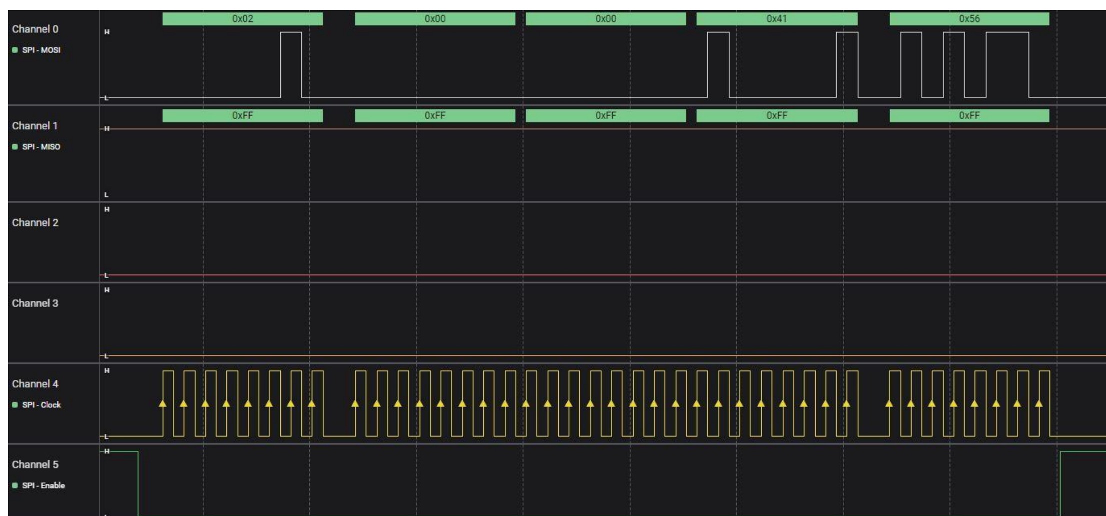

Imagen 36 – Escritura SPI

La imagen 37 es una lectura en SPI, enviando en este caso el código de lectura 0x03, la dirección 0x41, y recibiendo de la memoria el dato 0x56 escrito anteriormente.

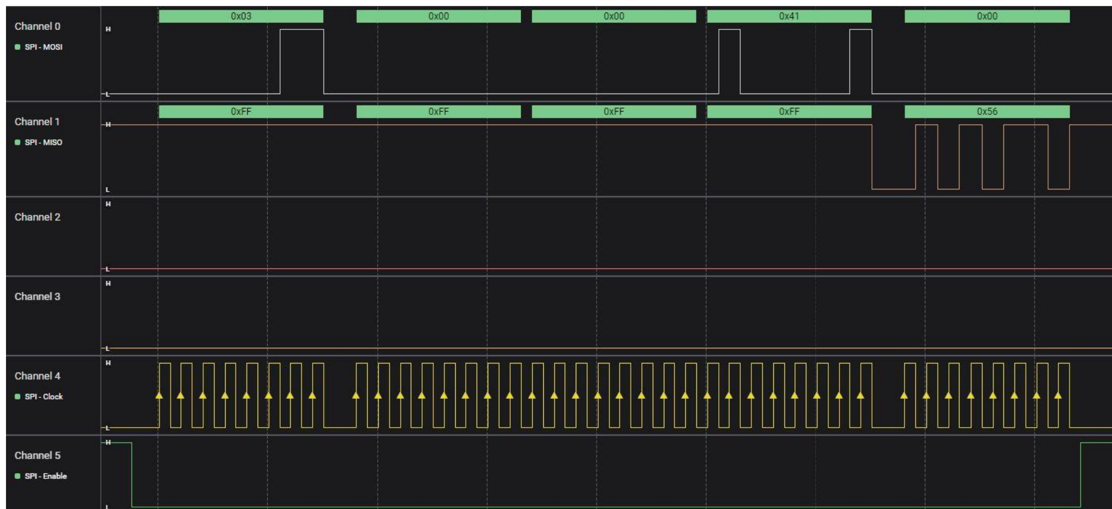

Imagen 37 – Lectura SPI

5.2. QSPI

(Quad Serial Peripheral Interface) ([6], s.f.)

Al igual que SPI es un protocolo de comunicación serie utilizado para la transferencia de datos entre un microcontrolador y uno o varios periféricos. Funciona igual que SPI, con arquitectura maestro esclavo, donde el maestro es el encargado de enviar el reloj al esclavo. Como podemos ver en la imagen 38, ahora tenemos cuatro canales para enviar los datos y los mismos cuatro para recibirlos, lo que nos permite aumentar considerablemente el tiempo de transmisión añadiendo solo tres cables más.

Líneas de comunicación:

SIO0, **SIO1**, **SIO2**, **SIO3**: Señales de entrada / salida de la FPGA conectadas a la memoria con la información, cambiando de dato en el flanco de bajada y cogiendo el valor en el flanco de subida de la señal *SCLK*.

SCLK: Por esta señal el maestro envía el reloj con la frecuencia requerida.

CS: Señal que se pone a 0 para indicar que se está transmitiendo por el canal, se mantiene a 0 hasta la finalización de la transmisión.

