Universidad de Sevilla.
Escuela Politécnica Superior de Sevilla.

Trabajo Fin de Grado en

Ingeniería Electrónica Industrial.

Desarrollo de un dispositivo esclavo de un bus VME sobre FPGA.

Autor: Juan José Cid López

Tutores: Carlos Jesús Jiménez Fernández, Francisco Asís Carmona Luque

Departamento de Tecnología Electrónica

Fecha: 05-2023

Universidad de Sevilla.
Escuela Politécnica Superior de Sevilla.

Universidad de Sevilla.
Escuela Politécnica Superior de Sevilla.

RESUMEN

En este trabajo de fin de grado se presenta el diseño hardware de un esclavo para el protocolo de comunicación VME. Este diseño se implementa en una plataforma System on Chip (SoC) Zynq de Xilinx, de forma que pueda comunicarse de forma estándar con otros dispositivos. El diseño del esclavo VME ha sido realizado en VHDL e implementado FPGA, mientras que la parte software consta de un programa en C que corre sobre un procesador ARM.

El bus VME admite múltiples formas de funcionamiento, pero el esclavo desarrollado sólo implementa las más significativas. Para asegurar el correcto funcionamiento, todos los modos de funcionamiento del esclavo que se han implementado, han sido verificados mediante simulaciones en las que al esclavo se le introducen mensajes para los distintos modos de operación. Aunque en la memoria sólo se presentan las más significativas, en un Anexo se muestran las imágenes del resto de verificaciones.

Una vez comprobado el correcto funcionamiento del esclavo mediante simulaciones, se pasó a verificar el funcionamiento haciendo uso de un diseño que implementa el comportamiento de un maestro VME. Comunicando el esclavo con el maestro se volvió a comprobar el correcto funcionamiento del esclavo. Al igual que en la simulación del esclavo aislado, en estas simulaciones se probaron todos los modos de funcionamiento implementados.

El siguiente paso es probar experimentalmente el funcionamiento del esclavo VME diseñado. Para ello se ha implementado el diseño del esclavo en una FPGA y se ha conectado a un bus VME. Para hacer esta conexión ha sido necesario diseñar un circuito de adaptación de tensión, empleando un chip con esa función, puesto que los niveles de tensión del bus no son compatibles con las tensiones de entrada/salida de la FPGA.

Todo este proceso seguido nos ha garantizado el correcto funcionamiento del diseño realizado. Con la realización de este trabajo se ha profundizado en múltiples aspectos vistos en los estudios de grado: diseño de sistemas hardware, verificación de los diseños, programación de microprocesadores, diseño de placas PCB y realización de pruebas experimentales con circuitos electrónicos.

Universidad de Sevilla.
Escuela Politécnica Superior de Sevilla.

ABSTRACT

In this final degree project is going to present the creation of a slave of the VME communication protocol, designed in VHDL for the FPGA part, and a small program in C for the software part.

An SoC (System on Chip) has been used for its realisation, which integrates both parts.

Once I have explained how the design has been carried out, I'm going to go on to simulate this bus carrying out its communications correctly. In this way, some of its transmissions are going to be explained, as well as all the others will be shown in an annexe with images.

Once it has been verified that everything is working correctly, we will move on to the demonstration part, where we will check, with the use of a VME master, which in this case will be another prototype, that the slave is working correctly. As in the verification, as many transmissions as possible will be reflected, in order to demonstrate that everything designed works in reality.

Before being able to test this slave with the master, it has been necessary to design a voltage adaptation circuit, using a chip with this function, as it will be necessary to be able to use this slave in reality.

Once all this has been done, different conclusions are going to be drawn, in order to be able to determine what we wanted to obtain, and if everything has worked as planned.

Universidad de Sevilla.
Escuela Politécnica Superior de Sevilla.

Universidad de Sevilla.
Escuela Politécnica Superior de Sevilla.

Índice

Índice de Figuras

- **Figura I.11: Que contiene la memoria para la lectura de dos - cero bytes (desalineada)**
- **Figura I.12: Simulación de lectura en el modo dos - cero byte (desalineado)**
- **Figura I.13: Simulación de escritura en el modo cuatro bytes, en una transmisión por bloque.**

Índice de Tablas

1. INTRODUCCION

En este Trabajo de Fin de Grado se va a realizar el desarrollo de un dispositivo esclavo de un bus VME sobre FPGA como parte de un proyecto dentro de la empresa AERTEC Solutions, integrado una tarjeta para la adquisición de datos. El esclavo del bus VME tendrá la función de comunicar los datos que se obtengan de diferentes sensores con el maestro, y una vez que se tengan, se pueda realizar con ellos cualquier operación que se desee (automatizar pruebas, control, ajuste del diseño, etc.). Para poder realizar este proyecto, ha sido necesario tener un maestro de dicho bus, ya que, si no, no se hubiese podido probar el correcto funcionamiento del esclavo desarrollado. Este maestro ha sido realizado también en un proyecto independiente dentro de la empresa, por un compañero tanto del trabajo, como de la carrera. Cabe destacar que el diseño del esclavo ha sido realizado de manera individual en el marco de este Trabajo Fin de Grado.

El objetivo de este trabajo es el diseño de un esclavo VME con un funcionamiento limitado a los parámetros que han sido marcados por la empresa. Los esclavos VME presentan multitud de funcionalidades [1], de las cuales solo se han realizado algunas de ellas, aquellas requeridas por la empresa. Además, se tiene un segundo objetivo, consistente en la creación de una interfaz de memoria que sea accesible desde un microprocesador, para así poder trasladar los datos de los diferentes sensores recibidos a través del bus VME a un sistema basado en microprocesador para que pueda ser procesado de la forma en que se desee.

Con la realización de este proyecto, se ha ahondado en el diseño de sistemas digitales basados en VHDL e implementados sobre FPGA, así como en la comunicación entre la parte FPGA y un microprocesador dentro de un System on Chip.

Por ello, se ha trabajado sobre los entornos de Xilinx, desarrollando sobre Vivado la parte VHDL y sobre Vitis la parte C. Se ha investigado y aprendido sobre aspectos de diseño digital sobre VHDL necesarios para el diseño del esclavo VME, sobre mecanismos de comunicación entre el diseño desarrollado en VHDL y el microprocesador mediante registros y buses estándar y sobre el acceso a los datos de dichos registros usando un programa C corriendo en el microprocesador.

Dentro del mecanismo de comunicación entra la parte FPGA y el microprocesador, se ha investigado y aprendido sobre el bus AXI, para la comunicación entre ambas partes, sobre el manejo de los diferentes relojes del sistema (aunque finalmente simplemente fue necesario emplear el del microcontrolador) y, sobre todo, el uso del bloque de memoria, proporcionado por Xilinx [2], utilizando para su comunicación tanto el bus AXI [3], como trabajando con señales internas, comunicándolas con el propio diseño VHDL.

Además de esto, también se ha trabajado sobre otros puntos importantes, los cuales han sido necesarios para el buen funcionamiento del diseño. Se ha investigado sobre diferentes errores que se han producido en los programas de Xilinx, además del diseño

de bloques IP, aunque finalmente no ha sido necesario su uso, pero si me gustaría remarcarlo.

Se ha investigado y aprendido, también, sobre la elección de diferentes componentes para el diseño, estudiando sus *datasheets*. También, en esta línea, se ha aprendido sobre el diseño PCB, diseñando una PCB propia, utilizando el entorno de diseño Altium. Se ha investigado sobre la correcta elección de los diferentes parámetros, además de realizar en algunos casos *footprints* propios y en otros emplear los del fabricante.

Por último, pero no menos importante, se ha profundizado en el protocolo de comunicación VME. Se ha aprendido el funcionamiento de éste, y la función de todas las señales que son necesarias para realizar la comunicación. Se ha aprendido sobre todos sus modos de direccionamiento, aunque, como he dicho antes, todos no se verán reflejados en este proyecto.

Una vez explicado todo lo aprendido en este proyecto, paso a hacer una breve introducción de los capítulos que se verán en este documento.

En el segundo capítulo, después de esta introducción, sobre el estado del arte, se va a realizar una explicación detallada sobre que es el bus VME. En él se van a explicar las diferentes señales que se emplean en la comunicación, además de explicar las comunicaciones que se van a implementar en el diseño. Además de esto, en este capítulo se explica la metodología de diseño empleada, la placa que se va a emplear y su justificación, y qué funcionalidades se van a utilizar de los diferentes programas de Xilinx, necesarios para el diseño del prototipo.

En el tercer capítulo, se explica cómo se ha realizado el diseño del esclavo. Cabe destacar que aquí se quiere comentar es cómo se ha realizado el diseño y por ello no se transcribirá el código desarrollado. Por tanto, lo que se expone es una explicación con palabras de cómo se ha realizado el diseño VHDL, tanto parte combinacional, como secuencial, explicando la máquina de estado que se ha implementado. Además, se explica el diseño de bloques necesario para el funcionamiento, y una explicación del bloque de memoria aportado por Xilinx, y como acceder a él, tanto desde la parte FPGA como desde la parte del microprocesador.

En el cuarto capitulo, se realiza la verificación del diseño. En este capítulo se explican las simulaciones necesarias para dejar claro que el diseño funciona correctamente, en las diferentes comunicaciones implementadas. Además, para reforzar esta idea, se ha añadido el ANEXO I en el cual se complementa la verificación, con imágenes de la simulación de todos los modos de comunicación que han sido implementados.

En el capítulo quinto se explica el uso del adaptador de tensión [5], necesario para poder comunicar el prototipo con el maestro real. En este se explica por qué se ha empleado ese adaptador y como ha sido realizado el diseño PCB necesario para poder implementarlo dentro del diseño real.

En el sexto capitulo se realiza la demostración. Para ello, se ha tomado el maestro diseñado por el compañero, y se han enfrentado ambos, para comprobar que las diferentes comunicaciones funcionan perfectamente. También se ha empleado el adaptador de tensión, con algunos problemas que serán comentados en el propio capitulo.

Por último, se extraen algunas conclusiones, donde se ha realizado un resumen de lo realizado aquí, y donde se ha justificado la demostración de que el trabajo ha sido realizado.

Una vez explicado qué se va a diseñar en este trabajo, qué se ha aprendido para poder implementar el diseño, y cómo se va a distribuir la diferente información recogida aquí, doy comienzo a la explicación detallada, capitulo a capitulo, del diseño, comenzando por el capítulo dos.

Capítulo 2. Estado del arte

En este segundo capítulo se va a explicar el bus VME, ya que para poder entender que se ha hecho, es necesario, primero saber qué se va a diseñar, además de cómo se va a diseñar. Por eso, la segunda parte del capítulo estará dedicará a la metodología de diseño del esclavo.

2.1 Bus VME.

El bus VME es un bus asíncrono multimaster, el cual se utiliza para poder leer o escribir en diferentes esclavos. Es un bus que fue desarrollado para los microprocesadores Motorola 6000, pero que hoy en día se encuentra en muchas otras aplicaciones [8]. En él se encuentra la figura de los maestros y los esclavos. Un maestro en este protocolo de comunicación es un elemento el cual actúa como; es decir, controla a los esclavos mediante una serie de señales de control. Los esclavos del bus son aquellos elementos a los cuales va la información que lanza el máster, o, de donde proviene la información, y es elegido aquel con el que se quiere comunicar mediante un bus de direcciones. El esclavo a su vez presenta una memoria interna donde se guardarán los datos necesarios y de donde se sacarán después por el maestro. Esta memoria es big endian, y está compuesta por bloques de 4 bytes. Cada esclavo presenta un rango de direcciones de memoria. Es importante este concepto ya que la transmisión de los diferentes bytes de la memoria se hará de diferentes maneras. Fuera aparte del bus, en este caso, esta memoria estará conectada con otros elementos que puedan escribir y leer desde la parte C dentro del mismo esclavo, pero esta explicación se dejara para más adelante, ya que forma parte del diseño.

También es necesario que se sepa que existen varios modos de funcionamiento, y de envío de datos, los cuales son elegidos mediante diferentes señales, pero los cuales se recogen en dos grandes grupos: transferencia de ciclo único, y por bloques. La transferencia de ciclo único, como su propio nombre indicar, consiste en enviar los datos que se quieran de una única vez, mientras que la transferencia por bloques es aquella donde los datos se envían en varios ciclos.

Las señales anteriormente mencionadas se encargan de controlar el modo en el que se va a trabajar, es decir, si se quiere leer o escribir, la dirección en la que se quiere tratar los datos, además de señales que gobiernen el proceso. De este modo, comenzaré explicando las diferentes señales que lo componen, empezando por las de control:

- AS: Señal activa en baja que informa a todos los esclavos que la información de A, LWORD y AM se puede tomar. Además, esta señal indicará el inicio de la transmisión. La señal se mantendrá a uno en su estado de reposo. Cuando el maestro que esté hablando quiera iniciar una transmisión, la pondrá a cero, y así los esclavos empezará a escuchar para luego, mediante la señal de Address, saber cuál es el esclavo con el que se quiere comunicar el maestro. Una vez acabada la transmisión en el caso de la transferencia de bloques, ésta se pondrá a uno de nuevo, si es de ciclo único, puede marcar el final de la transmisión, o lo marcaran los Data Strobe. Cabe destacar también que, en la transferencia por bloques, esta señal es la que marca el tamaño de la palabra, pero esto será explicado más adelante. En el diseño del esclavo, esto es una señal de entrada.

- DS0 y DS1: Bus de dos datos que indican que, si se está en modo escritura, los datos situados en el bus de datos son válidos, y si se está en modo lectura, indica que el esclavo puede ya situar la información en el bus de datos. Además, Los data strobe añaden información sobre los bytes que se quieren leer del bus de datos. Estos están siempre a uno, y en el momento que se uno de los dos se pone a cero, se sigue con el proceso. Cabe destacar que, si AS no se ha puesto a 0 antes, estos no se deben de poner a cero. Sus funciones complementarias serán explicadas más abajo, que serán dependientes de sus cuatro estados (“00”, “01”, ”10”, “11”). En el diseño del esclavo, este bus es de entrada. Dentro de la transmisión, pueden indicar el fin de esta.

- DTACK: Señal activa en baja que indica que los datos han sido recibidos, en el caso de escritura, o mandados, en el caso de lectura, de manera correcta En su estado de reposo se encuentra a uno. Cuando los datos se reciban o se escriban de manera correcta, se pone a cero, hasta que los DS, que indica el fin del tratamiento de datos, se pongan a uno, y así, esta señal se pondrá a uno también. Está gobernada por el esclavo, por lo que en su diseño será una salida.

- WRITE: Señal que indica si se quiere leer o escribir en la memoria del esclavo. Para indicar escritura en el esclavo, WRITE se pone a cero, y para indicar lectura del esclavo, se pone a uno. Está gobernada por el maestro, por lo que en el diseño del esclavo será una entrada.

Una vez explicadas las señales de control, doy paso a explicar los buses tanto de datos, como aquellos que corresponden a la dirección y a conocer el modo de funcionamiento:

- A[31:1]: Este bus será el encargado de indicar en qué dirección se quiere realizar la acción. De esta manera, cada esclavo tendrá asociada un número de direcciones concretas, para que así todos puedan escuchar la dirección que el maestro pone en el bus, y el que esté dentro del rango será el que realice la acción. Cabe destacar que solo podrá participar un esclavo por transmisión. El rango de memoria al que pertenece el esclavo se podrá elegir tanto en el diseño del esclavo, como físicamente mediante switches. Este bus será gobernado por el maestro, por lo que en el diseño del esclavo será una entrada.

- AM[5:0]: El nombre de este bus se refiere a *address modifiers*. Su funcionalidad es indicar cuántos bits se tienen que mirar de la dirección, además de indicar el modo de funcionamiento para el esclavo en el ciclo de transmisión correspondiente. Esta señal la gobierna el maestro, por lo que en el diseño del esclavo será una entrada. Más adelante se explicará cómo funcionan con respecto a los modos de funcionamiento y los bits de direcciones. Hay muchos modos de direccionamiento diferentes, sin embargo, no todos serán implementados, ya que la finalidad de este proyecto no lo requiere.

- D[15:0]: Este bus será el encargado de transmitir los datos por los diferentes sistemas que estén conectados. Es un bus bidireccional, y por tanto, en el diseño quedará configurado como de entrada y salida.

Una vez que se tiene ya conocimiento sobre todas las señales del bus, se va a pasar a explicar cómo se pueden seleccionar los modos de funcionamiento, los datos a tomar y el número de direcciones en función de todas estas señales. Así, para poder ejemplificar mucho mejor, se va a hacer uso de tablas.

En primer lugar, vamos a ver los distintos modos de funcionamiento. Para poder saber en cual estamos, tenemos que mirar los AM (Address Modifier). Para verlo más claro, pasaremos a ver la tabla 2.1:

ADDRESS MODIFIER							
HEX CODE	5	4	3	2	1	0	FUNCTION
3F	H	H	H	H	H	H	A24 supervisory block transfer (BLT)
3E	H	H	H	H	H	L	A24 supervisory program access
3D	H	H	H	H	L	H	A24 supervisory data access
3C	H	H	H	H	L	L	A24 supervisory 64-bit block transfer (MBLT)
3B	H	H	H	L	H	H	A24 non privileged block transfer (BLT)
3A	H	H	H	L	H	L	A24 non privileged program access
39	H	H	H	L	L	H	A24 non privileged data access
38	H	H	H	L	L	L	A24 non privileged 64-bit block transfer (MBLT)
37	H	H	L	H	H	H	A40BLT
36	H	H	L	H	H	L	Reserved
35	H	H	L	H	L	H	A40 lock command (LCK)
34	H	H	L	H	L	L	A40 access
33	H	H	L	L	H	H	Reserved
32	H	H	L	L	H	L	A24 lock command (LCK)
31	H	H	L	L	L	H	Reserved
30	H	H	L	L	L	L	Reserved
2F	H	L	H	H	H	H	Configuration ROM/Control & Status Register (CR/CSR)
2E	H	L	H	H	H	L	Reserved
2D	H	L	H	H	L	H	A16 supervisory access
2C	H	L	H	H	L	L	A16 lock command (LCK)
2B	H	L	H	L	H	H	Reserved
2A	H	L	H	L	H	L	Reserved
29	H	L	H	L	L	H	A16 non-privileged access
28	H	L	H	L	L	L	Reserved
27	H	L	L	H	H	H	Reserved
26	H	L	L	H	H	L	Reserved
25	H	L	L	H	L	H	Reserved
24	H	L	L	H	L	L	Reserved
23	H	L	L	L	H	H	Reserved
22	H	L	L	L	H	L	Reserved
21	H	L	L	L	L	H	Reserved
20	H	L	L	L	L	L	Reserved

L = low-signal level H = high-signal level

ADDRESS MODIFIER							
HEX CODE	5	4	3	2	1	0	FUNCTION
1F	L	H	H	H	H	H	User-defined
1E	L	H	H	H	H	L	User-defined
1D	L	H	H	H	L	H	User-defined
1C	L	H	H	H	L	L	User-defined
1B	L	H	H	L	H	H	User-defined
1A	L	H	H	L	H	L	User-defined
19	L	H	H	L	L	H	User-defined
18	L	H	H	L	L	L	User-defined
17	L	H	L	H	H	H	User-defined
16	L	H	L	H	H	L	User-defined
15	L	H	L	H	L	H	User-defined
14	L	H	L	H	L	L	User-defined
13	L	H	L	L	H	H	User-defined
12	L	H	L	L	H	L	User-defined
11	L	H	L	L	L	H	User-defined
10	L	H	L	L	L	L	User-defined
0F	L	L	H	H	H	H	A32 supervisory block transfer (BLT)
0E	L	L	H	H	H	L	A32 supervisory program access
0D	L	L	H	H	L	H	A32 supervisory data access
0C	L	L	H	H	L	L	A32 supervisory 64-bit block transfer (MBLT)
0B	L	L	H	L	H	H	A32 non privileged block transfer (BLT)
0A	L	L	H	L	H	L	A32 non privileged program access
09	L	L	H	L	L	H	A32 non privileged data access
08	L	L	H	L	L	L	A32 non privileged 64-bit block transfer (MBLT)
07	L	L	L	H	H	H	Reserved
06	L	L	L	H	H	L	Reserved
05	L	L	L	H	L	H	A32 lock command (LCK)
04	L	L	L	H	L	L	A64 lock command (LCK)
03	L	L	L	L	H	H	A64 block transfer (BLT)
02	L	L	L	L	H	L	Reserved
01	L	L	L	L	L	H	A64 single transfer access
00	L	L	L	L	L	L	A64 64-bit block transfer (MBLT)

L = low-signal level H = high-signal level

Tabla 2.1: Modos de transmisión en función de los AM. Fuente:<ANSI/VITA 1-1994 (S2011)>.