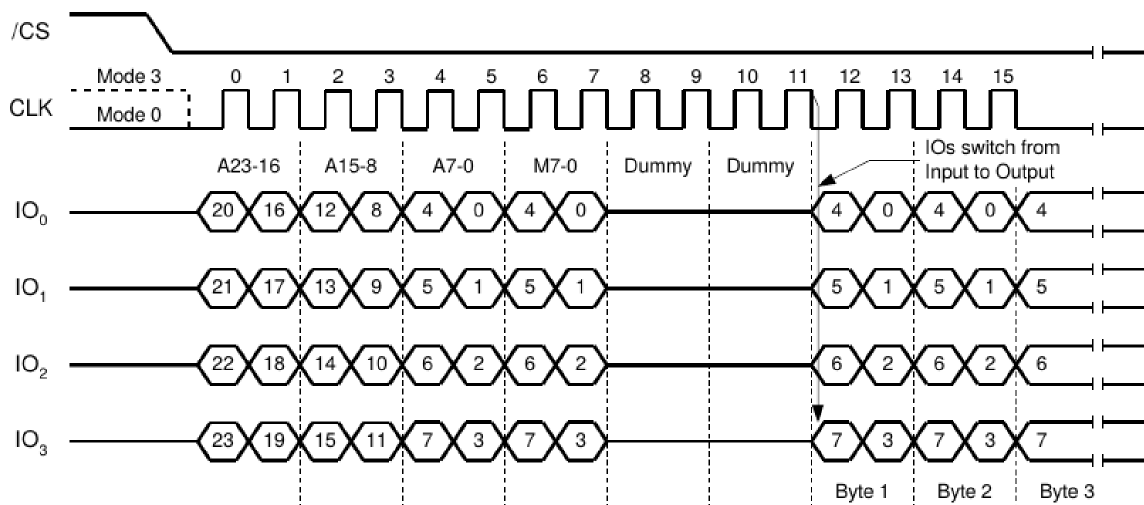

Imagen 38 - Funcionamiento QSPI ([7], s.f.)

5.2.1. Comunicación específica para la memoria QSPI utilizada

Los siguientes códigos, son enviados a la memoria al principio de cada transacción para que esta sepa en qué modo se va a efectuar la operación. Estos códigos solo se enviarán haciendo uso de la señal *SIO0* en modo SPI cómo podemos ver en la imagen 39, la dirección y los datos si se enviaran por los cuatro canales en modo QSPI, para indicar que operación se va a realizar:

Código 0xEB para indicar que es una lectura en QSPI.

Código 0x38 para indicar que es una escritura en QSPI.

Para la lectura de QSPI, transmitiremos a través de la señal *SIO0* ocho bits referentes al código correspondiente, a una frecuencia máxima de 30Mhz, en este caso enviaremos “0xEB” para indicarle a la memoria que queremos leer en modo QSPI. A continuación, enviaremos veinticuatro bits correspondientes a la dirección de memoria que queremos leer, utilizando las cuatro señales *SIO* del QSPI, empezando con el bit más significativo enviado por *SIO3* el segundo menos significativo por *SIO2*, y los siguientes por *SIO1* y *SIO0* respectivamente, consiguiendo enviar cuatro bits por cada ciclo de reloj. Seguidamente esperaremos seis ciclos de reloj tras los que recibiremos los ocho bits del dato almacenado en la posición de memoria indicada a través de los bits *SIO*, recibiendo el más significativo por *SIO3* de forma similar al envío de la dirección.

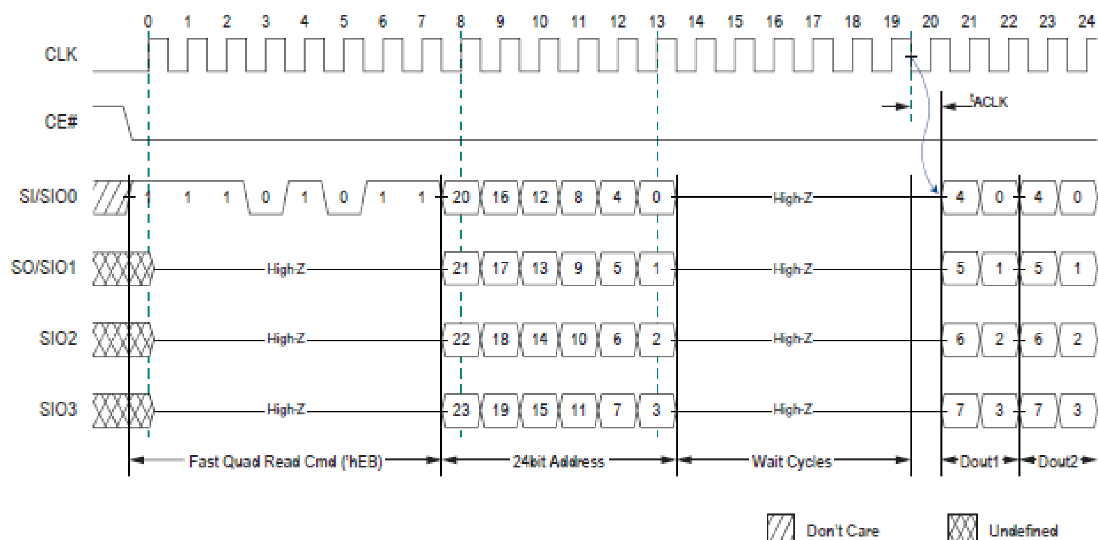

Imagen 39 - memoria QSPI Read ([11], s.f.)

Para la escritura de QSPI, transmitiremos a través de la señal *SIO0* ocho bits referentes al código a una frecuencia máxima de 50Mhz, en este caso enviaremos “0x38”, para indicarle a la memoria que queremos escribir en modo QSPI. A continuación, enviaremos veinticuatro bits correspondientes a la dirección de memoria que queremos escribir, utilizando las cuatro señales *SIO* de la misma manera que hacíamos en la escritura. A continuación, enviaremos el dato que queremos escribir en la posición de memoria por las señales *SIO*.

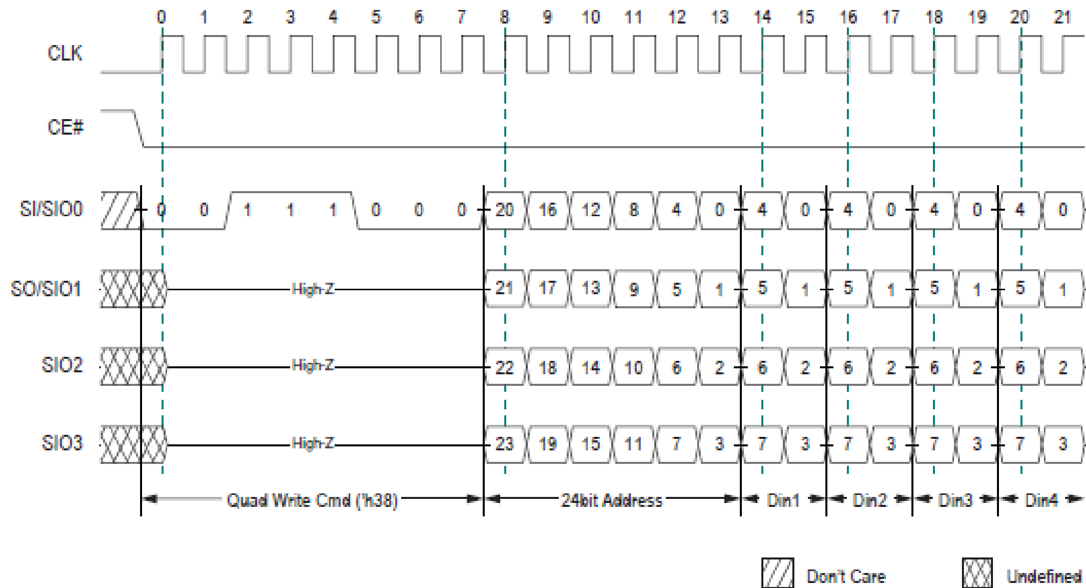

Imagen 40 - memoria QSPI Write ([11], s.f.)

5.2.2. Errores encontrados:

Al leer de la memoria en formato QSPI, cuando los valores que devuelve tienen más de un bit a 1 se produce inconsistencia en los datos debido a que la alimentación que proporciona la FPGA no es suficiente.

La solución es aportarle una alimentación extra con una fuente externa.

5.2.3. Modulo VHDL QSPI

Al ser muy similar a SPI, tiene los mismos módulos ligeramente modificados para que ahora en cada flanco de reloj, envíe o reciba cuatro bits, haciendo uso de los módulos *sender*, para envía el código de operación, *senderqpi* y *receiverqpi*, al igual que los módulos de SPI hacen uso del módulo div para funcionar a la frecuencia escogida por la interfaz de usuario.

También hace uso de *control_T*, señal de control para indicar el modo entrada salida de las señales *MISO* y *MOSI* dependiendo en qué estado se encuentre.

```vhdl
entity qspi is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           Start: in std_logic;
           Ends: out std_logic;

           Sclk : out std_logic;
           Miso0: in std_logic;
           Miso1: in std_logic;
           Miso2: in std_logic;
           Miso3: in std_logic;
           Mosi0: out std_logic;
           Mosi1: out std_logic;
           Mosi2: out std_logic;
           Mosi3: out std_logic;
           control_T: out std_logic;
           Cs:        out std_logic;

           slv_reg1       : out std_logic_vector(32-1 downto 0);
           slv_reg3       : in  std_logic_vector(32-1 downto 0);
           slv_reg4       : in  std_logic_vector(32-1 downto 0);
           slv_reg5       : in  std_logic_vector(32-1 downto 0);
           slv_reg6       : in  std_logic_vector(32-1 downto 0);
           slv_reg7       : in  std_logic_vector(32-1 downto 0);
           slv_reg8       : in  std_logic_vector(32-1 downto 0);
           slv_reg9       : in  std_logic_vector(32-1 downto 0);
           slv_reg10      : in  std_logic_vector(32-1 downto 0);
           slv_reg11      : in  std_logic_vector(32-1 downto 0)
           );
end qspi;
```

Imagen 41 - Código VHDL de la entity QSPI

5.2.4 Resultados

Las imágenes 42 y 43 están extraída de un analizador logico conectado a la memoria QSPI para mostrar el correcto funcionamiento del sistema empotrado.

La imagen 42 es una escritura en QSPI, enviando inicialmente el código de escritura 0x38 solo por la señal *SIO0*, la dirección 0xD5 y el dato a escribir 0x08.