En la tabla 2.1 se pueden observar todos los modos de funcionamiento del bus. Se puede ver como se agrupan en 4 grupos. Estos grupos estarán marcados por la cantidad de bits del bus de direcciones que se va a emplear. De esta manera, podrán ser A16, A24, A32 y A40, aunque en el diseño no se van a implementar cuarenta bits de direcciones, por lo que solo se emplearan tres de estos grupos. También se observa cómo hay muchas direcciones reservadas y que quedan a definición del usuario, pero en este caso, esas no se van a emplear. Cada uno de estos tres grupos, a su vez, se agrupan en supervisor, non privileged (user). Estos dos modos simplemente dictan a que direcciones de memoria se puede acceder y a cuáles no. Como esto queda en manos del usuario final, no se ha implementado, ya que, no interfiere implícitamente en la transmisión. En el caso de que se quisiera emplear, simplemente habría que definir donde se puede acceder o no. A su vez, esta subdivisión, se dividen en los cuatro modos

de funcionamiento esenciales, que serán la transferencia de bloques, tanto de treinta y dos bits como de sesenta y cuatro bits, y la transferencia de ciclo único, tanto de program access, como de data Access. En este caso, en cuanto a la diferencia de program y data, simplemente varia en cuando a la instrucción que se manda. De esta manera, le pasa igual que a supervisor/non privileged, por lo que queda en manos del usuario definir las instrucciones, y no se va a poner en práctica en este caso, quedando así, las dos transmisiones iguales. Tampoco se va a emplear la transmisión por bloques de 64 bits, ya que el multiplexado no se va a emplear. Por tanto, las transmisiones que se van a emplear en este diseño van a ser la transmisión por bloques de treinta y dos bits, en todas sus formas, y las de ciclo único, de igual manera.

Pasamos ahora a ver cómo saber los datos que se van a tomar del bus. Para poder saber eso, tenemos que ver LWORD, DS, A[1] y A[2].

LWORD	DS	A(1)	Modo de funcionamiento
1	01	0	1 byte, byte [0]
1	10	0	1 byte, byte [1]
1	01	1	1 byte, byte [2]
1	10	1	1 byte, byte [3]
1	0	0	2 bytes, byte[0-1]
1	0	1	2 bytes, byte[2-3]
0	0	0	4 bytes, byte[0-3]
0	01	0	unaligned 1, byte[0-2]
0	10	0	unaligned 2, byte[1-3]
0	0	1	unaligned 3, byte[1-2]

Tabla 2.2: Datos en función de DS, A(1), LWORD y A(2).

En primer lugar, para poder entender la tabla 2.2, es necesario entender la columna Modo de funcionamiento. El primer valor define como se van a transmitir los bytes. un byte, para las transmisiones de un solo byte, dos bytes para las de dos, y cuatro para las de cuatro. En el caso de la desalineadas, simplemente se hace referencia a los tres modos que hay. Además, el elemento siguiente dice qué byte o bytes se toman de la memoria, ya que esta se divide en cuatro bytes.

De esta manera, se puede observar en la tabla 2.2 los diferentes datos que se toman del bus. Es importante remarcar como estos se dividen por bytes, ya que, se va a

observar más adelante en la tabla 2.3 cómo se distribuyen dentro del bus estos bytes. También cabe destacar que, si se mira la tabla 2.2, se puede observar como el A(1) nos marca en que parte de los cuatro bytes estamos. Se puede asegurar que la distribución de la memoria es en grupos de cuatro bytes, y que por eso la dirección descarta el bit 0, ya que este no presenta ningún valor a la hora de obtener los datos de esta. El tratamiento de los datos queda en manos de los DS, y del A(1), ya que la dirección marcará si estamos en la parte alta o baja de la memoria, y los Data Strobe si estamos en el primer byte o segundo de cada parte.

Lo anteriormente explicado sería en el caso en el que los datos estén alineados, es decir, se tome o el bloque entero, o una parte u otra de él. En el caso de tomar datos desalineados, se puede observar en la tabla 2.2 como la cosa cambia, ya que lo que se hace es que, si se toman tres datos, el A(1) se sitúa a cero, y si se toman dos, se sitúa a uno. Además, en este caso, los Data Strobes no funcionan como con los alineados, por lo que, a la hora de transmitir, habrá que tenerlo en cuenta.

During the following types of cycles...	the data lines are used to transfer data as shown below:			
	D[31..24]	**D[23..16]**	**D[15..8]**	**D[7..0]**
Address-Only	<---------- no bytes transferred ---------->			
Single even byte transfers				
Byte(0) Read, Write or RMW			Byte(0)	
Byte(2) Read, Write or RMW			Byte(2)	
Single odd byte transfers				
Byte(1) Read, Write or RMW				Byte(1)
Byte(3) Read , Write or RMW				Byte(3)
Double byte transfers				
Byte(0-1) Read, Write or RMW			Byte(0)	Byte(1)
Byte(2-3) Read , Write or RMW			Byte(2)	Byte(3)
Quad byte transfers				
Byte(0-3) Read, Write or RMW	Byte(0)	Byte(1)	Byte(2)	Byte(3)
Single byte block transfers				
Single Byte Block Read or Write			<-------- Note 1 -------->	
Double byte block transfers				
Double Byte Block Read or Write			<--------- Note 2 -------->	
Quad byte block transfers				
Quad Byte Block Read or Write	Byte(0)	Byte(1)	Byte(2)	Byte(3)
Unaligned transfers				
Byte(0-2) Read or Write	Byte(0)	Byte(1)	Byte(2)	
Byte(1-3) Read or Write		Byte(1)	Byte(2)	Byte(3)
Byte(1-2) Read or Write		Byte(1)	Byte(2)	

Tabla 2.3: Distribución de los datos, en el bus de datos. Fuente<ANSI/VITA 1-1994 (S2011)>.

La tabla 2.3 muestra cómo van a estar los diferentes bytes a la hora de la transmisión. Cabe destacar que, en la primera columna, se dice que tipo de transmisión se está realizando, y en la segunda, como quedan los bytes distribuidos. Que los bytes tengan un número es debido a que hacen alusión a la memoria interna del bus.

En la tabla 2.3 se puede observar cómo se distribuyen los bytes en el bus en función del modo de transmisión y de qué parte del bloque de treinta y dos bits de memoria se quiera obtener. Además, se puede observar que el bus es big endian.

La tabla 2.3 muestra varios aspectos a tener en cuenta. En aquellos casos donde se manden bytes tanto de la parte alta, como de la parte baja de la memoria, los bytes se

distribuyen a lo largo de los cuatro bytes del bus en su posición. Sin embargo, en las transmisiones alineadas de dos y un bytes, esto no es así, ya que se puede observar, empezando por las transmisiones de dos bytes, que estos se situarán entre [15-0]. También, en la transmisión de un byte, se podrán situar en [15-8] o en [7-0], dependiendo del byte a transmitir. Esto quedara mucho más claro en el capítulo cuatro de verificación, donde analizaremos cada una de las transmisiones, y cómo funcionan.

Para que se entienda de una manera más clara lo anteriormente explicado, he resumido todo en la tabla 2.4, además de añadirle el A(0), ya que, en mi opinión, aunque el bus lo descarte, es necesario para poder entender cómo funciona el bus de datos, y las diferentes transmisiones.

Modo de funcionamiento	LWORD	DS	A(1)	A(0)	D[31-24]	D[23-16]	D[15-8]	D[7-0]
1 byte, byte [0]	1	1	0	0			byte[0]	
1 byte, byte [1]	1	2	0	1				byte[1]
1 byte, byte [2]	1	1	1	0			byte[2]	
1 byte, byte [3]	1	2	1	1				byte[3]
2 bytes, byte[0-1]	1	0	0	0			byte[0]	byte[1]
2 bytes, byte[2-3]	1	0	1	0			byte[2]	byte[3]
4 bytes, byte[0-3]	0	0	0	0	byte[0]	byte[1]	byte[2]	byte[3]
unaligned 1, byte[0-2]	0	1	0	0	byte[0]	byte[1]	byte[2]	
unaligned 2, byte[1-3]	0	2	0	0		byte[1]	byte[2]	byte[3]
unaligned 3, byte[1-2]	0	0	1	1		byte[1]	byte[2]	

Tabla 2.4: Resumen de la distribución del bus de datos, y las diferentes transmisiones,

Cabe destacar que, para la transmisión de bloques, la distribución es igual. De esta manera, el maestro deberá de ir cambiando los Data Strobes en función de cómo vaya la transmisión, y quedará en manos del esclavo el tratamiento de la dirección.

Paso ahora a explicar el funcionamiento del bus [7]. Para poder observar de mejor manera que tiene que ocurrir en él, se va a ilustrar con imágenes.

En primer lugar, se va a explicar el protocolo de direccionamiento. Este protocolo se basa en que el maestro tiene un tiempo máximo desde que lanza AM, LWORD y A, y pone el AS a 0, y el esclavo tiene un tiempo máximo para poder procesar estas señales. Esto queda reflejado en la figura 2.1:

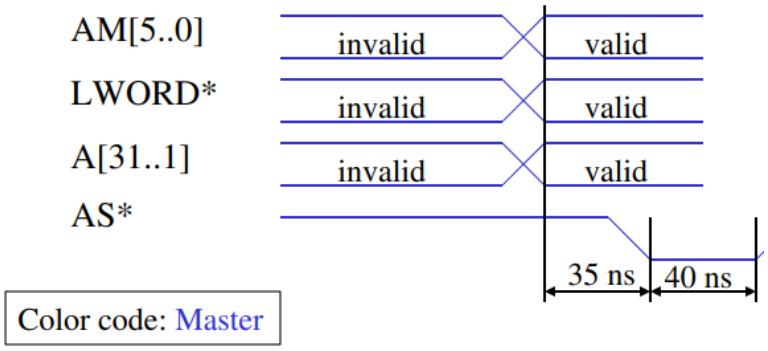


Figura 2.1: Protocolo de direccionamiento en VME. Fuente: <https://www.sgo.fi/~jussi/eiscat/ChannelBoardCourse/3_VME_presentation.pdf>

El máster no puede saber si el esclavo ha recibido bien estas señales, por lo que este seguirá con el proceso hasta recibir el DTACK. Si no se recibe este, se recibirá BERR, pero esta señal no interesa en nuestro diseño, ya que es una señal para gobernar el maestro, y este diseño solo se basa en el esclavo. Es muy importante, que, como se ve en la figura 2.1, toda la información del bus de A, LWORD y AM quede definida antes del AS, ya que, si no es así no funcionara de manera correcta, debido a que, en cuanto el AS se ponga a cero, se comenzará con el tratamiento de los datos y se producirán errores.

Una vez entendido cómo funciona la recepción de la dirección, pasó a explicar cómo funciona el proceso completo. Para ello comenzaré mostrando la figura 2.2:

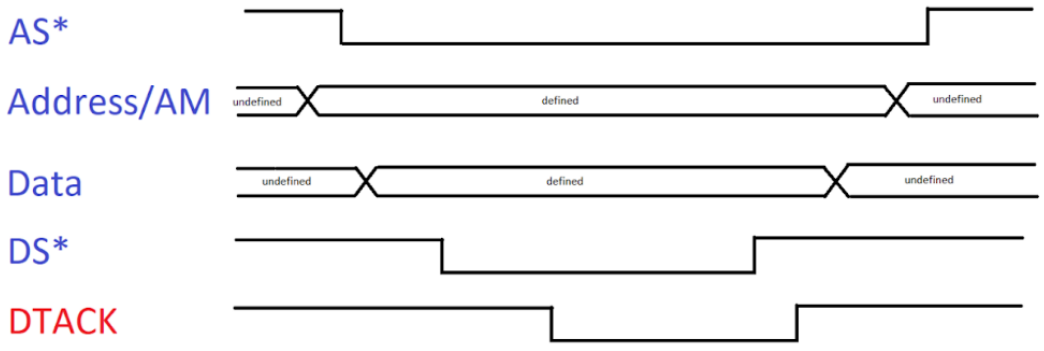


Figura 2.2: Proceso de escritura de datos en una transferencia single cycle.

La figura 2.2 recoge lo que tiene que ocurrir en un proceso de escritura de datos en el esclavo, en un proceso de ciclo único. En primer lugar, el maestro tiene que definir de manera correcta el bus A, el AM y la señal LWORD (como vimos en la figura 2.1). Una vez definidos, se puede empezar con la transmisión. Se comenzará pasando el AS se

su estado de reposo a cero. Una vez que el AS cambie (se comience la transmisión), el maestro sitúa los datos de manera correcta en el bus de datos (D). Cuando los datos hayan sido correctamente mandados, el maestro cambia los DS de “11”, al valor que se requiera (dependiendo de los datos que se quieran sacar). Esto indica al esclavo que los datos han sido correctamente situados, por lo que este los capta, y el esclavo pone el DTACK a cero, comenzado con el tratamiento de los datos. El esclavo cuenta de un tiempo, que lo marcará el maestro con los DS, para poder tomar los datos. Cuando este tiempo concluya, el maestro cambiará los DS a su estado de reposo. Una vez esto ocurra, el DTACK pasa a uno, y cuando el maestro reciba el DTACK a 1, cambiará el AS a uno, terminando así la transmisión. Cabe destacar, que el WRITE no queda reflejado en la figura 2.2, pero que, igual que el A, AM y LWORD, se situará a cero antes de iniciar la transmisión.

Una vez explicada la escritura de los datos en el esclavo, pasaremos a explicar la lectura de los datos.

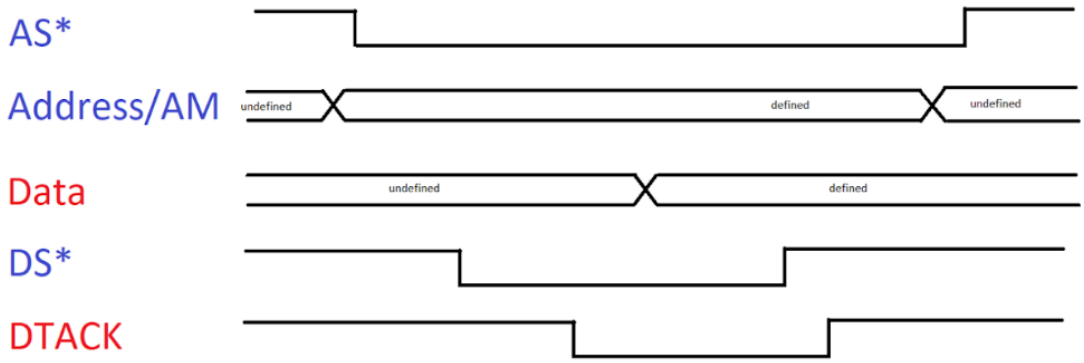


Figura 2.3: Proceso de lectura de datos en una transferencia single cycle.

En la figura 2.3 se observa que ocurre en un proceso de lectura de datos del esclavo en un proceso de ciclo único. Se puede observar cómo las señales del maestro funcionan exactamente igual que antes, por lo que no serán comentadas de nuevo. Sin embargo, donde se tiene que situar el foco es el bus de datos. Este bus, en la lectura, será gobernado por el esclavo, ya que este se encargará de mandar los valores al maestro para que este los trate. De esta forma, el bus de datos se actualizará una vez el DTACK pase a cero. De esta manera, una vez que el maestro procese los datos y los guarde, pondrá los DS a uno, y continuará la transmisión como antes. Igual que se comentó antes, en la figura 2.3 no queda reflejado WRITE, pero este se situará a uno a la misma vez que A, AM y LWORD.

Cabe destacar que, en esta explicación, se ha puesto que el AS se ponga el ultimo a uno. Esto no tiene por qué ser así, ya que los Data Strobes también pueden indicar el fin de la transmisión. Esto dependerá del tiempo que se tarde en tomar los datos, y del funcionamiento del maestro. No obstante, en la NORMATIVA, hay veces que se pone a uno antes uno u otro, por lo que el diseño se ha realizado para que esto pueda ocurrir. En el único caso en el que el AS se pone a ubo el ultimo será en la transferencia de bloques, ya que marca el final. En los ejemplos reales se podrá observar como el diseño funciona en ambos casos de manera correcta.

Por último, queda por explicar cómo funciona el bus en las transmisiones de bloques. Para conocer cómo funciona, se tiene que entender varias cosas antes.

En primer lugar, se debe saber que la señal AS será la encargada de gobernar cuántos bytes se quieren sacar de memoria, es decir, cuantos ciclos va a durar. Así, el maestro pondrá la señal AS a cero, y después de esto irá poniendo DS a “11” y al valor que se quiera cada ciclo, hasta poner AS a uno, que indicará que se termina la transmisión. Por tanto, se puede decir que los DS gobernaran las subtransferencias de la transferencia, mientras que el AS será el que gobierne la transferencia global.

Otro detalle importante es que hay que saber que la transmisión de bloque se hace de treinta y dos o sesenta y cuatro bits, aunque el diseño solo realizará la de 32 bits (en la demostración sólo quedará definida para 16 bits, debido a que la placa de desarrollo sólo presenta dieciséis bits bidireccionales). Para poder transmitir los 64 bits, se hará un multiplexado entre el bus de datos y el bus de direcciones. También se tiene que saber que existe un número máximo de bytes transmitidos, siendo doscientos cincuenta y seis bytes para las transmisiones de bloques de treinta y dos bits, y mil veinticuatro bytes para aquellas transmisiones de sesenta y cuatro bits. Una vez conocidos estos parámetros, se pasará a ver una transmisión de bloques real:

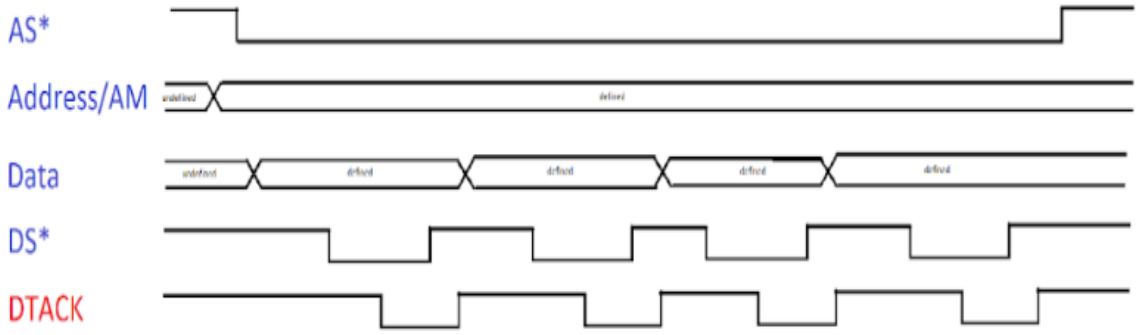


Figura 2.4: Proceso de escritura de datos en una transferencia block transfer.

En la figura 2.4 se puede observar la transmisión por bloques, en el caso en el que se esté escribiendo en la memoria del esclavo. Además, se observa lo comentado anteriormente, que básicamente sería que esta transmisión es gobernada por AS, y cada ciclo lo marca los DS. Cabe destacar que el proceso en el que la dirección tendrá que aumentar (ya que cada dato se tendrá que ir escribiendo en una dirección de memoria superior a la anterior) será realizado por el esclavo, por lo que, más adelante, cuando se vea el diseño de este, se explicará cómo se realiza esto. Por último, se muestra la figura 2.5:

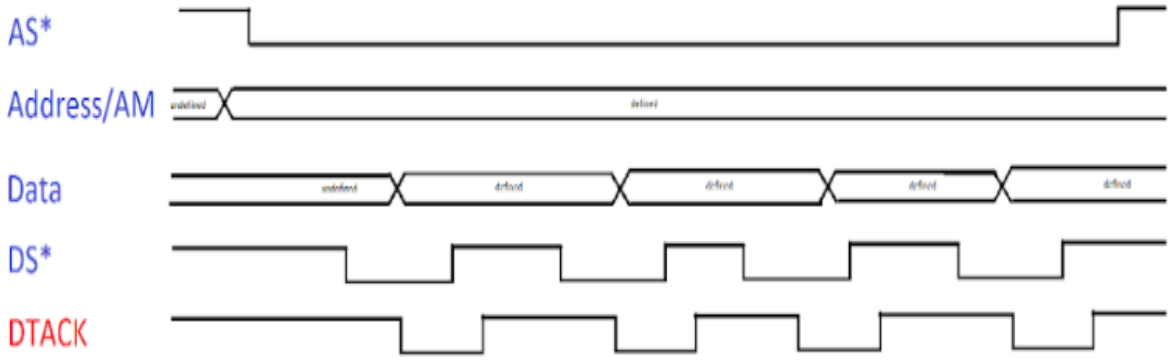


Figura 2.5: Proceso de lectura de datos en una transferencia block transfer.

Básicamente lo mismo que se ha hablado con la escritura, solo que, como es lectura, los datos los gobierna el esclavo, y se sitúan en el bus de datos después de que DTACK se active.

Por último, para poder entender cómo funcionan los Data Strobe anteriormente mencionado, se va a incluir el ejemplo de transmisión de un byte, ya que en las otras dos, no cambian los DS, sino A(1):

		DS1*	DS0*	A1	LWORD*
First data transfer	Byte (2)	low	high	high	high
	Byte (3)	high	low	X	X
	Byte (0)	low	high	X	X
	Byte (1)	high	low	X	X
Last data transfer	Byte (2)	low	high	X	X

X = high or low.

Figura 2.6: Funcionamiento de los Data Strobes en la trasmisión por bloques de un byte. Fuente: <ANSI/VITA 1-1994 (S2011)>.

Aquí se observa cómo, el primer byte que se transmite depende de A(1) y de los data strobe, y cómo se comportan los data strobe a lo largo de la transmisión.

En este subapartado se ha comentado un pequeño resumen de todas las funcionalidades y modos de direccionamiento del bus VME, los cuales no serán implementados en su totalidad en este diseño. Más adelante, cuando se explique el diseño y se vea su funcionalidad completa, muchos de estos conceptos serán aclarados.

2.2 Metodología del diseño.

En este apartado se va a explicar con que se ha diseñado el proyecto, y qué elementos han sido necesarios para ello. De esta manera, se va a comenzar a hablar sobre el que se ha empleado.

Debido a que ha sido necesario emplear tanto una parte hardware, como una parte software, se optó por emplear un SoC. Un SoC (System on Chip) es un chip que integra

tanto la parte hardware (FPGA), como la parte software (microcontrolador). De esta manera se puede integrar un diseño en VHDL, y conectarlos al microcontrolador, para después poder realizar un programa en C que nos permita aumentar sus funcionalidades. Voy a explicar en primer lugar que SoC ha sido empleado, y en segundo lugar, como se ha empleado este SoC, explicando que programas se han empleado, y como se ha conectado la parte hardware y software.

2.2.1 Placa de desarrollo

Debido a que el prototipo ha sido realizado para una empresa, se ha empleado el SoC que entrara dentro de sus propuestas, y que permitiera poder desarrollar el bus VME de manera correcta, es decir, que presentase las entradas y salidas necesarias para ello. De esta manera, la placa a utilizar fue la ZedBoard, desarrollada por AVENT. Esta placa cuenta con un chip FPGA XC7Z020-1CSG484CES EPP de la familia zynq 7000 desarrollado por Xilinx, el cual, será también el utilizado para el desarrollo del proyecto final propio de la empresa.

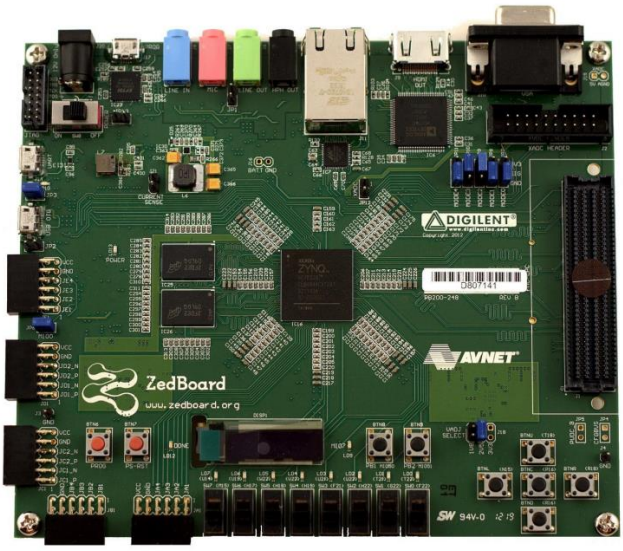


Figura 2.7: Placa de desarrollo.

Esta placa presenta un gran número de conectores que nos permiten realizar multitud de cosas. Sin embargo, hay que centrarse en los que interesan para nuestro diseño. Se comenzará hablando de los PMOD. Estos puertos permiten un uso bidireccional, por tanto, son idóneos para ser usados como el bus de datos. A la hora de usarlos, se debe tener en cuenta varios aspectos, como, por ejemplo, cómo se pueden usar los PMODs, y es que el puerto A y B son los únicos que están diseñados para señales single ended, por lo que serán los que se usarán en nuestro prototipo. Se valora también que, a la hora de realizar el diseño real, los puertos PMOD C y D se podrán emplear también para esta función, ya que presentan una forma diferencia, pero esta ha sido diseñada de manera externa, por lo que, conectándolos directamente con el pin que sea, se podrán

usar como entrada single-ended. Además, estos presentan doce pines, sin embargo, solo podemos usar 8 de cada, ya que tienen dos pines de GND y dos de Vcc.

El segundo puerto que va a ser usado en el diseño va a ser el puerto FMC [4]. Estos puertos son o de entrada o salida, no bidireccionales, por lo que, serán usados para las entradas y salidas (A, AM, LWORD, DS …) del circuito. La tensión de trabajo de este puerto puede ser seleccionada mediante el ajuste de Vadj. En nuestro caso, la tensión de trabajo será de 2,5V pudiendo llegar a 3,3V, aunque no es muy recomendable, por lo que para adaptar esta señal a 3,3V se utilizaran unos adaptadores de tensión, de los cuales se hablará más adelante. Para poder usar el puerto FMC, se utilizará una tarjeta especial que será la XM105.

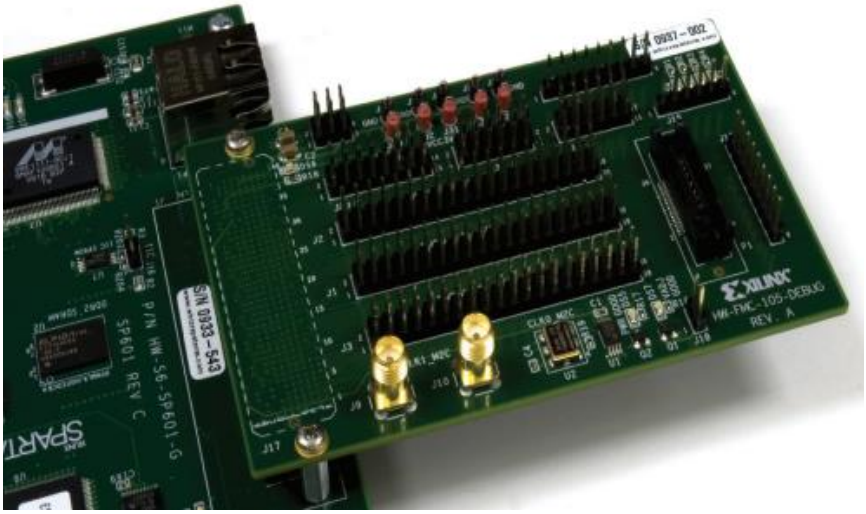


Figura 2.8: Imagen del XM105.

Esta placa se compone de varios conectores J. No obstante, esta placa también sirve para los HMC, por lo que muchos de estos conectores no se usarán. De esta manera, los que se emplean son el J20 y el J1 como entradas y salidas del diseño, quedará más claro cómo se conectan.

Es muy importante tener claro que, en el diseño real, como presentar más funcionalidades de las aquí descritas se emplearán prácticamente todos los posibles, ya que, en ese caso, al trabajar con el chip directamente, se podrán rutear las pistas como se quiera. Por ello también, se eligió este chip para el desarrollo, ya que permite un mayor aprovechamiento de los pines, y el no tener que usar varias FPGAs. No obstante, la funcionalidad real del diseño no tiene cabida en este TFG, pero sí que cabe destacar el porqué de la elección de esta placa de desarrollo.

2.2.2 Metodología de la utilización y programación de la placa.

En este apartado voy a hablar sobre cómo se han trabajado ambas partes del diseño, tanto el firmware, como el software, así como de su conexión entre sí.

Para la parte hardware se ha empleado Vivado. Esta es una herramienta que nos aporta Xilinx para poder obtener la total productividad de nuestros diseños, cuando se empleen productos diseñados por ellos, como en este caso, la familia Zynq-7000 SoC. Permite el diseño en VHDL, realizando la síntesis y la implementación, y generando el archivo .bit, que permite programar la placa de desarrollo.

Además de esto, permite la simulación del circuito, pudiendo observar cómo se comporta, y poder corregir de una manera más sencilla y visual los posibles fallos. También permite ver que está ocurriendo en el diseño cuando este esté funcionando, así como reconoce los errores en el código que se puedan tener a la hora de diseñar, así como errores que se tengan cuando se vaya a implementar.

Con Vivado podemos conectar de manera fácil las señales externas del diseño a las entradas y salidas del chip, mediante el archivo de constrains.

Por último, este nos permite la creación de un diseño de bloques, en el cual se pueden conectar diferentes bloques que el propio Vivado nos aporta con nuestros diseños, además de poder conectar el microcontrolador.

Con respecto a lo mencionado anteriormente, es importante explicar cómo se ha Realizado esta comunicación con el microcontrolador. Para ello se ha empleado el protocolo AXI4-Lite.

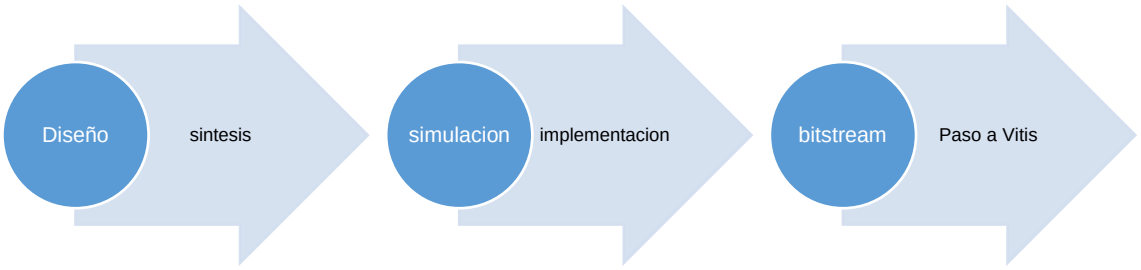
Este protocolo soporta las transmisiones tanto de 32 bits como de 64 bits, además de que transmiten todos los bits en cada transmisión. El acceso que hace este bus no se modifica y no se almacena, además de eliminar los accesos exclusivos.

Este protocolo es empleado en la comunicación con los diferentes bloques del diseño, y, además de este, uno de los bloques se comunica mediante la conexión directa de las diferentes señales de comunicación hacia el bloque del diseño VHDL.

Una vez explicado esto, pasamos a explicar cómo es el proceso por el cual se simula el diseño para determinar que este funciona de manera correcta. Antes de poder realizar esta simulación, es necesario realizar la síntesis del diseño, en donde nos indicaran los errores que haya en el código. Una vez realizada, se puede simular, para poder ver, en base a los bancos de pruebas que se hayan realizado, que va a ocurrir en nuestro circuito cuando este esté funcionando.

Una vez que se simula, es necesario obtener el bitstream. Este archivo puede ser programado ya en la placa, ya que presenta todas las características hardware que tiene el diseño, en el caso en el que solo se trabaje con hardware. Para poder pasar el diseño de Vivado a Vitis es necesario exportar la plataforma.

El flujo de trabajo queda reflejado en el siguiente diagrama de flujo:



Una vez exportada la plataforma, se obtiene el archivo xsa. Este archivo será la plataforma hardware que habrá que cargar en Vitis. De esta manera, se carga, y una vez se tenga, se construye, y se crea la aplicación. Aquí será donde se haga el desarrollo software del diseño. Además del desarrollo software como tal, es posible cambiar la configuración del PMU (Platform Managment Unit) firmware dentro de Vitis, además de poder añadir una configuración FSBL (Fisrt Stage Boot Loader), pudiendo así realizar las configuraciones que se quieran, y aumentando el abanico de posibilidades.

El propio entorno de Vitis permite programar la placa directamente, ya sea en modo depuración, lo que permite ir viendo paso por paso que está ocurriendo, y depurando el código, como directamente funcionado sin depuración.

Gracias al software suministrado por Xilinx, se tienen dos entornos que facilitan mucho el diseño en sistemas SoC, además de poder obtener gran funcionalidad de los mismos.

Una vez explicada la parte más teórica de este Trabajo de Fin de Grado, voy a pasar a la parte de diseño, donde se explicará que se ha realizado en el proyecto, y como se ha realizado, así como verificar que funciona, y demostrarlo mediante el prototipo físico.

Capítulo 3. Diseño del Esclavo del bus VME.

En este capítulo se va a explicar cómo se ha diseñado el esclavo de un bus VME, desarrollando paso por paso que se ha hecho, y los modos de funcionamiento que finalmente se van a emplear.

El esclavo VME básicamente lo que hará será, en primer lugar, esperar a que le lleguen las diferentes señales por el bus. Una vez AS se ponga a cero, este procesara los AM, LWORD y WRITE para ver el tipo de comunicación, y, además, si es el modo escritura, no hará nada, y si es modo lectura, sacara los datos de memoria. Una vez que esto ocurra, cambiaran los DS en función del modo de comunicación, y se situaran los datos en el bus D. Una vez realizado esto, este esperara a que AS y DS se pongan a uno nuevamente. En todo este proceso, diferentes señales que iran a una memoria interna deberán ir realizando el proceso necesario para poder obtener los datos de esta, o introducir datos nuevos.

Este diseño va a ser realizado mediante VHDL, en la herramienta de Xilinx, Vivado, además de que va a estar combinado con una pequeña parte C, que será para inicializar la memoria, en Vitis. Como se comentó con anterioridad, la placa de desarrollo donde se probará este prototipo será una Zedboard.

En una primera aproximación del diseño del bus VME, se empezó a diseñar en diferentes procesos a la vez, pero rápidamente se cayó en la cuenta de que una máquina de estado es la mejor opción de todas, ya que al ser un proceso que es gobernado por diferentes señales, la máquina de estado permite esperar en los estados hasta que estas señales cambien, y poder ir paso por paso. En este capítulo se explicarán tanto la máquina de estado, como las señales del circuito, sin olvidar todos los elementos necesarios para que se comuniquen la parte FPGA con el microcontrolador. Antes de comenzar la explicación del diseño, voy a comenzar explicando el bloque de memoria que se va a emplear, el cual ha sido imprescindible para el funcionamiento del prototipo.

3.1 Bloque de memoria RAM.

Necesito una memoria virtual que haga de buffer de los datos, la cual en C será comunicada con la memoria real. De esta manera, la memoria a utilizar será la BRAM. Este bloque de memoria permite la comunicación tanto con la parte hardware como con la parte software.
Para poder entender la comunicación del diseño hardware con la memoria, se necesita entender cómo funciona el bloque.

En primer lugar, se tiene que saber que la memoria cuenta con dos puertos de comunicación con esta. Se ha aprovechado esto para poder comunicar uno con la parte C, y el otro con el diseño hardware. Para poder conectar esta memoria con la parte C, hacen falta varios bloques auxiliares. Así, se ha utilizado el AXI BRAM Controller para

conectar el puerto A con este (Cabe destacar que este también tiene la posibilidad de tener dos puertos, aunque en este diseño solo se utilizara 1). Estos dos bloques quedarán comunicados mediante el bus BRAM_PORTA. Además, este bloque se conectará al reloj del microcontrolador, y el reset al sistema reset. El AXI BRAM Controller tendrá que conectarse al AXI Interconnect, que básicamente sirve para conectar los diferentes módulos con el procesador (en este caso, solo me ha hecho falta uno). No se entrará en más detalle del AXI Interconnect y del AXI BRAM Controller, ya que simplemente se han utilizado en su estado estándar, y no hemos tenido que hacer un estudio de su funcionamiento para poder emplearlos.

Una vez explicados todos los módulos necesarios para poder comunicar la memoria con la parte SoC, pasamos a ver como conectarlo a la parte hardware.

La conexión a la parte hardware es la más compleja, ya que no se interconectan bloques mediante buses, si no que se conectan directamente las señales de la memoria, a señales del circuito, por lo que, para poder entenderlo mejor, se pasa a explicar qué hace cada señal:

- clkb: Es el reloj correspondiente al puerto B. Todas las operaciones que se realicen en este puerto serán sincronizadas con él. En este caso, la señal que le llega sería la señal negada del reloj que gobierna nuestro diseño. Por tanto, están retrasadas medio ciclo entre sí. Esto se realiza para que la salida o entrada de datos se realice medio ciclo antes o después de que se tengan que situar, o se tengan que tomar los datos del bus, debido a que, el reloj que se niega es el mismo que se emplea en el bus.

- addrb: Bus [31:0] encargado de comunicar la dirección en la que se va a leer/escribir de la memoria. Se conecta en el diseño con addr_buf, que será la señal que contenga la dirección concreta con la que se quiere trabajar en el momento. Cabe destacar que nuestro bus de direcciones es de [31:1], sin embargo, esto no ha afectado, ya que el bus pasaría a los bits menos significativos, y el más significativo se rellenará con un cero. En la memoria es una entrada, por lo que addr_buf es una salida en mi circuito.

- dinb: Bus [31:0] donde se situarán los datos que querrán ser escritos en la dirección de memoria correspondiente. Estos se conectan con D_in que será el buffer encargado de mandar estos datos. Este bus es entrada en la memoria, por lo que, en mi diseño, D_in será una salida.

- doubt: Bus [31:0] donde se situarán los datos que querrán ser leídos en la dirección de memoria correspondiente. Estos se conectan con D_out que será el buffer encargado de introducir estos datos en el diseño hardware. Este bus es una salida de la memoria, por lo que D_out será una entrada en el diseño.

- enb: bandera encargada de comunicar si se quiere realizar una acción en la memoria, o no. Si enb es igual a cero, no se realiza ningún movimiento de datos, sin embargo, si esta pasa a ser uno, se realizará la acción que se quiera, en función de lo que dicten las demás señales. En el diseño se conectará con enable, la cual será una salida, ya que en el bloque de memoria es una entrada.

- web: Bus [3:0] el cual dicta si se quiere leer o escribir, y en el caso que se quiera escribir, que posiciones se quieren escribir. Si web es igual a “0000”, significa que la acción a realizar es leer. Sin embargo, si alguno de estos bits es igual a uno, se quiere escribir esa posición. Esto será explicado mejor, más abajo, cuando se explique el funcionamiento de la memoria, pero por hacer una leve introducción, como los datos se agrupan de cuatro bytes en cuatro bytes, cada byte de los cuatro que se pueden escribir de cada dirección es referido a un bit del web. También cabe destacar que cada byte se refiere a una dirección, pero que, sí se llama a una de las cuatro direcciones que contiene un bloque de memoria, sacará los cuatro bytes. En el bloque de memoria es una entrada, por lo que en el diseño es una salida.

- rstb: señal de reset que será conectada al reset global del sistema. Mucho ojo, es un reset activo en alta, por lo que tendrá que ser conectado con su correspondiente en el sistema de reset.

Una vez conocidas todas las señales involucradas en el proceso, paso a explicar cómo funciona. Se va a explicar el funcionamiento en base a las conexiones del puerto B, ya que el puerto A se controla desde la parte C, que, además, es mucho más fácil, y lo explicaré más adelante.

En primer lugar, tenemos que saber que la memoria se encuentra en el modo Write First Mode, lo que provoca que los datos de entrada se escriben simultáneamente en memoria y en el bus de datos de salida. De esta manera, una transmisión quedaría de la siguiente manera:

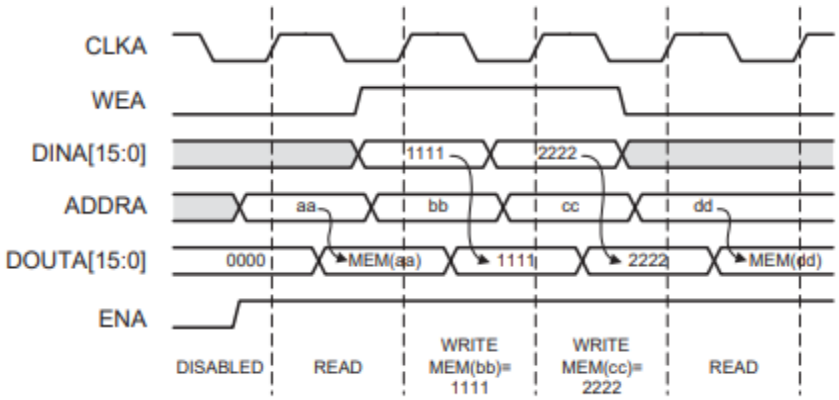


Figura 3.1: Proceso de escritura/lectura en memoria. Fuente <Block Memory Generator v8.4. LogiCORE IP Product Guide. Vivado Design Suite>.

En la figura 3.1 podemos observar como si el wea (en el caso de mi diseño concreto sería el b, pero este es un ejemplo de los documentos de Xilinx) está a cero, se saca la información de la dirección aportada por addra y se sitúa en douta, sin que dina intervenga. Cabe destacar que para que esto ocurra, ena tiene que estar a uno, y se produce con el flanco de reloj. Cuando wea pone a uno alguno de sus bits, se pasa a escribir en la memoria (en este caso, para el ejemplo pone todos los bits a uno), en la

dirección que marca adra. Además, el dato cambiado pasa a douta, lo cual nos puede ayudar a ver que se haya guardado de manera correcta.

El proceso de manejo de la memoria en la parte C, es decir, en Vitis, es más fácil, ya que simplemente controla el acceso el AXI BRAM Controller y, de esta manera, con las instrucciones recogidas en la carpeta "xil_io.h" podemos escribir sobre nuestra memoria. Además, se necesita conocer su dirección de inicio, la cual se encuentra en la librería "xparameters.h". También tenemos que tener en cuenta que aquí no habrá que controlar ni el wex ni el enx ni nada, por lo que no nos tendremos que preocupar si se escribe en una dirección que involucre dos direcciones. De esta manera, si en la parte C se quiere inicializar los primeros 200 datos de la memoria, se podría hacer de la siguiente manera:

```
#include <stdio.h>
#include "xparameters.h"
#include "xil_io.h"

#define bram_base XPAR_BRAM_0_BASEADDR

u16 almacenamiento_memoria[200];
int main()
{
	u16 write_data;
	u32 bram_address_write = bram_base;

	for(int loop = 0; loop<200; loop++)
	{
		write_data = 0xA2 + loop;
		Xil_Out8 (bram_address_write,write_data);
		bram_address_write = bram_address_write + 1;
	}

	bram_address_write = bram_base;

	while(1){
	}

}
```

Mediante un loop, se puede escribir en memoria, en cada dirección (cada dirección ocupa un byte).