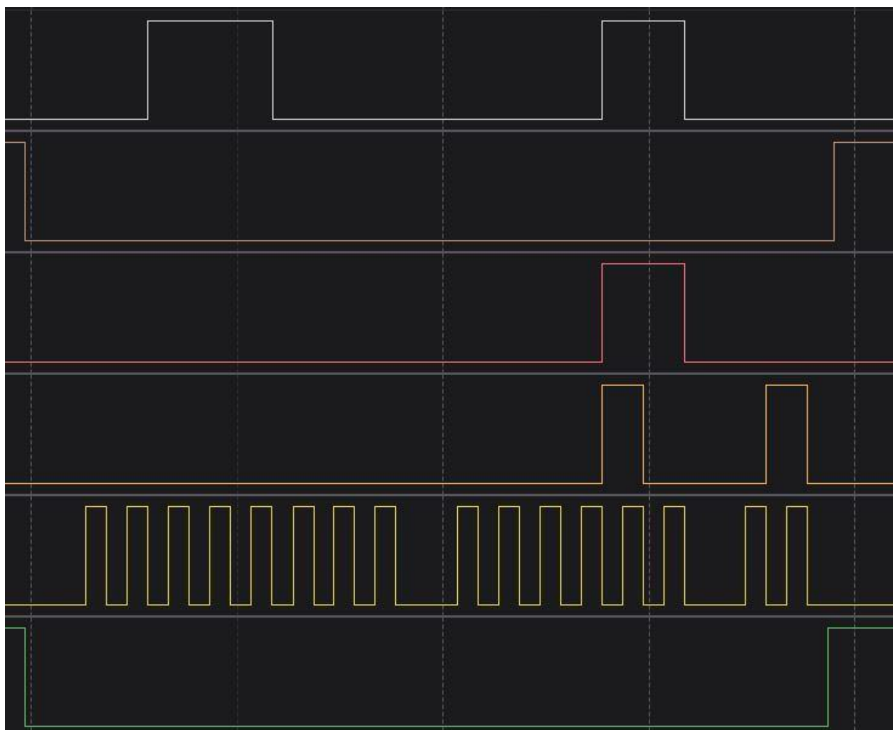

Imagen 42 – Escritura QSPI

La imagen 43 es una lectura en QSPI, enviando en este caso el código de lectura 0xEB, la dirección 0xD5, y recibiendo de la memoria el dato 0x08 escrito anteriormente, con los seis ciclos de espera antes de recibirlo.

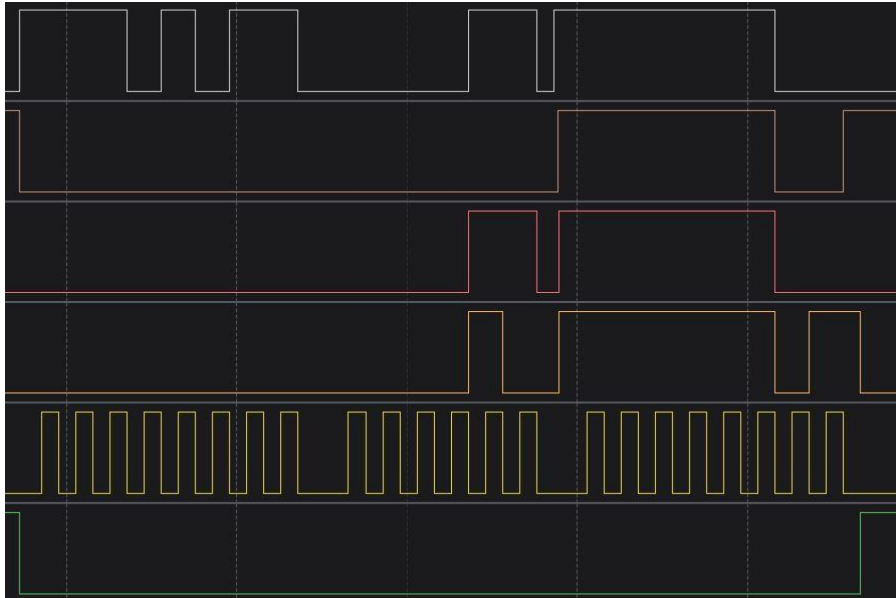

Imagen 43 – Lectura QSPI

5.3. Protocolo paralelo

Se trata de un protocolo de comunicación asíncrono, donde hay varias líneas de datos y direcciones, donde cada una transmitirá un dato en binario tan grande como señales haya implementadas. También tendremos dos líneas de control para manejar la escritura y lectura de los datos en la memoria.

Líneas de comunicación:

Address: Bus de varias señales de salida de la FPGA conectadas a la memoria, para indicar la dirección de memoria de la que leer o escribir el dato proporcionado por *Data*.

Data: Bus de varias señales de entrada / salida de la FPGA conectadas a la memoria que transmite los datos.

We: Señal que se activa para marcar la escritura de la dirección seleccionada en la memoria. Como podemos ver en la imagen 44, se trata de una escritura, al tener la señal *We* en alta, del dato "0x1" en la posición "0x6".

Oe: Señal que se activa para marcar la lectura de la dirección seleccionada en la memoria.

Dependiendo de las señales que le lleguen, la memoria leerá o escribirá en la posición de memoria definida en *Address* el dato de la señal *data*.