Un tema muy importante que se debe tener en cuenta en nuestra memoria es que es little endian. Esto resulta un problema ya que el bus con que el vamos a trabajar es big endian. Así, cuando se tenga que escribir en la memoria, hay que invertir los datos, en grupos de 1 bytes, para que el tratamiento de estos con la memoria sea correcto. Vamos a ilustrarlo con un ejemplo:
El bus manda los siguientes datos: 0x11223344

Según la filosofía del bus, 0x11 corresponde al address 0x1, 0x22 al address 0x2, etc. Para que cada address se corresponda con el de nuestra memoria, éste tiene que entrar de la siguiente manera: 0x44332211

Por lo que internamente, en el buffer que será empleado para mandar los datos hacia la memoria será donde invertiré los datos.
Cabe destacar que cada dirección de memoria corresponde a un byte.
Otro detalle importante es que la memoria se almacena de cuatro en cuatro bytes. De esta forma, si se pide el dato correspondiente entre una de las 4 cuatro direcciones que ocuparía cada bloque de memoria, nos saca los cuatro bytes correspondientes. Veámoslo con un ejemplo. tenemos los siguientes datos almacenados en memoria:

0x0000050 -> 3- -- -- -0 7- -- -- - 4 B- -- -- -8 F- -- -- -C
90 A2 78 00 f3 FC 34 12 56 9d 67 a3 00 45 89 BC

Si se piden los datos del rango de address desde 0x54 a 0x57, doubt va a sacar los siguientes valores: 0xf3FC3412 (little endian). Internamente, si solo se quieren los valores de la dirección 0x55 y solo un byte, tendremos que eliminar los demás datos.

A la hora de escribir es exactamente igual, solo que se tiene que controlar además el web para que este funcione de manera adecuada, se escriba en las posiciones concretas que queremos.

Por último, se debe tener en cuenta que, en el dispositivo real, cuando se quiera comunicar la memoria virtual con la memoria real, habrá que tener en cuenta también aquí que la memoria del diseño es little endian y la memoria real es big endian.

El uso de esta memoria ya predeterminada por el propio entorno ha facilitado mucho el diseño de nuestro circuito, sin embargo, el funcionamiento de la memoria es algo complejo de entender, que nos ha demorado más tiempo de lo esperado. De ahí la importancia que se le ha dado a su correcta explicación. No obstante, cuando se explique el funcionamiento de nuestro diseño, quedará reflejado de manera más clara.

3.2 Señales y variables del diseño.

En primer lugar, para poder conocer con que trabaja el diseño aquí definido, se van a describir tanto entradas como salidas del circuito desarrollado, además de las señales internas y variables que se emplean en el proceso. Cabe destacar que no se va a mostrar el código directamente, sin embargo, para poder entender tanto la parte de diseño, como la parte de verificación y demostración, será necesario entender las diferentes señales que rigen los diferentes procesos, y más adelante, los propios procesos. De esta manera, comenzaré explicando las señales de entrada y salida pertenecientes al bus VME:

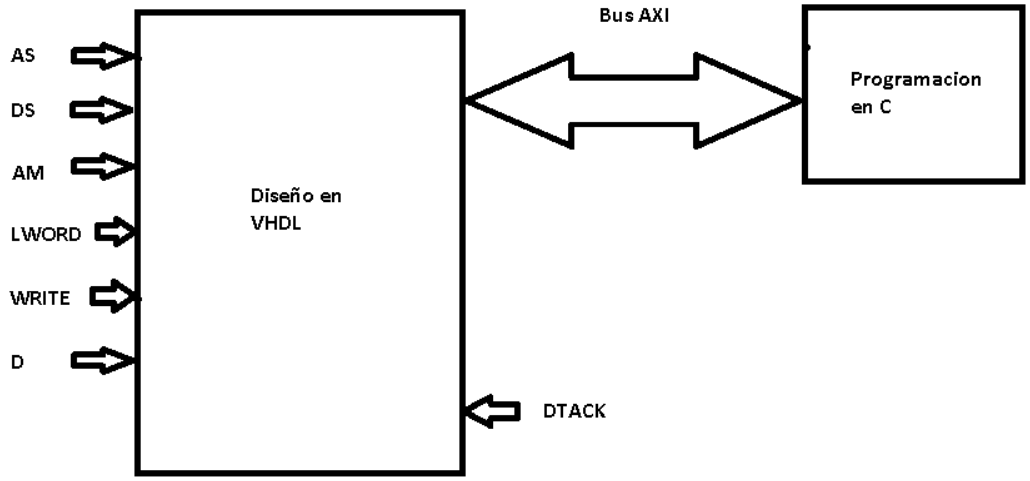


Figura 3.2: Entradas y salidas de nuestro diseño.

- A: Este bus es el referido al bus de direcciones del VME. Es un bus de entrada de treinta y dos bits. En el diseño real está conectado a la FMC de la placa.
- D: Este bus es el referido al bus de datos del VME. Es un bus bidireccional, de treinta y dos bits. En el diseño real están conectados a los PMODS de la placa.
- AM: Este bus es el referido a los address modifier. Es un bus de seis bits, siendo una entrada al esclavo. En el diseño real será conectado a la FMC de la placa.
- LWORD: Esta señal es una señal, de entrada, referida al LWORD del bus VME. En el diseño real será conectado a la FMC de la placa.
- WRITE: Esta señal es una señal de entrada, referida al WRITE del bus VME. En el diseño real será conectado a la FMC de la placa.
- AS: Esta señal es una señal de entrada, referida al AS del bus VME. En el diseño real será conectado a la FMC de la placa.
- DS: Este bus es referido a los data strobe. Es un bus de dos bits, siendo una entrada al esclavo. En el diseño real será conectado a la FMC de la placa.
- DTACK: Esta señal es una señal de salida, referida al DTACK del bus VME. En el diseño real será conectado a la FMC de la placa.

Si se quiere conocer más acerca de estas señales externas, véase el capítulo 2, el apartado 2.1 donde se explican de manera más extensa las diferentes funcionalidades

de estas, y su aplicación dentro del bus. Para conectar estas señales a las salidas externas de la placa que han sido definidas, se hará uso del archivo de constrains, sin embargo, no entraremos en detalle, ya que, a la hora del funcionamiento del bus, esto no afecta, y mientras se conecten en sus elementos (FMC o PMODs), se pueden conectar como se quiera. Una vez explicadas las señales referidas al bus VME, paso a analizar las señales referidas al bloque de memoria:

- D_in: Bus de entrada de treinta y dos bits, el cual se usa para comunicar los datos que salen de la memoria con el circuito. Esta es una entrada.

- D_out: Bus de salida de treinta y dos bits, el cual se usa para mandar los datos que recibe el bus, con la memoria.

- clk: Esta sería el reloj del sistema. Esta es una señal de entrada. Además, esta señal proviene del microcontrolador, siendo, básicamente, una señal de reloj de 100mhz, a la cual están conectados todos los relojes del sistema.

- reset: Esta sería la entrada de reset del sistema, siendo una señal std_logic. Esta señal estará conectada con el sistema de reset del diseño de bloques, y será activa en baja. La diferencia de esta señal, con el reset del bloque de memoria, es que el reset del bloque de memoria es activo en alta, aunque, realmente provienen de la misma señal.

- enable: Señal de salida que se usará para controlar la recepción o el envío de datos a la memoria. Es activa en alta. Esta es una salida.

- web_out: Bus de salida de cuatro bits, equivalente a la señal de web del bloque de memoria. Esta es una salida.

- addr_buf: Bus de salida de treinta y dos bits que se usa para comunicar la dirección que se manda al circuito con la dirección de la memoria. No se conecta directamente con el bus de direcciones ya que esta señal no funciona exactamente igual que el bus A, en el caso en el que se esté desarrollando una transferencia por bloques, ya que tendrá que ir cambiando y aumentando su valor, en función de la dirección en la que se escriba.

En este caso, estas señales no van al exterior, por lo que simplemente se conectaran en el diseño de bloques, al bloque de memoria. Se puede observar como reset y clk no forman parte de las señales del bloque de memoria.

Una vez explicadas todas las señales externas, se va a explicar las señales internas, las cuales juegan un papel fundamental para el buen funcionamiento del diseño. De esta manera, se tienen:

- num_datos : Bus de cuatro bits que indica cómo se toman los datos, en función de DS1, DS2, A(1), LWORD y A(2). De esta manera, el tratamiento de los datos presenta la siguiente forma:

```
1111 -> D31-D00 (bytes [0-3])
1110 -> D31-D08 (bytes [0-2])
0111 -> D23-D00 (bytes [1-3])
0110 -> D23-D08 (bytes [1-2])
0011 -> D15-D00 (bytes [2-3])
1100 -> D32-D16 (bytes [0-1])
  1000 -> D32-D24 (byte [0])
  0100 -> D23-D16 (byte [1])
  0010 -> D15-D08 (byte [2])
  0001 -> D07-D00 (byte [3])
```

Para ver cómo se toman los datos del bus en función de las señales, habrá que irse al capítulo 2 en el apartado 1, donde se explica el funcionamiento del bus VME, concretamente en la tabla 2.2. Hay que destacar que, num_datos, está diseñado basándose en la configuración del bus de datos, es decir, que presenta la forma de una memoria big endian y que, por tanto, es inverso al web, ya que este presenta una forma de memoria little endian.

- addr_buf_std_v_e: Bus de treinta y dos bits que almacena el bus de direcciones para hacer con él las operaciones que se requieran, en el caso de que se trabaje con block transfer. Las operaciones a realizar son el aumento de la dirección en uno, dos o cuatro, en función del tipo de transmisión por bloques que tengamos.

- addr_buf_u_e: Bus de treinta y dos bits que almacena el bus proveniente del address, cambiándolo de std_logic_vector a unsigned, como buffer intermedio, para poder después pasar a integer y poder realizar con él las operaciones que correspondan.

- addr_buf_i_e: Señal que almacena el bus addr_buf_std_v_e, pero pasado a integer, para poder realizar las operaciones de suma y resta necesarias para poder trabajar con el bloque de memoria, aunque su valor será pasado más adelante a std_logic_vector.

- addr_buf_std_v_s: Señal de treinta y dos bits donde se almacena el bus de address que se actualiza en función de cómo se quiera trabajar en el bloque de memoria. Este valor será el que se pase a addr_buf siempre que sea necesario actualizar el valor de address.

- addr_buf_u_s: Señal de treinta y dos bits donde se almacena el valor de address actualizado, es decir, la nueva dirección donde se quieran sacar los datos para el nuevo ciclo, proveniente de addr_buf_i_e pasado a unsigned. Su función es la de buffer intermedio que se emplea para el paso de integer a std_logic.

- WRITE_not: Señal que tiene la función de almacenar el complemento de WRITE . Este se emplea en el IOBUF, para gobernar si está en modo entrada o salida. De esta manera sería '1' para entrada y '0' para salida.

- D_out_buff: Bus de dieciséis bits el cual es el encargado de almacenar el valor que proviene de D, cuando el IOBUF se encuentre en modo entrada.

- D_in_buff: Bus de dieciséis bits el cual es el encargado de mandar el valor que corresponda por D, cuando el IOBUF se encuentre en modo salida.

- web_buf: Bus de cuatro bits el cual almacena el web que va a la memoria en caso en el que se esté en modo escritura en ésta. Si estamos en modo lectura, el web sería "0000". Cabe destacar, como se ha dicho antes, que presenta una forma inversa a num_datos.

También, dentro de las señales internas tenemos la que se refiere a los diferentes estados de la máquina. Para poder emplear esta se ha creado un tipo de señal de la siguiente manera:

```
type STATE_TYPE is (S0, S1, S2, S3, S4);
signal CURRENT_STATE: STATE_TYPE;
```

Para terminar este apartado, se pasa a explicar las diferentes variables que se emplean en el proceso principal del diseño aquí realizado, que realizan funciones auxiliares:

- contador_bytes: variable integer la cual se emplea para poder saber si a la hora de la transferencia por bloques nos hemos pasado en cuanto al número de bytes se refiere. Cabe destacar que para la transferencia de treinta y dos bits, el máximo sería de doscientos cincuenta y seis bytes, y para sesenta y cuatro bytes, de dosmil cuarenta y ocho bytes.

- blk_trnf: señal que indica si se está en una transferencia de bloques o no. De esta manera, si está a '1', se está en una transferencia por bloques, y si está a '0', se está en una transferencia de ciclo único.

- n_datos_dir: bus de dos bits que nos indica el tamaño de direcciones que tendremos que tener en cuenta. Funciona de la siguiente manera:

```
00 -> A16
01 -> A24
10 -> A16
```

- user_sup: señal que nos indica si estamos en modo supervisor o modo user. Estas funcionalidades no presentan ninguna diferencia con respecto a la transmisión, por lo que no pasará nada con ellas en nuestro diseño. Sin embargo, se implementan banderas para saber en qué modo estamos, ya que, en un futuro, sería posible emplear algunas de sus funciones.

- block_single: bus de dos bits el cual nos indica el modo que se usará para mandar los datos, quedando de la siguiente manera:

```
00 -> data,single cycle
01 -> program,single cycle
10 -> block transfer 32-bits
11 -> block transfer 63-bits
```

Con data y program, pasa lo mismo que con user sup, por lo que no se tendrán en cuenta actualmente.

Una vez explicadas las señales y variables del diseño, necesarias a la hora del correcto funcionamiento del circuito e indispensables para poder entender los siguientes apartados de este capítulo, se va a pasar a explicar el grueso de lo que es el diseño final.

3.3 Explicación del circuito

En este apartado se va a explicar cómo funciona el circuito que ha sido implementado. Para una mejor comprensión, este se va a dividir en la parte combinacional del diseño, que será un apartado más reducido, y el grueso del diseño, que es la parte secuencial.

3.3.1 Parte combinacional del diseño.

Como parte combinacional del diseño, debemos situar 3 asignaciones.

La primera y la más sencilla es en la que se asigna el valor de WRITE_not. Simplemente lo que se hace aquí es complementar el write para guardarlo en otra señal. Esta señal estará conectada a la T del IOBUF, es decir, que controla cuando este se encuentra en modo entrada o salida. Debido a que cuando se escribe en el bus, es decir, los datos necesitan estar como entrada, el WRITE es igual a ‘0’, se realiza la negación para mandar un ‘1’ a la entrada T del bus, y ponerlo en ese estado. Para la escritura pasa lo mismo, solo que, al contrario.

Se tiene una segunda asignación, la cual será la encargada de controlar la salida web_out. Básicamente la funcionalidad que se recoge aquí es que, como el web_out es el que va directamente conectado al web de la memoria (véase el apartado 3.1 para obtener más información), este tiene que actuar como tal. Así, cuando se quiera escribir en la memoria, este tiene que poner a '1' los bytes que quiera escribir, y si se quiere leer de la memoria, se tendrá que quedar a '0'. Por tanto, en la asignación se tiene que, si WRITE es igual a '1', es decir, que se quiere leer de memoria, el web_out se pondrá a "0000", mientras que, si no es así, web_out cogerá el valor de de web_buf, que es un buffer intermedio que se controla dentro del circuito secuencial, y que tiene el web que se quiera, en el momento indicado. La actualización de los valores del web_buf se realiza en la parte secuencial.

Por último, se tienen 2 asignaciones que tienen funciones similares, ya que éstas se usarán para pasar el valor actualizado de la dirección de memoria, que está en integer para poder realizar cálculos sobre él, y pasarlo a un std_logic_vector de treinta y dos bits. De esta manera, tenemos una asignación que pasa de integer a unsigned de treinta y dos bits, almacenando el valor en un buffer intermedio (addr_buf_i_e), y luego otra asignación que pasa este buffer unsigned a un std_logic de treinta y un bits (addr_buf_std_v_s). No obstante, hay que aclarar que este valor que se almacena en addr_buf_std_v_s no será el valor que irá directamente a la memoria, si no que se situará en el bus de address cuando se requiera.

También se genera el IOBUF de treinta y dos bits, el cual se conecta con las entradas o salidas necesarias. En el diseño real, este será de treinta y dos bits, sin embargo, para este prototipo solo se usarán dieciséis. Para poder generarlo, es necesario emplear un generador (GEN)

Una vez que se ha explicado la parte combinacional del circuito, se pasará a explicar la parte secuencial de este.

3.2.2 Parte secuencial del diseño.

En la parte secuencial del diseño queda definida la máquina de estado que engloba el funcionamiento del esclavo VME. No obstante, antes de entrar en materia, explicaré dos puntos necesarios para poder entenderla.

En primer lugar, será necesario conocer las señales que irán en la lista de sensibilidad del diseño. Esta serían las siguientes:

```
reset, clk
```

Estas señales son necesarias aquí ya que estas serán usadas en los diferentes if-else del diseño, y por tanto, para que puedan ser comparadas es necesario que aparezcan en esta lista.

Por último, se debe saber que se realiza un reset asíncrono al inicio, en el cual, se tiene la inicialización de varias señales y variables. Se entra en el caso de que se active el reset (recordemos que el reset es activo en baja), si no se sigue con el codigo más abajo. De esta manera se tiene la inicialización del enable, que tiene que estar a '0' para que la memoria no interactúe durante el proceso, sólo cuando se quiera. El DTACK a '1', ya que este es el valor de reposo de la señal. CURRENT_STATE se sitúa en s0 ya que este es el estado inicial del proceso. num_datos, block_single, user_sup, n_datos_dir se ponen a '0' ya que he considerado que este es el estado de reposo. contador_bytes se queda a '0', ya que todavía no se ha comenzado ninguna transmisión. blk_trnf se sitúa a 0, ya que he considerado, como antes, que éste será su estado de reposo.

Una vez explicado estos dos detalles, se pasa a explicar la máquina de estado. Se compone de cinco estados, en los que se realizan las acciones necesarias para el funcionamiento del bus. De esta manera, comenzaré a explicar paso por paso el diseño.

- **Estado s0**: estado de espera de inicio.

En el s0, en primer lugar, lo que se mira si AS está a '0', es decir si se quiere comenzar una transmisión. Si no está a '0', se mantiene en s0, si no, almacenamos el bus de address en nuestros buffers, es decir, en addr_buf, en addr_buf_std_v_e y en addr_buf_u_e. Una vez se almacenan los valores, se pasa a comprobar los Address Modifiers. Los valores de n_datos_dir, user_sup, block_single y blk_trnf cambiarán en función de estos, teniendo en cuenta la tabla 3.1:

AM	n_datos_dir	user_sup	block_single	blk_trnf
0x08	10	0	11	1
0x09	10	0	00	-
0x0A	10	0	01	1
0x0B	10	1	11	1

0x0C	10	1	00	-
0x0D	10	1	01	-
0x0E	10	1	01	-
0x0F	10	1	10	1
0x29	00	0	00	-
0x2C	00	1	00	-
0x2F	-	-	-	-
0x38	01	0	11	1
0x39	01	0	00	-
0x3A	01	0	01	-
0x3B	01	0	10	1
0x3C	01	1	11	1
0x3D	01	1	00	-
0x3E	01	1	01	-
0x3F	01	1	10	-

Tabla 3.1: Variables n_datos_dir, user_sup,block_single, blk_trnf, en función del modo en el que se vaya a trabajar.

Podemos ver, además, en la tabla 3.1 los modos de funcionamiento que van a ser empleados. El modo 2F simplemente se nombra. Más adelante, en el capítulo 4, de verificación, se podrá observar cómo funcionan cada uno.

Así, una vez coincidan los Address Modifiers y se modifiquen las diferentes variables se cambiará el estado a s1, pasando así al siguiente estado. Cabe destacar que lo idóneo sería mirar primero la dirección, sin embargo, como de los Adress Modifiers depende los bits del bus de address que se va a comprobar, necesitamos procesar primero estos.

- **Estado s1**: Estado de comprobación de la dirección.

En el estado s1, en primer lugar, se comprueba si se mantiene el AS a ‘0’, por si se ha producido algún fallo y ha llegado una señal AS pero no se quiere una comunicación. De esta manera, si AS es igual a ‘1’, significa que no se quiere seguir con la comunicación, y por tanto vuelve al estado s0. Si no, se pasa a comprobar el bus de Address. De esta manera, comprobamos si la dirección que llega coincide con la que corresponde con el esclavo. En función de n_datos_dir se evaluarán más bits del address o no, pudiendo evaluar A16, A24 o A31. De esta manera si no coincide, se pasa al estado s0 y se espera que se produzca otra transmisión. Si coinciden, se pasará al

estado s2 para seguir con el proceso. Además, en este estado, se realiza en cambio de std_logic_vector a unsigned, por parte del bus de address, que se va a emplear en la transmisión por bloques.

- **Estado s2**: Estado de espera, para mandar o recibir datos.

En el estado s2, en primer lugar, lo que se realiza es comprobar si los Data Strobe han cambiado. Si no es así, se mantiene en este estado. Si es así, en primer lugar, se pasa s3. Además, si el WRITE es igual a '1', es decir, se está leyendo, se pone el enable de la memoria a '1'. En este estado también se realiza el cambio de unsigned a integer del bus de address, para, en el siguiente estado, poder realizar las sumas necesarias en cuanto a la dirección se refiere.
Por último en este estado, se analiza qué datos se quieren procesar del bus D, teniendo en cuenta los Data Strobe, A(1), A(2), LWORD y blk_trnf. Se puede observar que forma tiene en la tabla 3.2:

web	num_datos	DS	A(1)	A(2)	LWORD	blk_trnf
0001	1000	01	0	0	1	0
0100	0010	01	1	-	1	0
0010	0100	10	0	-	1	0
1000	0001	10	1	-	1	0
0011	1100	00	0	-	1	0
1100	0011	00	1	-	1	0
1111	1111	00	0	-	0	0
0111	1110	01	0	-	0	0,
1110	0111	10	0	-	0	0
0110	0110	00	1	-	0	0
0001	1000	01	0	0	1	1
0100	0010	01	1	-	1	1
0010	0100	10	0	-	1	1
1000	0001	10	1	-	1	1
0011	1100	00	0	-	1	0
1100	0011	00	1	-	1	0
1111	1111	00	0	-	0	0

Tabla 3.2: num_datos y web, en fucnion de los DS, A(1), A(2), LWORD y blk_trnf.

En la tabla 3.2 que también se va a tener en cuenta la variable que indica si estamos en transferencia de bloques o no (blk_trnf), ya que siempre que se realice una transmisión de este tipo, se volverá al estado s2 para ver si ha cambiado la dirección (que si que cambiara por que se realiza de manera interna) y los Data Strobe. Por eso también se comprueba el AS en este estado, ya que, si se está en el modo de transferencia por bloques, si el AS se pone a 1 en este estado, indicará que ha terminado la transmisión, por lo que se tendrá que volver al estado de reposo.

También se puede observar, en la tabla 3.2, que se comprueba A(2), pero que este nunca cambiara en el prototipo. Esto es debido a que, en el caso de que se quiera trabajar en el modo multiplexado (enviando datos por el bus de direcciones también), habría que comprobar el A(2). Sin embargo, este no sería el caso ahora mismo, pero queda registrado por si en un futuro se quisiera aplicar este modo.

Por último, se puede observar que el web, sería el inverso del num_dato. Esto es debido a lo comentado anteriormente de que el bus es big endian y la memoria little endian, por lo que, para que los datos se guarden de manera correcta, es necesario hacerlo así. Se pasa a explicar el estado s3.

- **Estado s3**: Estado de comprobación si se está en una transferencia de bloques o no

En este estado se empezará actualizando el buffer de dirección que va hacia el bloque de memoria. Después se comprueba si AS se pone a ‘1’. Esto se comprueba para ver, en el caso de la transferencia de bloques, si se ha terminado la transmisión. En el caso en que AS este a ‘1’, el enable de la memoria se pondrá a ‘0’ (en la transferencia de bloques, para la lectura, el enable se queda activado para poder tener el valor de la memoria en el momento exacto en el que lo queremos), se pasa al estado s4. Si AS sigue siendo ‘0’, se sigue.

Una vez realizado este proceso, pasamos a comprobar si los Data Strobes han cambiado. Si no es así, se mantiene en este estado. Si han cambiado, se pasa, en primer lugar, a comprobar si se está en modo escritura, o modo lectura. Si se está en el modo lectura, se pasará a comprobar en qué modo de transmisión se quiere trabajar. Esta comprobación se realiza tanto en escritura como en lectura, y se hace comprobando los valores de block_single. Por último, se comprueban los datos que se quieren mandar, comprobando num_datos, y en función de esto, se sitúan en el bus de datos los que correspondan, además de poner el DTACK a 0 y pasar al siguiente estado.

En el caso en el que se esté en una transferencia de bloques, tanto para lectura como para escritura, además de realizar todo lo necesario para poder leer o escribir de manera correcta en el bus, es necesario comprobar si se está dentro del número máximo de bytes transmitidos. Si ya se superó, se mantiene en este estado hasta que el AS se

ponga a ‘1’. Si no, se le sumarán los bytes que se vayan a transmitir, además de actualizar el valor de la dirección de memoria, una vez se haya procesado el actual, para poder tenerlo preparado para la siguiente transmisión.

En este estado, además, realizaremos el tratamiento de los datos, en función del num_datos. Por tanto, si WRITE está a ‘1’, mandaremos los datos de memoria que se requieran, y si WRITE está a ‘0’, se guardarán los datos en memoria. También hay que tener en cuenta que, a la hora de mandar los datos desde el esclavo, los bytes que no se escriban, se rellenaran de ceros. Esto se realiza en todos los tipos de transmisiones.

- **Estado s4**: estado final para la transmisión en ciclo único.

La función principal del estado s4, sería la de comprobar si los Data Strobe se ponen a 1. Si siguen a 0, se mantendrá en este estado. Si se ponen a 1 ambos, lo que se hará a continuación será comprobar si el AS está a 1 también. Si es así, pasamos al estado de reposo s0. Comprobamos, además, si estamos en una transferencia de bloques, y si es así, actualizaremos la dirección, y pasaremos a s2.

Debido a que la función principal de este trabajo es la de demostrar que se ha realizado el bus VME, y corroborar que todo funciona de manera correcta, no se ha indagado en el código del diseño, ya que los capítulos importantes de este serán el de verificación y demostración. No obstante, se ha dejado un último apartado dentro de este capítulo para explicar el diseño de bloques, ya que es una parte importante del diseño, y necesario para la comunicación con el microcontrolador. Además, en este diseño no se ha incluido una parte, que sería la función de los adaptadores de tensión. Para ello, se dejará un capítulo aparte, donde queda todo explicado de una manera más clara.

3.3 Explicación del diseño de bloques.

Debido a que ha sido una parte importante del diseño, se va a proceder a explicar el diseño de bloques que se ha realizado. De él se destaca el bloque de memoria, ya que todo lo demás ha sido usado en su modo estándar, por lo que no se realizara un análisis muy detallado.

A continuación, se muestra, en la figura 3.3.1 una imagen del diseño que ha sido realizado:

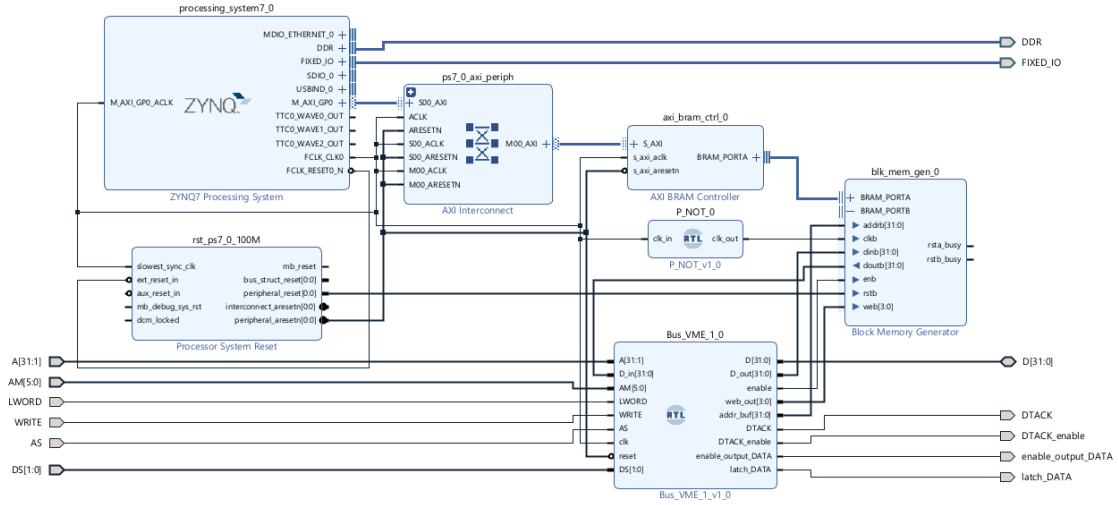


Figura 3.3: Diagrama de bloques

Se comienza explicando los bloques más pequeños que están a los laterales, en la figura 3.2. Estos hacen referencia a las diferentes señales externas del diseño. Anteriormente ya se hizo referencia a que, mediante el archivo de restricciones, estas señales se conectaran con el exterior. Además de las señales externas que utilizamos, también aparecen dos más, que serían DDR y FIXED_IO, aunque no van a ser comentadas ya que no se emplean en el diseño.

En cuanto a los bloques más grandes, comenzaré hablando del ZYNQ7 Processing System. Este bloque es el que hace referencia al microcontrolador. A este llega una señal, que sería M_AXI_GPO_ACLK, que proviene de la señal de FCLK_CLK0 que sale también del micro, y además sale la señal FCLK_RESER0_N, que el reset asíncrono, y M_AXI_GP0, que es bidireccional, y que es la encargada, mediante bloques auxiliares, de la comunicación con la parte hardware del diseño.

A continuación, hablaré del bloque AXI Interconnect. Este se usa para interconectar todos los GPIO que vayan entre la parte hardware y software, además de conectar los diferentes controladores de bloques ya diseñados (por ejemplo, la memoria). Este bloque a su vez, requiere de un sistema de reset. Para ello, se emplea el Processor System Reset, que está conectado al reset del microcontrolador, y de él salen los diferentes reset que usará el sistema.

El AXI BRAM Controller es un bloque que su utilidad es la de controlar el bloque de memoria. Este presenta dos puertos, pero en nuestro caso solo se va a emplear uno. Este está conectado vía bus bidireccional con el AXI Interconnect.

Block Memory Generator es el bloque de memoria. Cumple con los requerimientos del dispositivo de Xilinx en el que se va a implementar [2]. También presenta dos puertos, que se emplean uno para la parte hardware, y otro para la parte software. Para conocer

más acerca de este bloque, habrá que ir al apartado 2.2.1, en el cual se explica su funcionalidad.

Aparece un bloque que no ha sido mencionado hasta ahora, el cual he diseñado, que es P_NOT_v1_0. Este diseño simplemente realiza la funcionalidad de una puerta NOT. Básicamente se emplea para negar el reloj que va hacia la memoria, consiguiendo un retraso de medio ciclo de reloj. Esto tiene que ver con la escritura en memoria, para que se produzca cuando todas las señales se han estabilizado.

Por último, se tiene el bloque principal del bus VME, el cual ha sido explicado más arriba, ya que es el propio diseño que se ha realizado.

En definitiva, en este capítulo se ha hecho una explicación de que se ha realizado para que el diseño funcione de manera correcta. Más adelante, en los capítulos 4 y 6 quedará demostrado que esto es verdad, y se demostrara que funciona.

Capítulo 4. Verificación.

En este capítulo se van a mostrar las pruebas realizadas en simulación, con las cuales quedará comprobado que el diseño cumple con los requerimientos que necesita el protocolo VME. Se van a realizar simulaciones tanto de las transmisiones de ciclo único, como de la transferencia de bloques. además, habrá que diferenciar cuantos bytes se quieren trasmitir, y si están o no alineados.

Antes de comenzar este apartado, es necesario conocer cómo se han realizado las diferentes simulaciones. Para ello no se ha creado un test_bench, si no que se han forzado los valores de las señales de entrada, y se ha visto que va ocurriendo con las señales de salida. También hay que tener en cuenta una cosa, y es que, en cuando al bus de datos, una vez que se fuerza su valor, este solo responderá cuando se vuelva a forzar. Este concepto es muy importante, ya que, para poder ver que va ocurriendo en la memoria, los valores que se saquen se verán en D_in_buff y D_out_buff (recordemos que son la salida y entrada del IOBUF, y no nos fijaremos en D, ya que, ahí, solo forzaremos valores.

Otro punto fundamental que también se debe tener constancia es que, en el diseño, el bus de direcciones desprecia el bit cero. De esta manera, se pondrá un valor de dirección sin bit cero, y el diseño internamente se lo colocará. Es necesario entender esto ya que, en las simulaciones, la dirección que nos llegue será diferente a la de la memoria (se le añadirá un cero) y, por tanto, puede crear confusión. Para poder observar de manera clara que ocurre en cuanto a las direcciones habrá que fijarse en la señal addr_buff.

Por último, se va a mostrar cada transmisión con un único ejemplo. De esta manera, aquí solo se explicará una transmisión de ciclo único de escritura y lectura alineada, una de escritura y lectura desalineada, y, por último, dos de escritura y una de lectura en transferencia de bloques. Además de estas pruebas, he realizado pruebas de todas las transmisiones restantes. Para poder verlas, habrá que irse al ANEXO I, donde se refleja una captura de cada una, sin explicación, pero que demuestra que todos los modos de transmisión funcionan correctamente.

4.1 Transmisiones de ciclo único.

En este apartado se van a mostrar pruebas para todas las transmisiones de ciclo único. Para poder ver que interactúan unas con otras sin problemas, se va a tener la escritura de un tipo, y lectura de otro.

4.1.1 Escritura.

En este apartado se muestran las pruebas que comprueban el funcionamiento del esclavo en modo escritura tanto alineada como desalineada. En cuanto a la escritura alineada se verá en el caso en el que sea de un byte, y en el caso de la transferencia desalineada, se verá en el caso en el que se quieran transmitir los tres últimos bytes (byte uno – tres).

Comenzare con la escritura alineada:

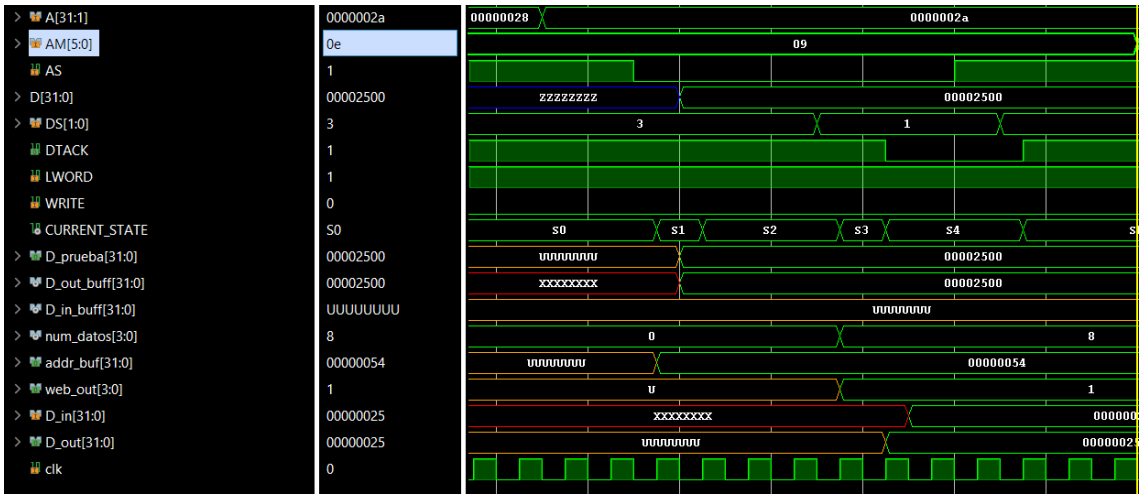


Figura 4.1: Simulación de escritura en el modo un byte, alineada.

Se puede observar en la figura 4.1 el ciclo de escritura en memoria, cuando se está en el modo de byte a byte. Concretamente se escribe en el primer byte del bloque de memoria. Como puntos importantes para tener en cuenta, hay que fijarse en las entradas, las cuales han sido forzadas según se vio en el apartado 2.1 del capítulo 2, para poder realizar esta transmisión. De esta manera, hemos situado bus las direcciones en la dirección 0x2a, que corresponde con la dirección 0x54 (se puede observar en addr_buf), ya que internamente, se le ha añadido el bit cero. Los AM se han situado en 0x09, es decir, que se mirarán los treinta y un bits de direcciones, y se está en una transmisión de ciclo único. Hay que destacar que el AM escogido ha sido uno de los tantos que se pueden elegir, y a medida que se vayan desarrollando las diferentes transmisiones, iré alternando el AM para que se pueda apreciar que todos los mencionados están implementados y funcionando correctamente. El LWORD está a ‘1’, ya que es necesario ese valor para esta transmisión. También, el WRITE está a ‘0’, lo que indica que estamos en modo escritura.

De esta manera, una vez ya configurados estos valores, se cambia el AS a ‘0’, para iniciar la transmisión, y después de 20 ns, situamos los datos en el bus D. Esperamos otros 10 ns, y situamos “01” en los DS, los cuales deben tener este valor, para poder transmitir de la forma que queremos. Pasados 20 ns, el DTACK ya habrá cambiado, lo

que indica que ya se ha procesado la información, y ponemos el AS a ‘1’. Luego de 10 ns, los DS también pasan a valer “11”.

Otro punto importante que se puede observar es el ciclo que se necesita para escribir en la memoria. Lo mencionado anteriormente sobre la dirección de memoria y el bus de direcciones se cumple, ya que, si se le añade un ‘0’ a la derecha de 0x2a, éste pasa a ser 0x54. Además, se puede observar cómo el 0x25 (byte que se quiere escribir en memoria), pasa a D_out, señal de entrada a la memoria, en la posición del primer byte, no como en el bus de datos, que está en el segundo. Esto es así, ya que lo marca el bus. El dato es acompañado por el web_out.

Se puede observar como el byte ‘0’ que se quiere escribir en memoria se sitúa en la posición del byte tres del bus. Esto es así ya que para la transmisión byte a byte, como se vio en la tabla 2.3, solo se emplea la parte alta del bus, al igual que para la transmisión de dos bytes. En el caso de la transmisión de 4 bytes, y en la desalineada, que veremos a continuación, se emplean todos los bytes. También cabe apreciar como la memoria interna utilizada, como ya se ha comentado con anterioridad, es Little endian, de ahí que se guarde en la parte derecha.

Una vez explicada la parte de escritura alineada, en la cual solo se ha explicado de un byte, pero que es en todos los casos igual, procedo a explicar la desalineada, en la cual, el proceso es el mismo que en la escritura alineada, pero en la que se ve de manera más clara y detallada lo que se viene comentando desde el principio sobre la diferencia entre las memorias:

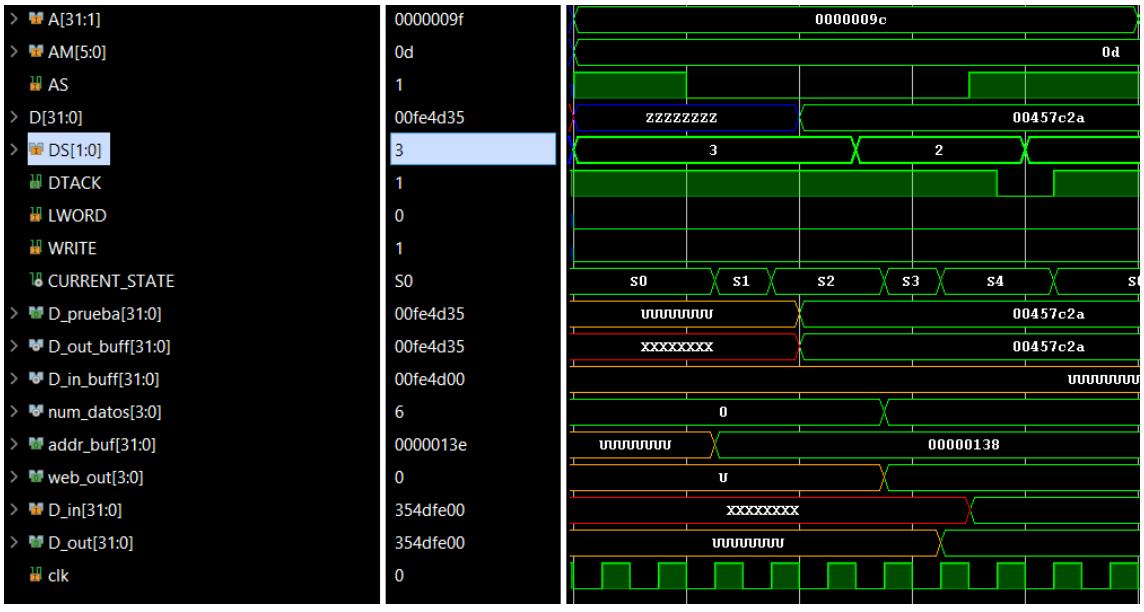


Figura 4.2: Simulación de escritura en el modo tres-un bytes (desalineado)

En la figura 4.2, se tiene la transmisión de ciclo unico de escritura desalineada. Se puede observar que el AM ha cambiado, sin embargo, simplemente cambian lo comentado anteriormente sobre user/supervisor y data/program, por lo que, con respecto a este diseño no cambian. Ademas, se puede observar como el valor de los DS es de “10”, el de LWORD es de ‘0’, y el de A(1) es igual a cero. Esto es asi ya que, como se vio en la tabla 2.2, son los valores requeridos para este tipo de transmision.

Una vez explicado esto, se pasa a ver lo que se ha querido reflejar en esta transmisión. Si se observa D y D_out (Para observar D_out habrá que fijarse en la segunda columna al lado de las señales), podemos observar como los valores se invierten. Esto es debido a que el bus es big endian y la memoria interna es little endian. Por tanto, con esta simulación, queda demostrado que esta funcionalidad se aplica al diseño.

4.1.2 Lectura.

En este apartado se van a ver dos tipos de lectura. Una primera lectura alineada de dos bytes, y una segunda lectura desalineada del primer y segundo byte. Para poder observar que se ha leido bien de la memoria, se van a mostrar en cada caso varias lecturas a la vez.