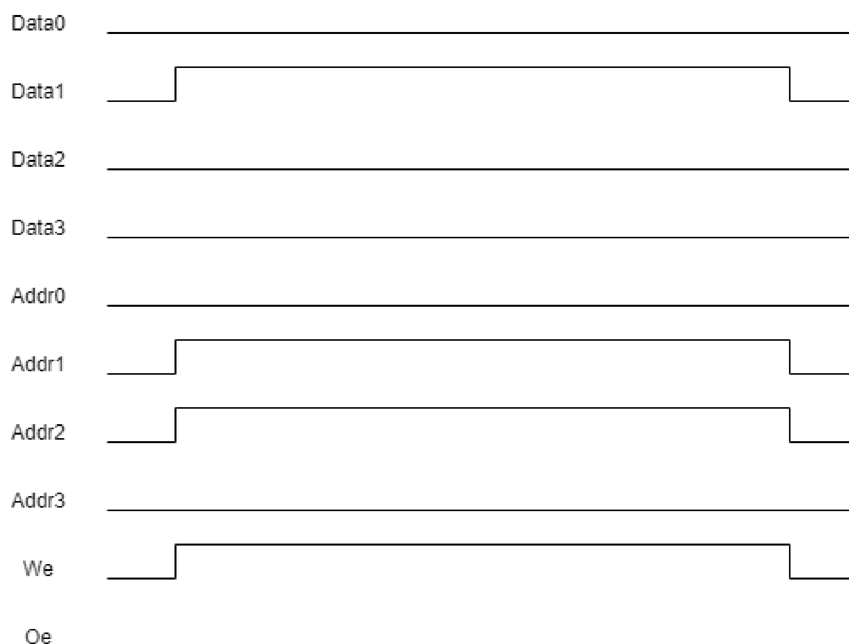

Imagen 44 - Funcionamiento Paralelas

5.3.1. Comunicación específica para la memoria Paralela utilizada

Para el protocolo paralelo se ha utilizado la memoria CY62167DV30, ([12], s.f.).

Para el uso de la memoria paralela no se han podido utilizar todos los pines de direcciones y datos por falta de conexiones, se han implementado veinte pines para la dirección de lectura / escritura, y cuatro para los datos.

Para la lectura de la memoria paralela, se transmiten por los pines de *addr* la dirección de memoria que se desea leer, la señal *WE*_n se pone a 1 y la *OE_n* a 0. La memoria devolverá el dato leído de la posición indicada por los pines de data.

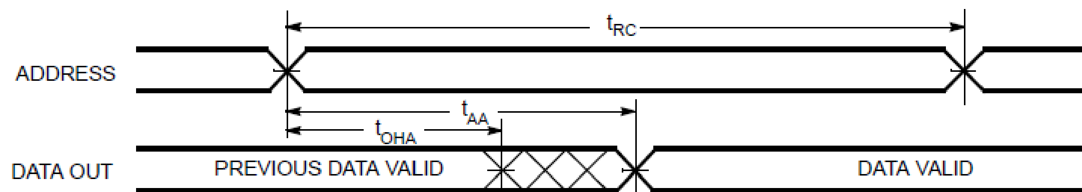

Imagen 45 - memoria Paralelas ([12], s.f.)

Para la escritura de la memoria paralela, se transmiten por los pines de *addr* la dirección de memoria que se desea escribir, la señal *WE_n* se pone a 0 y la *OE_n* a 1 y por los pines de datos el dato que se desea escribir.

5.3.2. Errores encontrados

Ha sido necesario aumentar el tiempo que las señales mantienen el valor seleccionado, hasta poder conseguir una buena lectura y escritura de los datos, ya que la memoria soporta una frecuencia máxima de 18 MHZ.

5.3.3. Módulo VHDL protocolo paralelo

El módulo del protocolo de las memorias paralelas tiene las mismas señales del control que los demás módulos, el *clk*, *reset*, *start* y *end*. También tiene las señales propias del control de la memoria, CE1_n, CE2, OE_n, BHE_n, BLE_n, que será necesario utilizar por estar punteadas ya en nuestro adaptador. Las señales que si será necesario controlar son, WE_n poniéndolo a “0” para realizar la operación de escritura al estar invertida, manteniéndola en “1” en cualquier otro momento y OE_n poniéndolo a “0” para realizar la operación de lectura, también invertida, manteniéndola también a “1” en otros casos.

Otras señales necesarias son:

a) Las señales de datos, de las cuales solo usaremos cuatro de las ocho de las que dispone la memoria por falta de pines de la FPGA, *Data_out* y *Data_in* ambas de un ancho de cuatro bits. Estas señales serán bidireccionales y hará uso de *Control_T* para marcar el uso de una u otra.
b) Las señales de dirección, de las cuales solo usaremos diez de las veintiuna disponibles en la memoria por falta de pines en la FPGA.
c) Al igual que el resto de los módulos, el protocolo paralelo también recibirá los registros con la información de toda la operación.

```
entity parallel is
    Port ( clk : in STD_LOGIC;
            rst : in STD_LOGIC;
            Start: in std_logic;
            Ends: out std_logic;

            CE1_n : out std_logic;
            CE2: out std_logic;
            WE_n: out std_logic;
            OE_n: out std_logic;
            BHE_n: out std_logic;
            BLE_n: out std_logic;
            Data_out : out std_logic_vector(3 downto 0);
            Data_in : in std_logic_vector(3 downto 0);
            Addr : out std_logic_vector(19 downto 0);
            Control_T: out std_logic;

            slv_reg1      : out std_logic_vector(32-1 downto 0);
            slv_reg3      : in std_logic_vector(32-1 downto 0);
            slv_reg4      : in std_logic_vector(32-1 downto 0);
            slv_reg5      : in std_logic_vector(32-1 downto 0);
            slv_reg6      : in std_logic_vector(32-1 downto 0);
            slv_reg7      : in std_logic_vector(32-1 downto 0);
            slv_reg8      : in std_logic_vector(32-1 downto 0);
            slv_reg9      : in std_logic_vector(32-1 downto 0);
            slv_reg10     : in std_logic_vector(32-1 downto 0);
            slv_reg11     : in std_logic_vector(32-1 downto 0)
        );
end parallel;
```

Imagen 46 - Código VHDL de la entity parallel

5.3.4 Resultados

Las imágenes 47 y 48 están extraída de un logic analyzer conectado a la memoria paralela para mostrar el correcto funcionamiento del sistema empotrado.

La imagen 47 es una escritura con el protocolo paralelo, solo observando cuatro señales de dirección, dos de dato y el *OE_N* y *WE_N*. En este caso hemos seleccionado la dirección 0x02 y el dato 0x06.

La imagen 48 es la lectura con el protocolo paralelo descrito anteriormente.