A la hora de poder ver que se lee bien de la memoria, es necesario escribir primero en ella. De esta manera, en ambos casos se explicará primero la escritura, y una vez esto esté claro, se explicará como se realiza la lectura, ya que la memoria, al inicio de las simulaciones está vacia, y por tanto hay que llenarla.

Comienzo explicando quá se ha escrito antes de realizar la lectura de dos bytes, mostrando la figura 4.3:

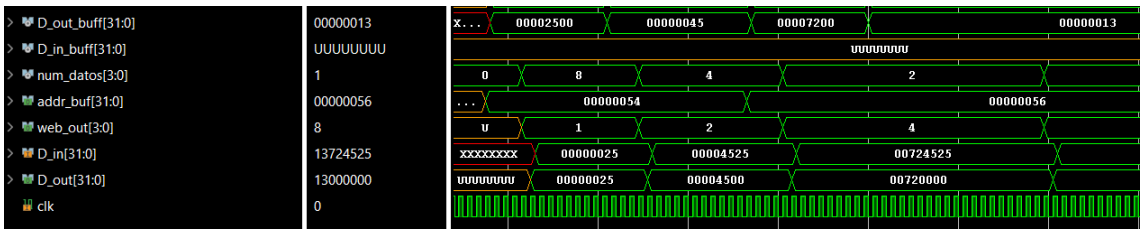


Figura 4.3: Que contiene la memoria para la lectura de dos bytes (alineada)

A la hora de escribir estos datos en memoria, se ha realizado la transmisión de un byte. De esta manera, se puede observar en D_in, en la columna siguiente el valor que tendra en la direccion 0x54 – 0x57, que será:

0x57 ……. 0x54
13 72 45 25

La figura 4.3 tambien nos demuestra como todos los tipos de transmisiones de escritura de un byte funcionan a la perfección. Con esto me refiero a que, independientemente del byte que se quiera escribir dentro del bloque de memoria de cuatro bytes, este se escribe.

Una vez explicado esto, paso a mostrar la transmision de ciclo unico de lectura de dos bytes, mostrado en la fugura 4.4:

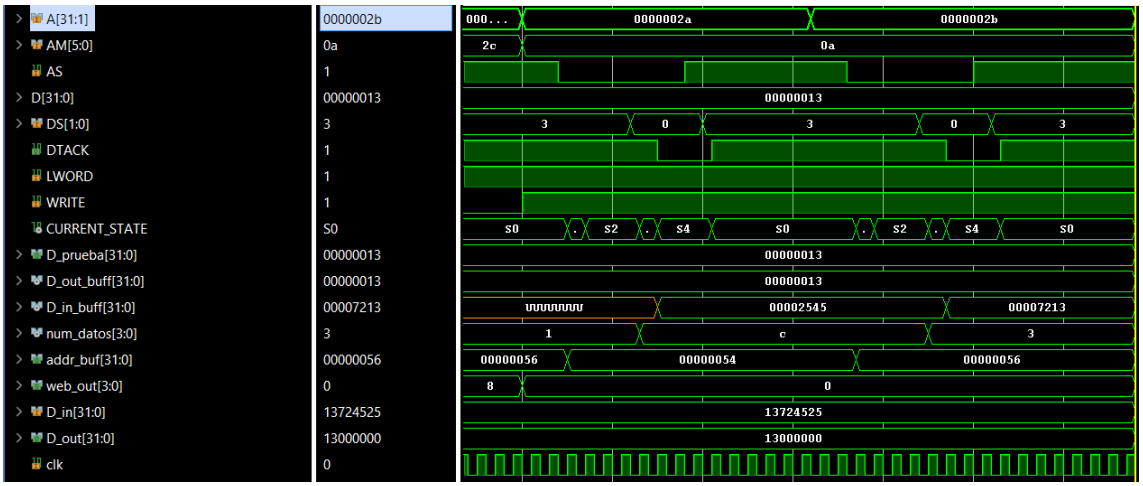


Figura 4.4: Simulación de lectura en el modo dos bytes.

En la figura 4.4, se pueden observar dos ciclos seguidos de escritura en dos bytes. Como se puede ver, no se trata de una transferencia de bloques, ya que el AS, entre ambos ciclos, se pone a ‘1’.

Este caso es muy parecido a las transmisiones de lectura, ya que, las señales de control presentan el mismo comportamiento, por lo que nos centraremos en aquellas que cambian. Se pasa a observar la señal WRITE. Como vimos en el apartado 2.1 del capitulo 2, esta señal, en el caso de transmisión de ciclo único de lectura, debe de estar a ‘1’, y si puede observar qu este es el caso. Además, los DS se ponen a “00”. Esto es asi ya que, como se vió también en el apartado referido anteriormente, en el caso en el que se quieran transmitir dos byte, estos tienen que presentar este valor, aunque, a la hora de cambiar, funcionen de la misma manera. Se puede observar también como se ha escogido un valor diferente en los AM. Esto se debe a lo comentado sobre que se quiere probar que transmite en todos los AM para los que se ha diseñado.

Se puede ver en la figura 4.4, como para poder elegir entre los bytes de la parte alta, o baja del bloque de memoria de cuatro bytes, se emplea el A(1). Asi, si este es igual a uno, se escoge la parte alta, y si es igual a cero, la parte baja.

Para terminar con el analisis de esta transmisión, voy a pasar a explicar las señales de memoria. Con respecto a la otra transmisión, no cambia mucho, sin embargo, hay una

señal en la que hay que fijarse. Se puede observa como web_out esta a cero todo el rato. Esto se debe a lo comentado en el apartado 3.3.1, del capitulo 3, ya que, mediante una instrucción concurrente, si el WRITE esta a '1', este se pone a '0'. Ademas, se puede ver reflejado tambien que, en el caso de la lectura, la inversion de los valores es de little endian a big endian, no como en el caso de escritura, por lo que, en el diseño, se ha tenido que realizar de manera diferente.

Una vez explicada la trnamisión de lectura alineada, paso a explicar la lectura desalineada, empezando por explicar que es lo que se ha escrito en memoria:

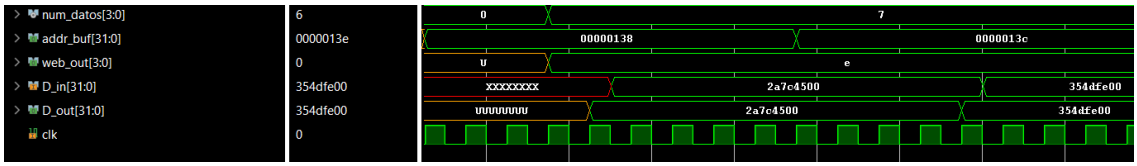


Figura 4.5: Que contiene la memoria para la lectura de dos-un byte (desalineada)

En la figura 4.5 se puede observar lo escrito en la memoria antes de leer en desalineado el byte uno y dos. Se puede analizar que son dos transmisiones, trambien desalienadas, de los bytes tres, dos y uno. Así, la forma que presentaría la memoria, una vez que se han escrito estos bloques, ya que en este caso se escribe en dos bloques, es la siguiente:

0x13b ……. 0x138 --- 0x13f ……. 0x13c
2a 7c 45 00 35 4d fe 00

Una vez conocido que se ha escrito en la memoria, paso a explicar como se ha leido esta, en este tipo de transmisión, donde se va a poder ver, al igual que antes, dos transmisiones, y donde analizaremos aquellas señales que cambien:

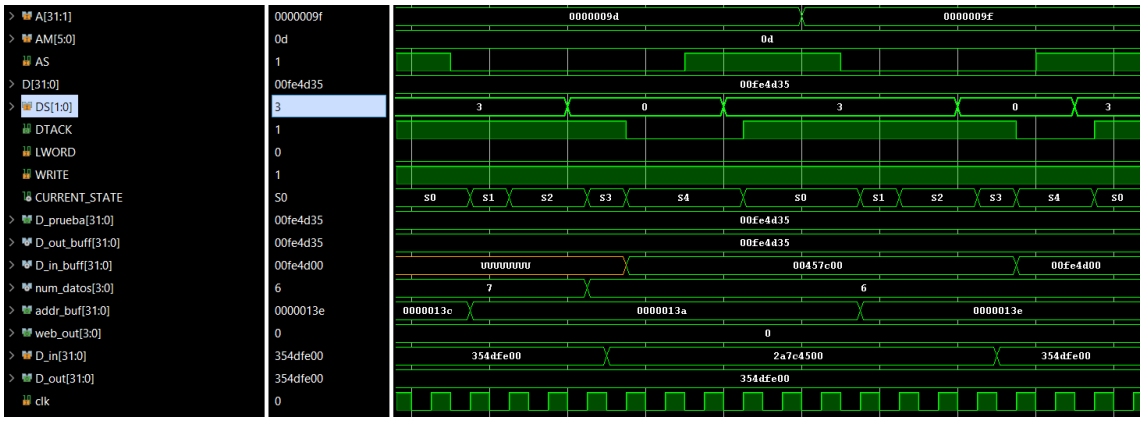


Figura 4.6: Simulación de lectura en el modo dos - un byte (desalineado)

Al igual que en la figura 4.4, en la figura 4.6 se pueden observar varios cambios. Al igual que se comento antes, WRITE esta a '1'. En ambas transmisiones, el AM esta a 0x0d,

por lo que ha cambiado con respecto a las trnamisiones de ciclo único mostradas con anterioridad, demostrando una vez mas el buen funcionamiento del esclavo, para aquellos AM definidos. Ademas de esto, habra que fijarse en LWORD, ya que, al ser una transmisión desalineada, este tiene que estar a '1', además del A(1), que en este tipo de tranmisiones, siempre estara a uno.

En cuanto a la memoria, no cambia nada con respecto a lo comentado con anterioridad, simplemente los valores que toma.

En este apartado he explicado cuatro ejemplos, dos de lectura, y dos de escritura, de la transmisión de ciclo único. Es obvio que quedan muchos más ejemplos por demostrar, sin embargo, a la hora de ver que ocurre en la transmisián, serían prácticamente iguales, simplemente cambiarían las señales de transmisión.

No obstante, en el Anexo I, quedan recogidas imágenes de simulaciones de todos los casos, para que quede bien reflejado que se han comprobado todos ellos, y quede también verificado que funcionan correctamente. Una vez aclarado esto último, se pasa a verificar las transmisiones por bloques.

4.2 Transmisiones por bloques.

En este apartado se va a verificar que las transferencias por bloques funcionan de manera correcta. En este tipo de transmisión, no implementa las desalineadas, por lo que solo se verá un caso de cada una de ellas, dos de escritura, de uno y dos bytes, y un caso de lectura de cuatro bytes. En todos los casos se van a transmitir cuatro bloques. Como se comento en el apartado 2.1 del capitulo 2, la transmisión por bloques de sesenta y cuatro bits no será implementada, ya que no se va a emplear el multiplexado.

Una vez hecha esta breve introduccion, comienzo explicando ambos casos de escritura.

4.2.1 Escritura.

Se ve a comenzar explicando la transferencia por bloques de un byte, en modo escritura:

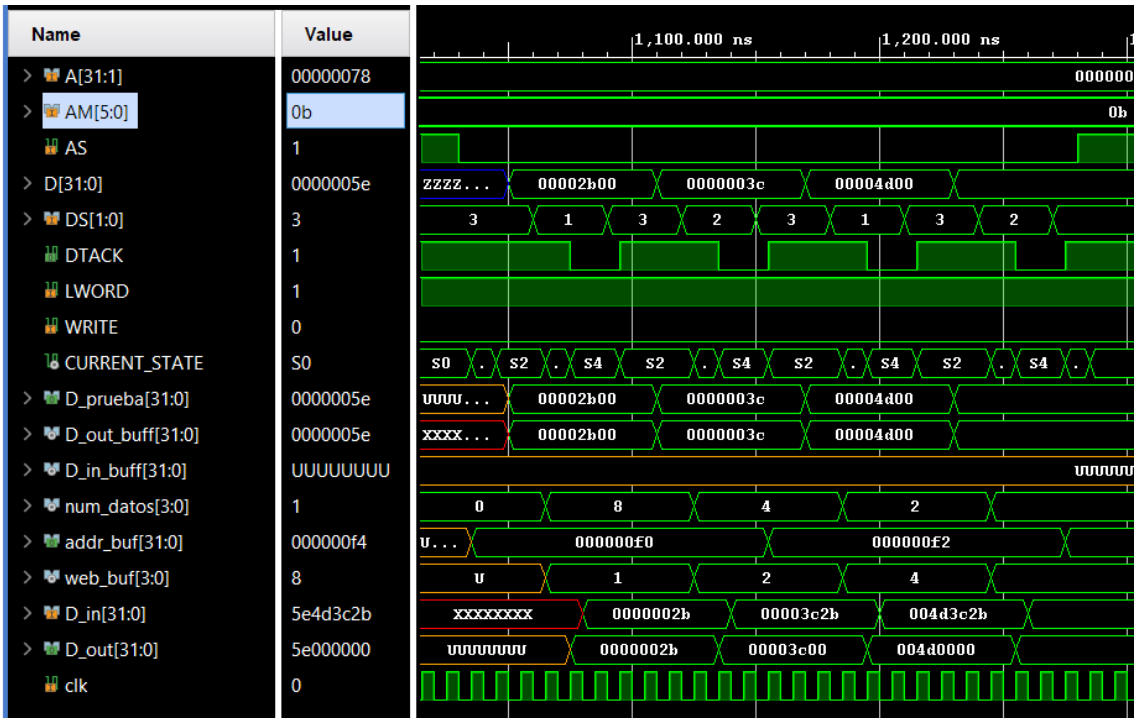


Figura 4.7: Simulación de escritura en el modo un byte, en una transmisión por bloque.

En la figura 4.7 se puede observar la transferencia de bloques de un byte, concretamente cuatro transferencia, como comente con anterioridad. En este caso, el AM es 0x0b, el cual corresponde con este tipo de transferencia, y por tanto, es el que marca si es de ciclo único o por bloques.

Un punto a tener en cuenta muy importante son los DS. Estos tienen que ir cambiando automaticamente en funcion del byte que se vaya a escribir. De esto se va a encargar el maestro. Sin embargo, el esclavo se tiene que encargar de ir cambiando el A(1), cuando se cambie de la parte baja, a la parte alta del bloque de memoria. Para poder observar esto, habrá que fijarse en el addr_buff, el cual va cambiando internamente, de dos en dos (se podría cambiar de uno en uno, pero he querido hacerlo asi para reflejar mejor la funcionalidad del bus), y, de esta manera, con el elegimos que parte del bloque de memoria se quiere escribir, y los DS el byte dentro de este.

Se puede observar también como funcionan los diferentes estados. En el caso de la transferencia de bloques, una vez que los DS pasan a su estado de reposo, se pasa a s2, donde se esperara a ver donde se va a escribir el siguiente valor en función de las diferentes señales. Es en este estado donde se compueba también si el AS a pasado a valer ‘1’, lo que indicaria que ya se ha terminado la transmisión.

Por último, la memoria funciona de manera similar a el caso de la transferencia de bloques. El unico cambio es addr_buf, que ya ha sido comentado, el cuial, internamente tiene que ir aumentando su valor.

Explicada la escritura de un byte, paso a explicar la transferencia de bloques de dos bytes, la cual sera muy similar a la anterior, salvo por el funcionamiento de alguna señal:

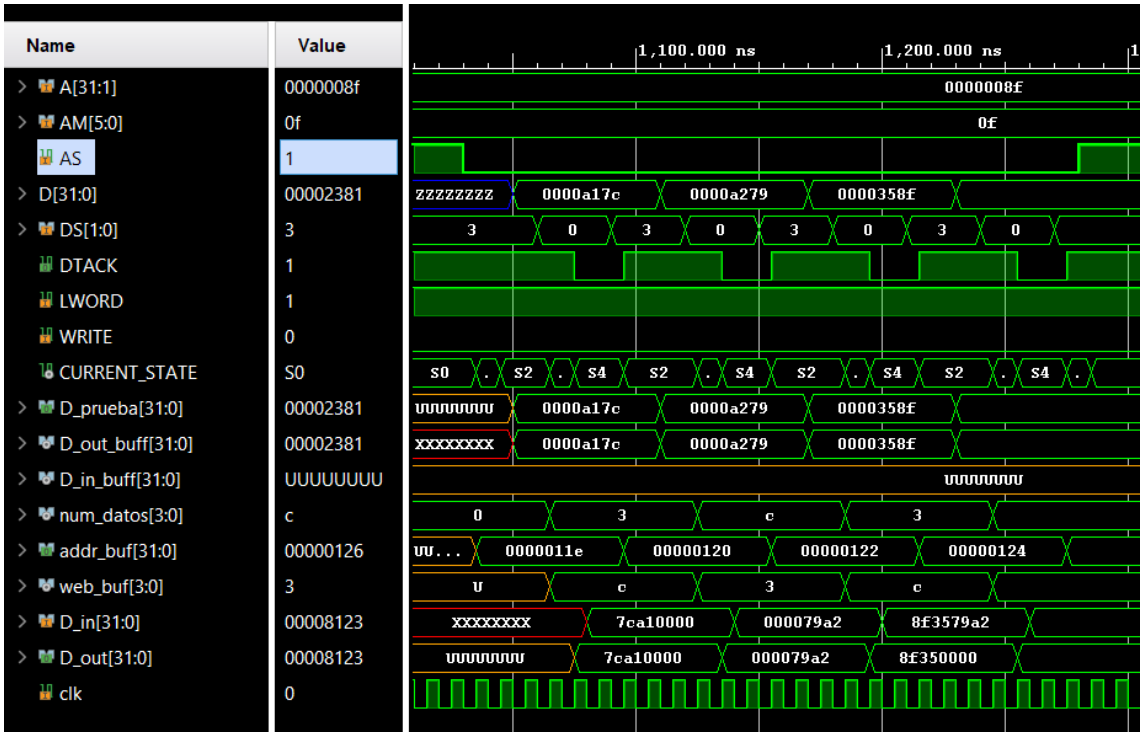


Figura 4.8: Simulación de escritura en el modo dos byte, en una transmisión por bloque.

En la figura 4.8 se puede observar una trnaferencia por bloques de dos bytes. En cuanto a las señales de control, el funcionamiento es el mismo, por lo que, habrá que fijarse en los DS. En este caso, no cambia entre uno y dos, ya que, en este caso, escribiremos sobre ambos bytes, y por tanto, no hay que elegir entre uno y otro.

Ademas se puede observar como addr_buff cambia de una manera mas rapida, esto es asi ya que ahora no se podran realizar dos transmisiones por cada valor de A(1), solo una, por lo que este tiene que aumentar su valor en cada ciclo.

La gran diferencia entre ambas transmisiones es que, con la transferencia de dos bytes, se lee mucho mas rapido de la memoria, sin embargo, en aquellos casos en los que se necesiten tomar datos para diferentes pruebas, y estos requieran de un trato byte a byte, la transferencia de un byte sera mas util.

Las transferencias por bloques permiten recopilar datos de manera mas rapida de la memoria del esclavo. Mas adelante en el capitulo 6, cuando se vea como funciona este esclavo con el maestro, solo se vera la lectura, ya que el maestro no implementa la transferencia por bloques en modo escritura.Tambie existe la transferencia de cuatro bytes, sin embargo, esta la vamos a analizar en el caso de una transmision de lectura en el siguiente apartado.

4.2.2 Lectura.

En este apartado, se va a verificar que la función lectura en la transferencia por bloques funciona, probando la de cuatro bytes. Para ello, al igual en la transferencia de ciclo único, vamos a ver que se ha escrito en la memoria, para observar que los datos se sacan de manera correcta.