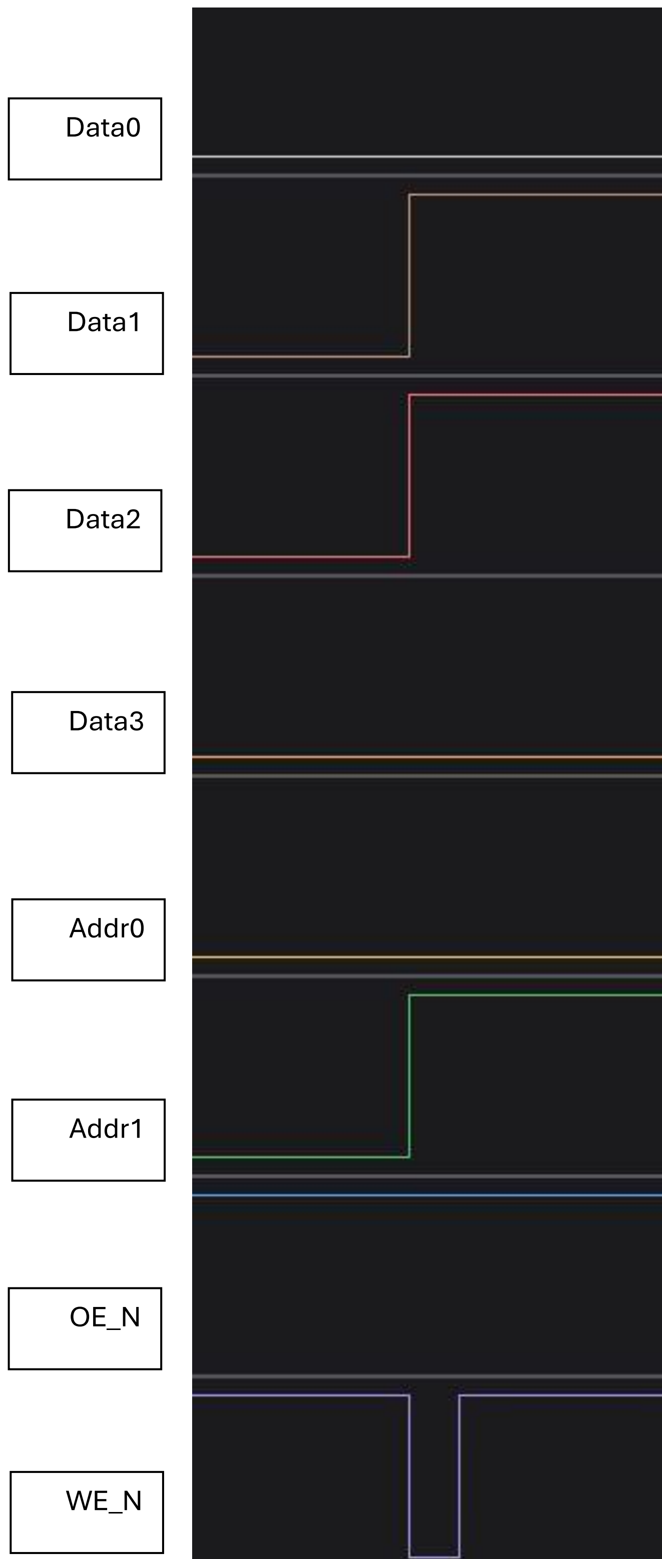

Imagen 47 -Escritura Paralela

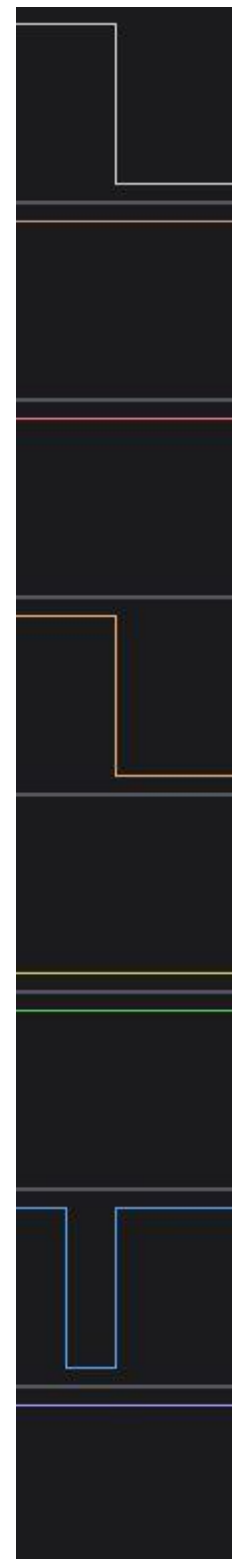

Imagen 48 – Lectura Paralela

5.4. I2C

(*Inter-Integrated Circuit)* ([8], s.f.) ([9], s.f.)

Es un protocolo de comunicación serie, síncrono, con una arquitectura maestro esclavo, donde el maestro es el encargado de enviar el *clk* al esclavo para una correcta sincronización. El esclavo puede mantener el *clk* a "0" si no está preparado para responder. Este protocolo dispone de dos líneas para transmitir información entre un microcontrolador y uno o varios periféricos.

Es un protocolo de comunicación serie utilizado para la transferencia de datos entre un microcontrolador y uno o varios periféricos.

Líneas de comunicación:

SDA: Señales de entrada / salida de la FPGA conectadas a la memoria que transmite los datos.

SCL: Señal que envía el reloj para sincronizar los datos con el esclavo, en este caso la memoria.

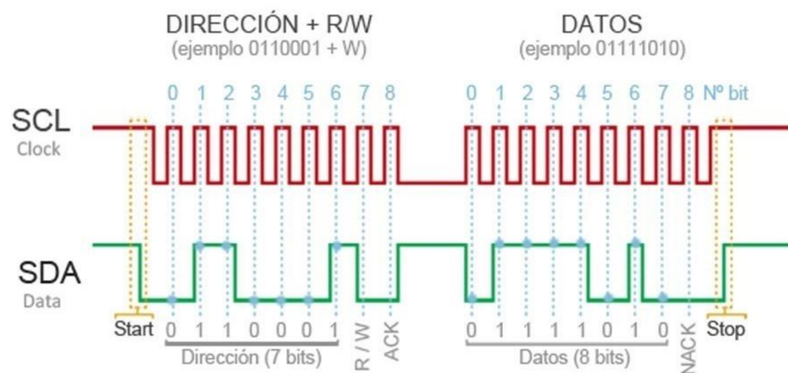

Imagen 49 - Funcionamiento I2C ([10], s.f.)

5.4.1 Comunicación específica para la memoria I2C utilizada

Para el protocolo I2C se ha utilizado la memoria CY14B101J, ([13], s.f.).

Se hace uso de la línea *SDA*, para enviar la dirección y el dato a escribir y para recibir los datos leídos, y de la línea *SCL* para enviar el reloj y sincronizar el envío y recepción de datos con la memoria.

Para realizar una escritura primero se marca un *start*, esto se realiza teniendo la señal *SCL* constante a nivel alto y realizando un flanco de bajada en *SDA*, como podemos observar al principio de la imagen 49. A continuación se envía por la señal *SDA* la dirección del esclavo

(*Memory Slave Address*), como podemos ver en la imagen 50, sería la dirección *"1010A2A1A16"* siendo el valor de A1, A2 (bits para seleccionar el dispositivo específico) y A16 (para seleccionar el dispositivo esclavo) cero, para esta memoria en específico y posteriormente se recibe el *Ack*.

A continuación, se envían los bits más significativos de la dirección de memoria con la recepción del *Ack*, seguido de los bits menos significativos con el *Ack*. Por último, se envía el Dato a escribir, seguido de la recepción del *Ack* junto a la señal de *stop*, teniendo el *SCL* constante a nivel alto y un flanco de subida en *SDA*.

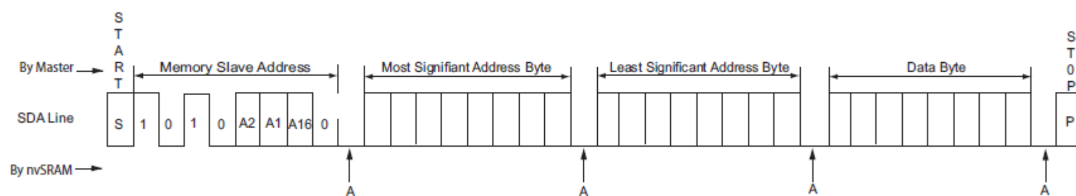

Imagen 50 Memoria I2C Write ([13], s.f.)

Para la lectura, se envía al igual que en escritura la señal de *start*, el Memory Slave Address y la dirección de memoria con sus respectivos *Acks*. A continuación, se marca otra señal de *start*, seguido del *Memory Slave Address*, pero esta vez con el último bit a uno, marcando una lectura. Por último, se reciben los bits del dato leído, se envía un *Ack* de recepción y se marca la señal de stop.

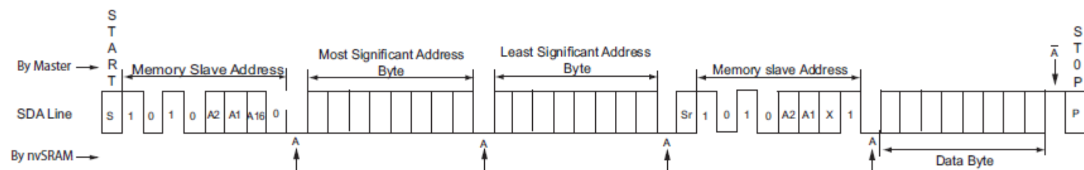

Imagen 51 Memoria I2C Read ([13], s.f.)

5.4.2. Errores encontrados

La FPGA no puede proporcionar suficiente energía y ha sido necesario usar una fuente de alimentación externa.

5.4.3. Modulo VHDL I2C

El módulo I2C también tiene las señales de control del reloj, el *reset*, *ena*, para indicar el comienzo de la operación y *busy* que se mantendrá a uno hasta finalizar la operación.

Al igual que el resto de los módulos tenemos las señales propias de I2C, *SDA* con su control *sda_t* para manejar la entrada salida y la señal de reloj *SCL*.

Este módulo se encarga de señalar el *start* en el protocolo, enviar la dirección de 6 bits a la memoria I2C junto al bit *wr*, seguidamente recibirá su *ack* y a continuación enviará o recibirá los datos de la memoria con su *ack* correspondiente y por último marcará el *end* para señalar la finalización.

```vhdl
ENTITY i2c_master IS
  GENERIC(
    input_clk : INTEGER := 50000000;
    bus_clk   : INTEGER := 100000);
  PORT(
    clk       : IN     STD_LOGIC;
    reset     : IN     STD_LOGIC;
    ena       : IN     STD_LOGIC;
    addr      : IN     STD_LOGIC_VECTOR(6 DOWNTO 0);
    rw        : IN     STD_LOGIC;
    data_wr   : IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
    busy      : OUT    STD_LOGIC;
    data_rd   : OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
    ack_error : BUFFER STD_LOGIC;
    sda_I       : IN   STD_LOGIC;
    sda_O       : OUT  STD_LOGIC;
    sda_T       : OUT  STD_LOGIC;
    scl_I       : IN   STD_LOGIC;
    scl_O       : OUT  STD_LOGIC;
    scl_T       : OUT  STD_LOGIC;

    mi2cestado: out std_logic_vector(4 downto 0)
    );
END i2c_master;
```

Imagen 52 - Código VHDL de la entity I2C

5.4.4 Resultados

Las imágenes 53 y 54 están extraída del analizador lógico conectado a la memoria I2C para mostrar el correcto funcionamiento del sistema empotrado.