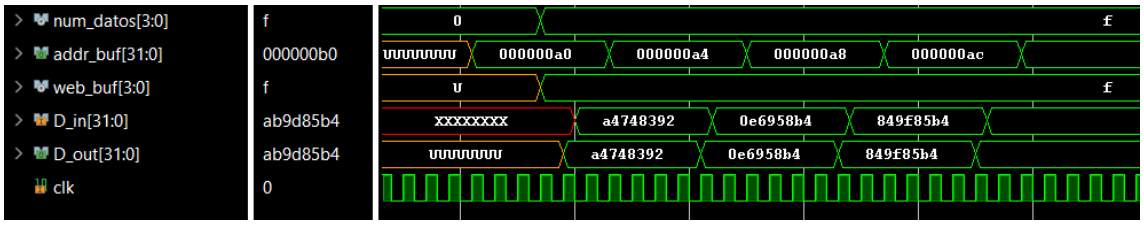


Figura 4.9: Que contiene la memoria para la lectura de cuatro bytes, en transmisión por bloques

En la figura 4.9 se puede observar que se ha escrito en la memoria. Ademas se puede ver que se ha escrito en transmisiones de cuatro bytes. De esta manera, la memoria tendria los siguientes valores:

0xa3 ……. 0xa0 – 0xa7 ……. 0xa4 – 0xab …….0xab – 0xaf ……. 0xac
a4 74 83 92 0e 69 58 b4 84 9f 85 b4 ab 9d 85 b4

Una vez que se sabe que valores va a tener la memoria, se pasa a analizar la transmisión:

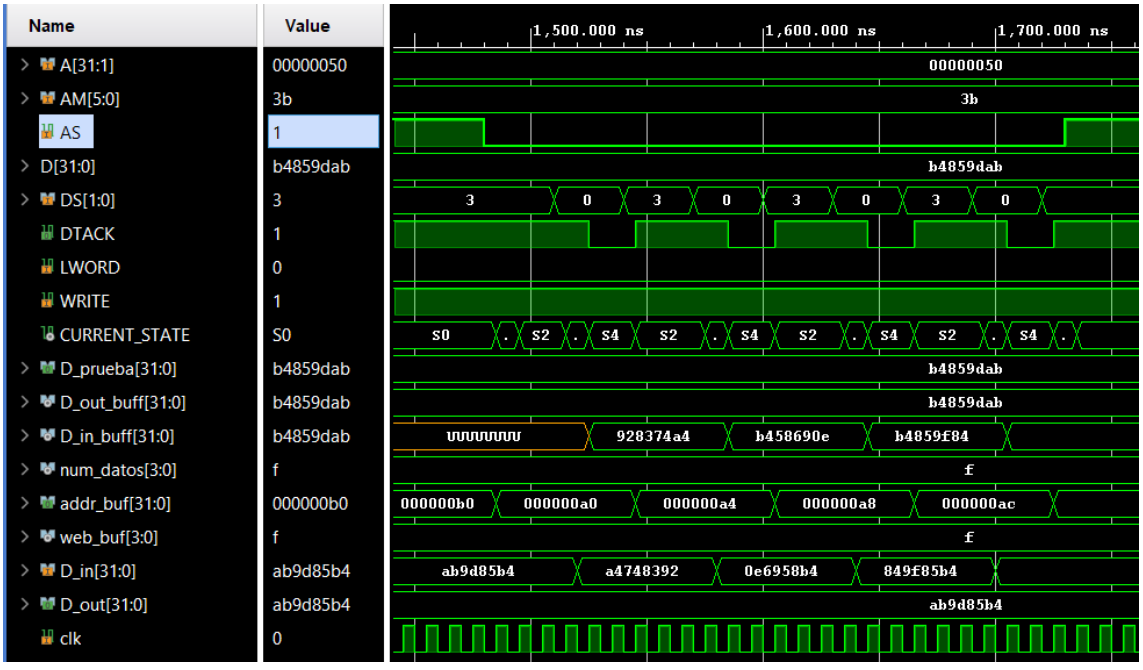


Figura 4.10: Simulación de lectura en el modo cuatro bytes, en una transmisión por bloque.

En la figura 4.10 se ve una transmision por bloques de cuatro bytes, en modo lectura, y transmitiendo cuatro ciclos. Se puede observar que las señales de control actuan igual que en las anteriores, solo cambia el funcionamiento de addr_buf, ya que, en este caso, A(1) tiene que ser siempre cero, por lo que va aumentando de cuatro en cuatro la direccion de memoria. Los DS actuan igual en la transmision por bloques de dos bytes.

En cuanto a la memoria, web_out estara a cero, ya que WRITE es igual a '1'. Se tomaran todos los bytes del bloque de memoria correspondiente, y se invertirán y verterán sobre D_in_buff, como se puede observar.

En conclusión, se ha verificado en este capítulo que el diseño funciona correctamente, demostrandolo con varias simulaciones de ejemplo, donde se han analizado las diferentes señales, y se ha visto que todo se cumple correctamente. Al igual que al terminar el apartado 4.1, para la transferencia de bloques quedan definidos todos los casos en el anexo I, donde, mediante imágenes se puede ver que todos los casos funcionan de manera correcta.

Una vez verificado el funcionamiento del diseño, llega el momento de probarlo. Para ello, en el caso en el que se quiera probar con un maestro real, es necesario hacer uso de unos adaptadores de tensión. De esta manera, en el siguiente capitulo, se va a definir que adaptadores de tensión se van a emplear, como se ha diseñado la PCB necesaria para su uso, y una vez quede definido esto, se pasar a demostrar que el diseño funciona, aunque este se probara con un maestro realizado con la misma placa de desarrollo, por lo que los adaptadores de tensión solo se emplearan para demostrar que funciona todo correctamente.

Capítulo 5. Adaptador de tensión.

En este capítulo voy a explicar el uso del adaptador de tensión y cómo funciona, ya que este es indispensable para poder emplear el diseño en el uso final que este tendrá y, por tanto, es indispensable

5.1 ¿Por qué es necesario su uso?

Debido a que este Trabajo de Fin de Grado ha sido realizado para una empresa, la finalidad del prototipo no se queda simplemente en el protocolo VME, sino que hay más.

Este bus VME será parte de una tarjeta la cual se encargará de obtener datos de unos sensores. Sus señales serán adaptadas mediante unos microchips, los cuales las pasarán por SPI a la parte del microcontrolador en C de un SoC. En la parte FPGA del mismo se implementa el esclavo VME para poder comunicarse mediante este protocolo con otras tarjetas.

Según el standard VME, tanto el maestro como el esclavo real trabajan con niveles de tensión a 5V. Las señales que salen de la placa de desarrollo actual van a dos tensiones, por una parte, aquellas señales se salgan de los FMC van a 2,5V y las que salga de los PMODs van a 3,3V. Por tanto, es necesario emplear un adaptador que pueda cambiar estas dos tensiones a 5V. Dicho de otra manera, un esclavo sobre FPGA que no incluya estos adaptadores no se podría comunicar con ningún dispositivo real, siendo esta la finalidad del proyecto de la empresa donde se engloba este Trabajo Fin de Grado.

Actualmente, debido a que un compañero ha diseñado un maestro del protocolo VME, con la misma placa de desarrollo, y por tanto, manejando las mismas tensiones, no sería necesario su uso para unas primeras pruebas. Sin embargo, como la funcionalidad real si lo requiere, estas pruebas han sido realizadas incluyendo los adaptadores, solo que se configura la tensión de entrada y salida de estos chips a la original de la FPGA en lugar de 5V por seguridad, demostrando que son viables, aunque en este caso no aporten funcionalidad. No obstante, esto último quedará mejor recogido en el capítulo siguiente, donde se demostrará que el prototipo funciona de manera correcta.

Una vez entendida la funcionalidad de estos adaptadores de tensión, voy a pasar a explicar cuál ha sido el elegido y por qué, además de mostrar el diseño de la PCB que se ha implementado.

5.2 Adaptador de tensión empleado.

A la hora de elegir el adaptador de tensión, se debía tener en cuenta varios factores. El primero y fundamenta, que tuviese una señal que gobernase cuando dejaba pasar la

señal al otro lado o no, con lo que se puede controlar que señales hablan con el maestro y cuáles no. Y otra función importante era que fuese bidireccional.

En una primera instancia se optó por emplear un adaptador de tensión que fue proporcionado por la propia empresa. Sin embargo, este no podía ser usado ya que no aportaba todas las funcionalidades que se requerían, además de ocasionar graves problemas, y hasta llegar a quemar las placas de desarrollo que se estaban empleando.

De esta manera, una vez descartado este, y tras una profunda investigación, se optó por elegir el chip SN74LXCH8T245. Este es un adaptador de tensión de la marcha Texas Instrument, el cual permite adaptar 8 bits a la vez. Con este chip se tienen las dos funcionalidades buscadas.

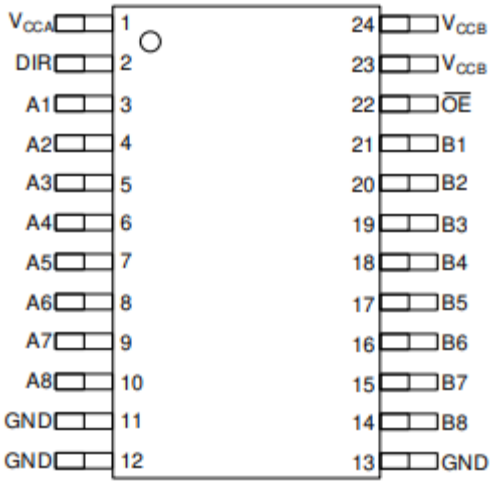


Figura 5.1: Diagrama de entradas y salidas del chip de adaptación de tensión.

En la figura 5.1 podemos observar las entradas y salidas del chip, y las cuales paso a explicar ahora:

- Vcca: alimentación del chip. Esta entrada, aparte de ser la que alimenta al chip, será la señal de referencia para las entradas (Ax), y, además, será la tensión a la que tendrá que ponerse las señales de control en el caso en el que se quiera un uno en alguna de ellas o en ambas.

- Vccb: Tensión de referencia para la salida. Esta tensión será la que se quiere a la salida, y por tanto, será la tensión a la que se tienen que poner aquellas entrada las cuales estén a uno.

- DIR: Señal de dirección. Esta entrada indicara en qué dirección se quiere transmitir por el bus, por tanto, es la señal que le da la funcionalidad de bidireccional necesaria para el funcionamiento del bus VME. De esta manera, si DIR es igual a uno, el flujo ira de A a B, y viceversa.

- OE: entrada de habilitación. Señal activa en baja que indicara si se quiere dejar pasar las señales o no. De esta manera, si OE es igual a uno, no pasaran estas señales, y si OE es igual a cero, sí que pasaran.

- Ax: Entrada/salida. Estas señales representan las entradas o salidas por el puerto A, habiendo ocho en total.

- Bx: Entrada/salida. Estas señales representan las entradas o salidas por el puerto B, habiendo ocho en total.

- GND: Toma de tierra. Todas están conectadas entre sí.

Una vez explicadas las señales, doy paso a explicar cómo funciona este chip dentro del diseño.

Para empezar, voy a comentar cuantos chips son necesarios para hacer funcionar el prototipo. Son necesarios nueve chips, ya que, se requiere uno solo para el DTACK. Esto es así, ya que el DTACK va a ser controlado desde la parte del esclavo, y los demás van a ser controlados por el maestro. Para el DTACK, como es una señal unidireccional, simplemente se pondrá la señal DIR a uno todo el tiempo, sin ningún cambio. Para la señal OE, ha sido necesario crear una señal interna dentro del diseño, que se llamara DTACK_enable, y que controlara cuando tiene que dejar pasar la señal y cuando no. Esto se controla en el estado s1, donde se comprueba la dirección, ya que, si el maestro manda la dirección correspondiente al esclavo, este habilitara el DTACK para que se puedan comunicar.

En cuando a las demás señales, exceptuando el bus bidireccional D, como sin unidireccionales, habrá que realizar lo mismo que con el DTACK, y en cuando a OE, siempre se quedara a cero, ya que estas siempre van a estar comunicando, y por tanto, siempre va a tener que dejar pasar la información.

Por último, para la señal de bus D es más compleja la situación. Para empezar, al ser un bus bidireccional, DRI tendrá que ir cambiando en función de la dirección en la que se quieran mandar los datos. De esta manera, en una primera instancia se pensó en crea una señal que se encargase de eso, sin embargo, finalmente lo que se hizo fue conectar esta señal con la señal del WRITE, ya que este cambia en función de si se lee o se escribe, lo que indica a su vez en qué dirección se quieren mandar los datos. Cabe destacar que, en función de cómo se conecten los chips, habrá que negar o no la señal. En mi caso, debido a como estaban conectados los chips, no se ha tenido que hacer.

En segundo lugar, para la señal OE, en una primera instancia se pensó en hacer diferente si se estaba escribiendo o leyendo. Esto provoco problemas a la hora de la coordinación entre el maestro y el esclavo, por lo que, finalmente, presenta una funcionalidad igual que la de DTACK_enable. No obstante, se ha querido realizar en dos señales diferentes para separar ambas y que así permita ver de una mejor manera que ocurre internamente.

Una vez explicadas las funcionalidades, voy a pasar a mostrar una imagen de una simulación, en la cual se muestra, y se verifica que funcionan de manera correcta.

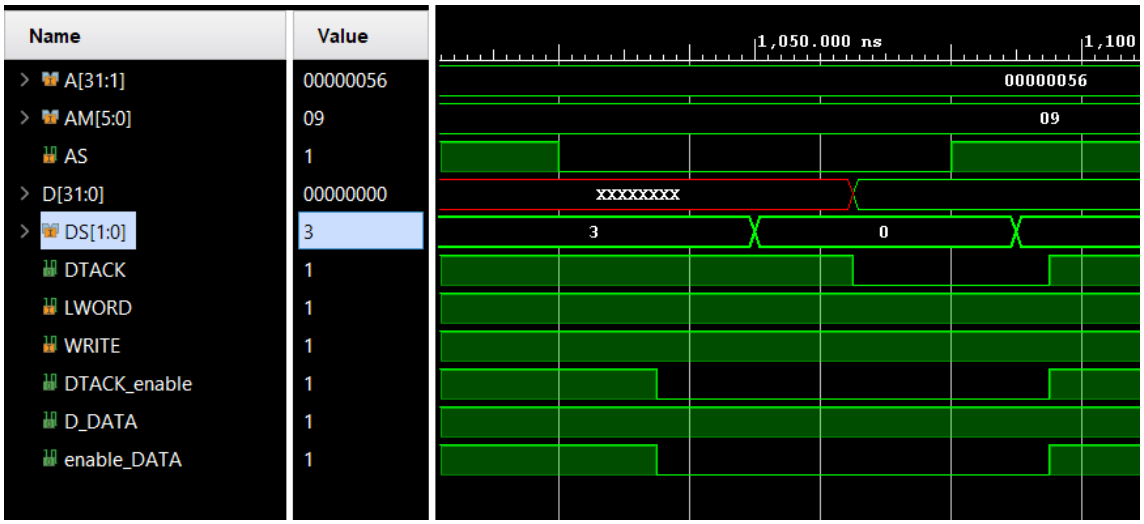


Figura 5.2: Simulación del funcionamiento de las señales del control del chip de adaptación de tensión.

En la figura 5.2 se puede observar cómo funcionarían estas señales que controlan los adaptadores. D_DATA sería la señal que indica la dirección en la que se quieren mandar los datos, que, como ya dijimos, está directamente conectada con WRITE.

Enable_DATA se corresponde con el OE de los datos, es decir, es la señal que indica cuando tiene que dejar pasar los datos y cuando no. Como se comentó con anterioridad, presenta la misma forma que DTACK_enable, que es la señal que indica cuando tiene que dejar pasar la señal de DTACK.

De esta manera, y con estas tres señales, se manejan aquellos adaptadores que necesiten un control, sirviendo, además, de control del propio bus, ya que, de esta manera, los esclavos con los que no se quiera comunicar el maestro no tendrán acceso a estas líneas, y por tanto, no podrá haber errores.

Una vez explicado el funcionamiento del chip, y como ha sido adaptado el diseño a él, es necesario entender la PCB que se diseñó para este caso, por lo que el siguiente apartado va dirigido a esa función.

5.3 Diseño PCB del adaptador de tensión.

Para este diseño PCB, en primer lugar, se tuvo que elegir entre las varias opciones que hay de programas de diseño, aunque realmente fue una decisión sencilla, ya que la empresa trabaja con Altium e Eagle. Entre estos dos se optó por Altium.

Altium es un programa de diseño PCB premium el cual facilito mucho el diseño. Este incluye ya toda la normativa relacionada con el diseño PCB, por lo que no hubo que preocuparse de eso, además de que tiene una interfaz muy sencilla que facilita mucho el diseño.

De esta manera, el diseño tiene dos componentes:

- El chip adaptador de tensión, empleando uno en cada placa.
- Los conectores externos, empleando cuatro en cada placa.

En primer lugar, para analizar el diseño, es necesario observar el circuito esquemático:

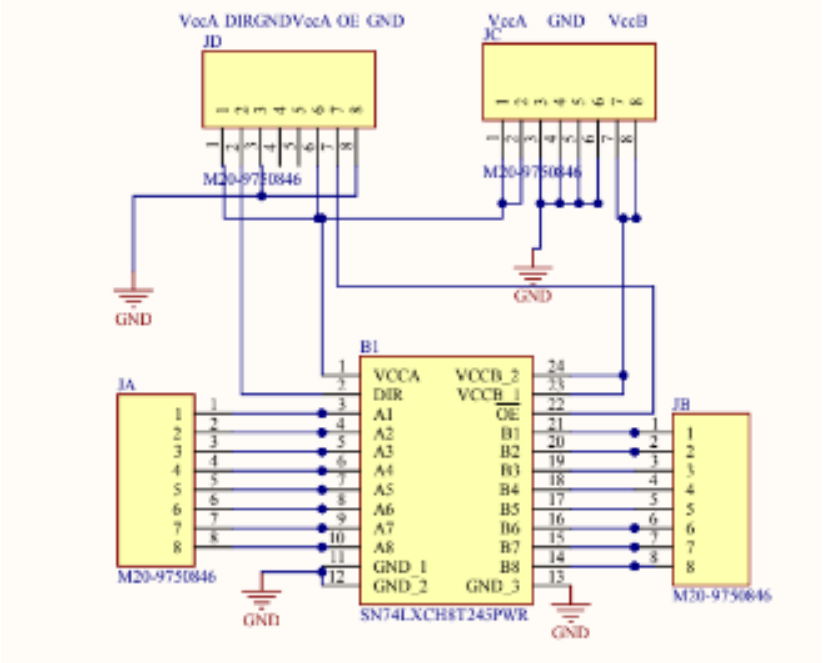


Figura 5.3: Circuito esquemático de la PCB del adaptador de tensión.

En la figura 5.3 se pueden observar las conexiones entre el chip y los conectores, la cual es directa, por lo que no hay una gran perdida. Los pines de GND están directamente referidos a tierra y no conectados entre sí ya que se va a incluir un plano de masa, que es básicamente la unión de todas las tierras mediante los huecos libres de la placa. Una vez visto el esquemático, paso a mostrar que forma tendrán los adaptadores en la realidad:

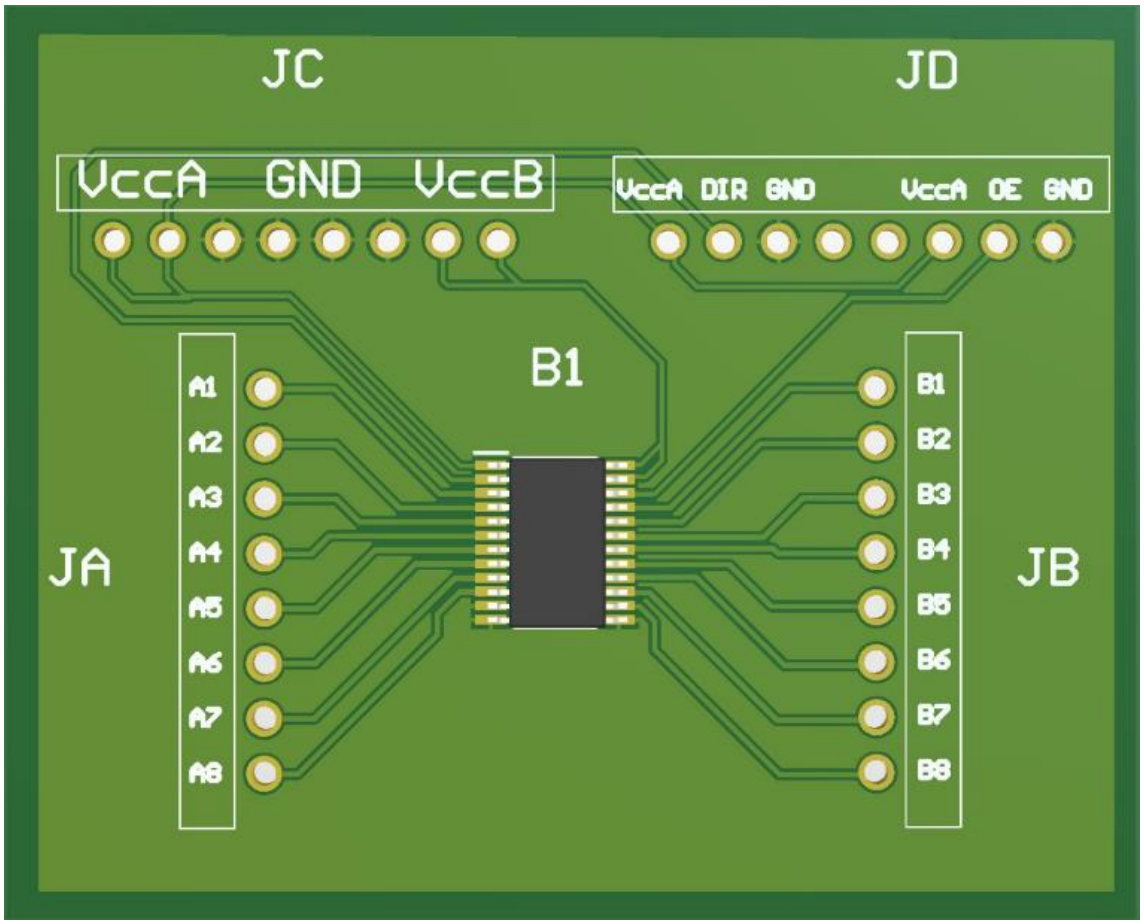


Figura 5.4: Diseño PCB del chip de adaptación de tensión.

En la figura 5.4 se puede observar que forma tendrán finalmente estos adaptadores. Los agujeros en fila de ocho representan los conectores machos de 8x1, y el chip central, el adaptador de tensión. El recuadro de color verde más claro representa el plano de masas mencionado con anterioridad.

Cabe destacar que este no ha sido el diseño definitivo empleado en la empresa, pero si el diseño con el que se han realizado las pruebas. Esto se ha debido a que, pese que me permitió hacer las demostraciones correctamente, y demostrar así que funcionan, tras un tiempo alimentado, estos se quemaban, por lo que actualmente se esta evaluando como solventar este problema.