La imagen 53 es una escritura en I2C, enviando inicialmente el *Memory Slave Address* indicado anteriormente, seguido de la dirección en este caso 0x04, y por último el dato 0xC3, todo con su ACK correspondiente.

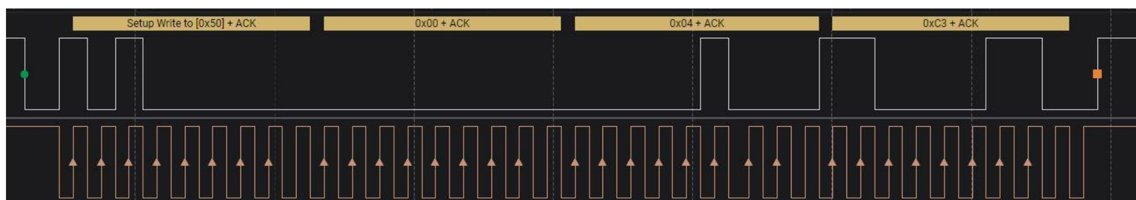

Imagen 53 -Escritura I2C

La imagen 54 es la lectura en I2C de los datos escritos anteriormente, donde se recibe el dato 0xC3 de la dirección 0x04.

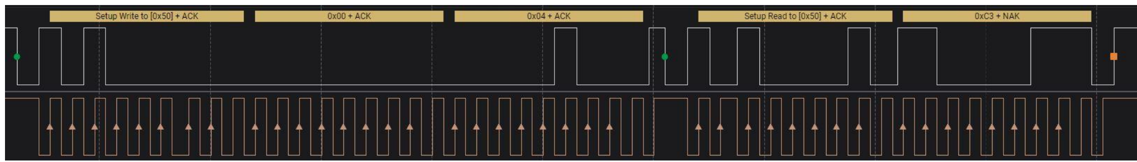

Imagen 54 – Lectura I2C

6. Conclusiones y trabajo futuro

Como trabajo futuro se podría implementar en *hw* bucles *for* para leer y escribir en varias posiciones de la memoria en el VHDL y hacer una comparativa con el C y ver la mejora de rendimiento. También se podría añadir la opción por la cual mediante los registros seis y siete actualmente no usados, se implantase la función de introducir el tamaño de la memoria en específico que se va a usar y el sistema calculase todas las direcciones y tamaños de palabra.

Posibles mejoras de este controlador son añadir nuevos protocolos y probar otras memorias con el fin de aumentar las capacidades del proyecto. También se puede hacer una comparativa de tiempo entre los distintos protocolos implementados en el proyecto, para comprobar que protocolo es más conveniente en cada caso.

7.Bibliografía

[1]. (s.f.). Obtenido de Xilinx: https://china.xilinx.com/support/documents/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[2]. (s.f.). Obtenido de Docs AMD: https://docs.amd.com/r/en-US/ug953-vivado-7series-libraries/IOBUF

[3]. (s.f.). Obtenido de MCI educación: https://cursos.mcielectronics.cl/2022/08/23/serial-peripheral-interface-spi/#:~:text=SPI%20funciona%20de%20una%20manera,ambos%20lados%20en%20perfecta%20sincron%C3%ADa

[4]. (s.f.). Obtenido de Vida Embebida: https://vidaembebida.wordpress.com/2017/02/08/protocolo-de-comunicacion-spi/

[5]. (s.f.). Obtenido de Luis Llamas: https://www.luisllamas.es/arduino-spi/

[6]. (s.f.). Obtenido de Prodigy Technovations: https://www.prodigytechno.com/qspi-protocol#:~:text=QSPI%20Protocol%3A%20Introduction,chip%20memory%20is%20not%20enough

[7]. (s.f.). Obtenido de Gisselquist Technology: https://zipcpu.com/blog/2019/03/27/qflexpress.html

[8]. (s.f.). Obtenido de Hetpro: https://hetpro-store.com/TUTORIALES/i2c/

[9]. (s.f.). Obtenido de Wikipedia: https://es.wikipedia.org/wiki/I%C2%B2C

[10]. (s.f.). Obtenido de Aprendiendo Arduino: https://aprendiendoarduino.wordpress.com/2017/07/09/i2c/

[11]. (s.f.). Obtenido de Mouser Electronics: https://eu.mouser.com/datasheet/2/1127/APM_PSRAM_QSPI_APS6404L_3SQR_v2_3_PKG-1954826.pdf

[12]. (s.f.). Obtenido de Mouser Electronics: https://www.mouser.com/datasheet/2/100/CY62167DV30-29316.pdf

[13]. (s.f.). Obtenido de Infineon: https://www.infineon.com/dgdl/Infineon-CY14C101J_CY14B101J_CY14E101J_1_MBIT_(128K_X_8)_SERIAL_(I2C)_NVSRAM-DataSheet-v17_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0ebfc9013449