Una vez explicado cómo funciona el adaptador de tensión, y por qué se emplea, voy a dar paso al último capítulo de este proyecto, que se corresponde con el capítulo de demostración. En él se podrá ver que el prototipo funciona correctamente enfrentándolo a otro prototipo de un maestro VME.

Capítulo 6. Demostración.

En este último capítulo se va a demostrar que el diseño funciona de manera correcta, probándolo con un maestro, el cual ha sido diseñado por un compañero.

Antes de empezar cabe destacar una cosa. A la hora de probar los chips de adaptación de tensión, estos se quemaban cuando pasaba un tiempo de funcionamiento. Tras una investigación se concluyó que podía ser que, a la hora de diseñar la PCB, era necesario incluir 2 condensadores en las alimentaciones. No obstante, esto no ha sido un problema para realizar algunas de las demostraciones, y por tanto, se ha podido demostrar que todo funciona correctamente, si estos chips funcionan. De esta manera, la solución a este problema simplemente seria incluir en el diseño de los adaptadores dos condensadores en paralelo en las entradas de Vcca y Vccb.

El prototipo que se va a implementar solo presenta 16 bit de datos. Por ello solo se van a ver esas transmisiones, salvo una excepción, en el caso de la lectura de cuatro bytes, ya que, pese que al maestro no le llegaran todos los bytes, en el debugger se podrá ver claramente que si se envían los 4 bytes. Las transmisiones desalineadas, en este caso, quedan descartadas.

Una vez aclarado esto, paso a explicar que se va a poder observar cómo demostración del funcionamiento del protocolo de comunicación. En el caso de la escritura en memoria, se va a ver, mediante el debbuger que contiene Vivado, que está ocurriendo dentro del esclavo con sus diferentes señales. Para ello ha sido necesario incluir aquellas señales que se quieran mostrar en este, además de ajustar el trigger para que, cuando se produjese un cambio en los DS, este capturara lo que ocurre antes y después dentro del diseño. Además, para este caso, vamos a poder observar también que los datos que se mandan se guardan en el bloque de memoria. Esta parte se observará en Vitis.

En el caso de la lectura, simplemente se verá que tiene la memoria en un inicio, y después se verá cómo cambian estas señales mediante el debugger, y ver que los datos que se envían son los correctos.

Por último, al igual que el capítulo de verificación, vamos a dividir este en dos apartados, uno para las transmisiones de ciclo único, y otro para las transmisiones por bloques. Cabe destacar que el maestro diseñado solo presenta transmisiones por bloques en el modo lectura, por lo que solo en ese caso serán evaluadas estas.

Una vez se ha organizado el contenido del capítulo, paso a explicar con más profundidad las diferentes pruebas realizadas.

6.1 Transmisiones de ciclo único.

En este apartado se van a realizar las demostraciones necesarias para corroborar que las transmisiones de ciclo único funcionan correctamente. Para estas demostraciones, los chips de adaptación de tensión sí que funcionaron en algunos casos, por lo que se especificara en aquellos en los que se haya empleado. Comenzare con la escritura.

6.1.1 Escritura.

Debido a que solo hay dos transmisiones que se puedan probar de escritura, voy a explicar la transmisión de dos bytes, pero también voy a situar las imágenes de la transmisión de un byte, para demostrar que funcionan de manera correcta. Estas dos transmisiones has sido las únicas que se han podido comprobar con los chips, ya que a partir de aquí dejaron de funcionar.

Comienzo pues, explicando la transmisión de ciclo único de dos bytes en modo escritura. En primer lugar, voy a explicar que ocurre internamente en la FPGA cuando se realiza una de estas transmisiones. Cabe destacar que en esta imagen se van a ver las señales reales que nos manda el maestro, su comportamiento, y además cómo se comportan las señales del propio esclavo.

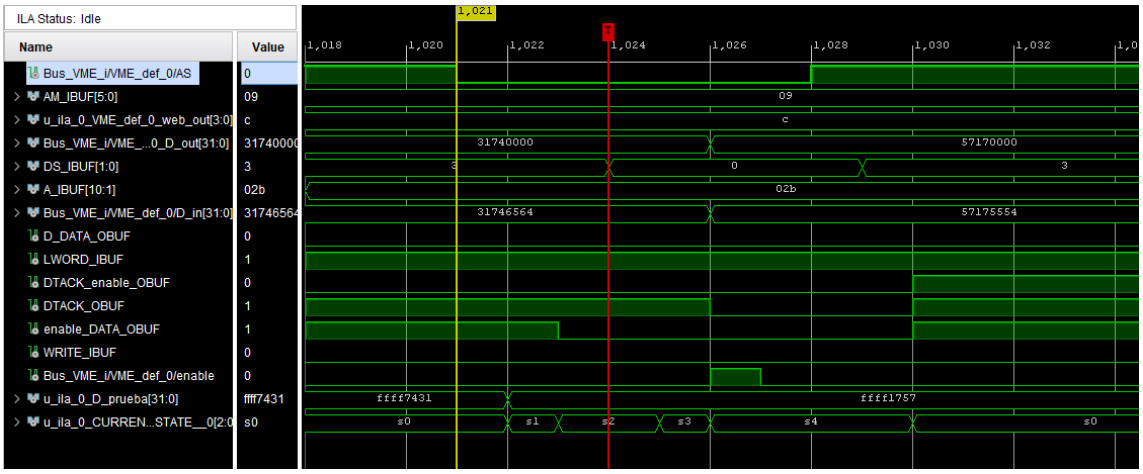


Figura 6.1: Transmisión de ciclo único de dos bytes, en escritura.

En la figura 6.1 se pueden observar las señales externas dentro de la transmisión, y algunas señales internas necesarias para el buen entendimiento del protocolo.

Por una parte, se tienen las señales de control, las cuales presentan el mismo comportamiento que se tiene en la verificación. En este caso, los DS deben de cambiar a “00” ya que es su posición para la transmisión de dos bytes.

En cuanto a las señales de información, el AM se situarán a 0x09 ya que el maestro solo ha implementado ese AM, y el 0x0b para la transferencia de bloques, por lo que solo se emplearan estos dos en todas las transmisiones. En este caso es muy importante fijarse en la dirección que ha sido escrita, en el bus de A sería igual a 0x2b, lo cual se traduce como la dirección 0x56. También nos fijaremos en los datos enviados que se guardan en memoria, lo cuales son 0x3174. Esta información es muy importante, ya que más adelante se verá que se ha guardado correctamente en esta.

A parte de estas señales externas, podemos observar cómo las señales necesarias para el buen funcionamiento de los adaptadores de tensión funcionan de manera adecuada. Así, si lo comparamos con la señal interna CURRENT_STATE, la cual también se puede observar cómo funciona de manera adecuada, se puede observar que cambian cuando se quería, es decir, D_DATA cambia en función del WRITE, y DTACK_enable y enable_DATA cambian cuando la dirección es la correcta. (En este ejemplo concreto hubo un fallo con el maestro al principio en una transmisión, lo que ocasiono que el DTACK_enable se quedase bloqueado. No obstante, cuando se observen las transmisiones siguientes se verá que funciona perfectamente, igual que en la verificación)

En este caso no se va a poder mostrar si se ha escrito bien o no en la memoria, debido a que, en este punto, los chips de adaptación encargados de transistor los datos ya estaban dando problemas.

No obstante, voy a mostrar ahora la transmisión de ciclo único de un byte, en escritura, y en este caso sí que se podrá observar de manera adecuada que se escribe bien en memoria.

En la figura 6.2 se puede observar esta transmisión:

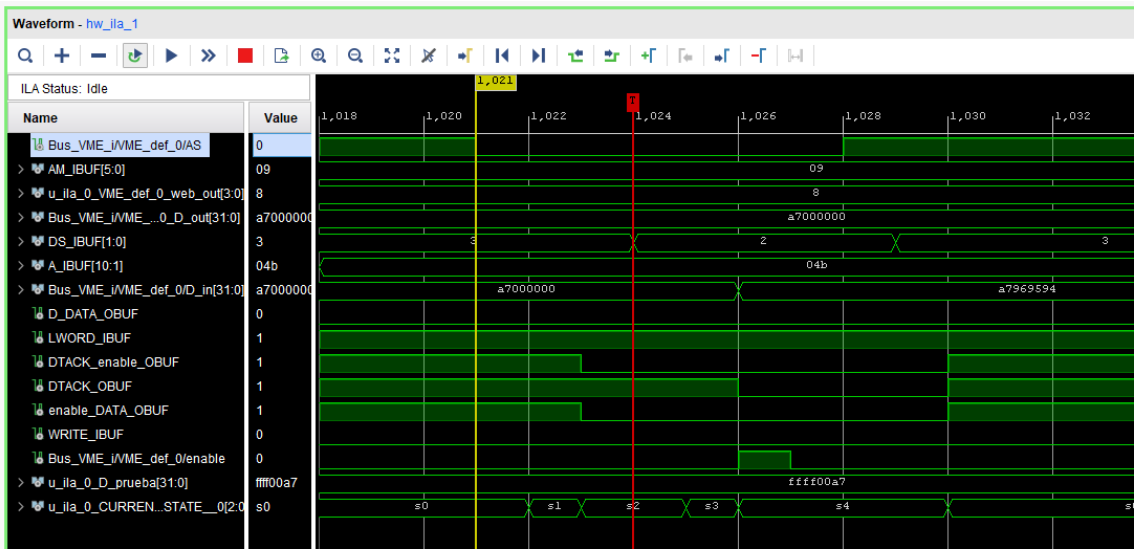


Figura 6.2: Transmisión de ciclo único de un byte, en escritura.

El funcionamiento es muy similar al de la figura 6.1. Las únicas diferencias entre estas dos transmisiones son los DS, que en este caso es necesario que se ponga a “10”, la dirección, que en este caso A es igual a 0x4b, lo que se traduce en la dirección 0x97 en la memoria (en este caso es impar debido a los DS), y el bus de datos, el cual presenta 0xa7, que será el dato que se escriba en esta dirección.

En la figura 6.2 se puede observar de manera clara el funcionamiento de DTACK_enable, la cual presenta el mismo funcionamiento que el enable_DATA, y que, en este caso no se ha producido ningún problema.

Una vez se ha explicado brevemente que ha cambiado, voy a mostrar que en la memoria se escribe de manera correcta la información. Para ello, voy a mostrar la figura 6.3:

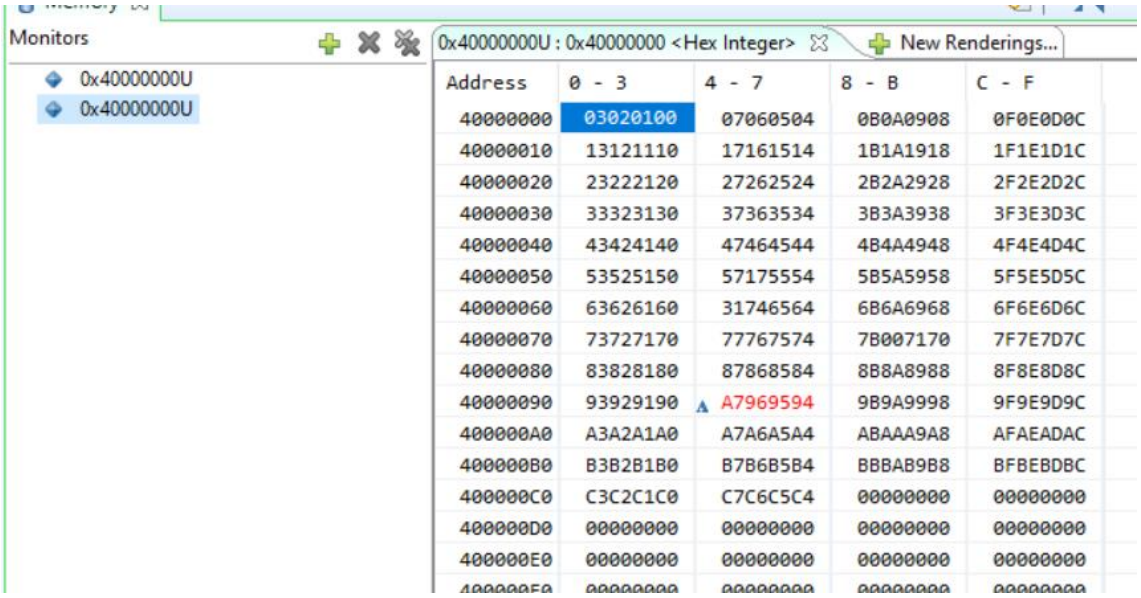


Figura 6.3: Valor escrito en memoria por la transmisión de la figura 6.2.

En primer lugar, es necesario entender por qué esta la memoria llena de valores. Esto es debido al programa que se ha escrito en Vitis, para poder inicializar de valores la memoria, para que a la hora de leer de esta se puedan obtener valores y no solo ceros. Esto será explicado más adelante en el apartado de lectura.

También cabe destacar una cosa y es que, ya en esta transmisión, los datos se mandaron sin chips de adaptación de señales, es decir, se conectó con un hilo los pines de las dos placas.

Una vez introducido esto, si se fija uno en los números en rojo, se puede observar que se corresponde con el bloque de memoria que va de la dirección 0x94 a la 0x97, es decir, que engloba la dirección en la que se quiere escribir. En este caso, pese a que arriba aparezca la dirección escrita como si la memoria fuese big endian, la memoria es Little endian, por lo que el 0xA7 que se tenía que escribir en la dirección 0x97, se ha escrito en el primer byte empezando por la izquierda.

Así, se puede decir que se ha demostrado que las dos transmisiones de ciclo único de escritura posibles funcionan de manera correcta, además de haber demostrado también el buen funcionamiento de la memoria. Estas dos transmisiones han sido las únicas que han podido ser realizadas con los chips de adaptación de tensión, por lo que a partir de ahora simplemente se han conectados los pines de ambas placas con un hilo.

6.1.2 Lectura.

Antes de empezar con la lectura de los datos de memoria, es necesario ver que se ha escrito en la memoria. De esta manera, en la figura 6.4 se muestra que contiene esta:

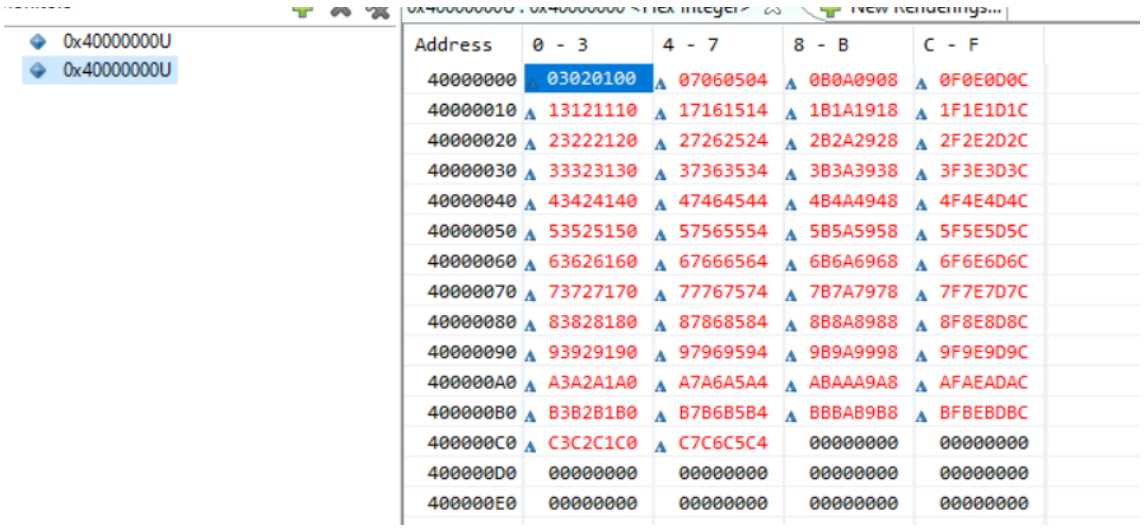


Figura 6.4: Valores iniciales en memoria.

Estos valores se han escrito mediante el programa diseñado en Vitis para su inicialización. Este se puede ver en el apartado 3.1, donde se muestra el funcionamiento de la memoria, y cómo se comporta esta, además de cómo es escribe en ella en la parte C. Esta imagen va a servir de ayuda para demostrar que los valores que se envían por el bus de datos hacia el maestro son correctos, por lo que más adelante será referenciada.

Una vez explicado esto, voy a mostrar la transición en modo lectura que se va a demostrar. En este caso, se va a poder ver la transmisión de ciclo único de cuatro bytes, en modo lectura. Antes mostrar que ocurre dentro del prototipo, es necesario saber que esta transmisión no llega de manera correcta al maestro, ya que solo se ha implementado un bus de datos de dieciséis bits. No obstante, como ya se ha visto en funcionamiento transmisiones de uno y dos bytes, me parece necesario demostrar también que las transmisiones de más bytes, o bytes desalineados funcionarían correctamente, por lo que, para ello, lo demuestro con la de cuatro bytes. En la figura 6.5 se puede observar esta transmisión:

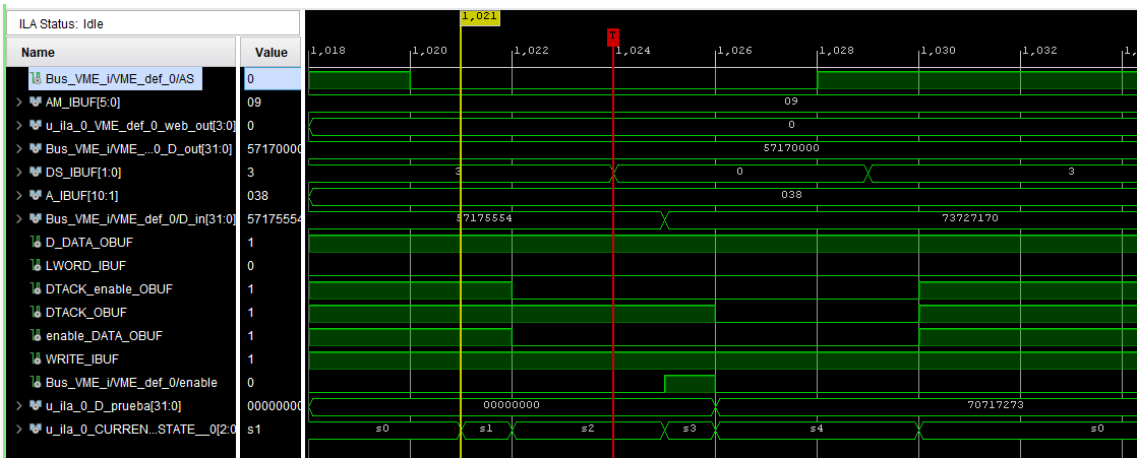


Figura 6.5: Transmisión de ciclo único de cuatro bytes, en lectura.

Se puede observar cómo las señales de control presentan el mismo funcionamiento que en las anteriores transmisiones. Los DS en este caso tienen que valer “00” cuando se produzca el cambio. Además, se puede observar cómo las señales de control de los adaptadores de tensión también funcionan de manera adecuada, para el buen funcionamiento de estos, sin embargo, en esta transmisión, no se ha empleado ninguno.

En cuando a las señales de información, los AM presenta el valor de 0x09, es decir, el que se tiene que emplear para este tipo de transmisiones. El bus de A presenta el valor 0x38, que se traduce como la dirección 0x70. De esta manera, observando el bloque de memoria correspondiente a esa dirección en la imagen 6.4, se puede observar en la señal D_prueba como coinciden los valores, aplicando lo que se lleva comentando en todos los capítulos sobre que la memoria del bus es big endian, y la memoria interna de la FPGA es Little endian.

De esta manera, queda demostrado que la lectura funciona de manera correcta en las transmisiones de ciclo único. Así, una vez que se ha enseñado cómo funciona el esclavo en cuando a las transmisiones de ciclo único, voy a pasar a explicar el funcionamiento de las transmisiones por bloques, las cuales solo están implementadas en el maestro la parte de lectura, y serán las únicas que se muestren.

6.2 Transmisiones por bloques.

En este capítulo voy a demostrar que las transferencias por bloques funcionan de manera adecuada. Debido a que el maestro solo implementa estas transferencias para el modo lectura, estas serán las que se demuestren. Esto es debido a que el diseño real el cual será realizado por la empresa no emplea transferencias por bloques, por lo que no se estuvo mucho tiempo diseñándolas.

También veo necesario aclara que, en este punto, al igual que en las transferencias de ciclo único de lectura, los chips de adaptación de tensión no han sido empleados, por lo que, las conexiones han sido realizadas con un hilo.

Como solo se van a mostrar las de lectura, no se va a dividir el capítulo en subapartados. En cuanto a las transmisiones que se han realizado, se van a poder observar tanto la transmisión por bloques de un byte, como la de dos bytes. Realmente, en cuanto a la transmisión son muy parecidas, pero creo necesario, ya que son solo dos, demostrar que ambas funcionan de manera adecuada. Cabe destacar también que la transferencia de cuatro bytes también funcionaria, pero en este caso si he querido mantener la esencia de que el bus de datos es simplemente de dieciséis bits, y no de treinta y dos.

En la figura 6.6 se puede observar la transferencia por bloques de un byte:

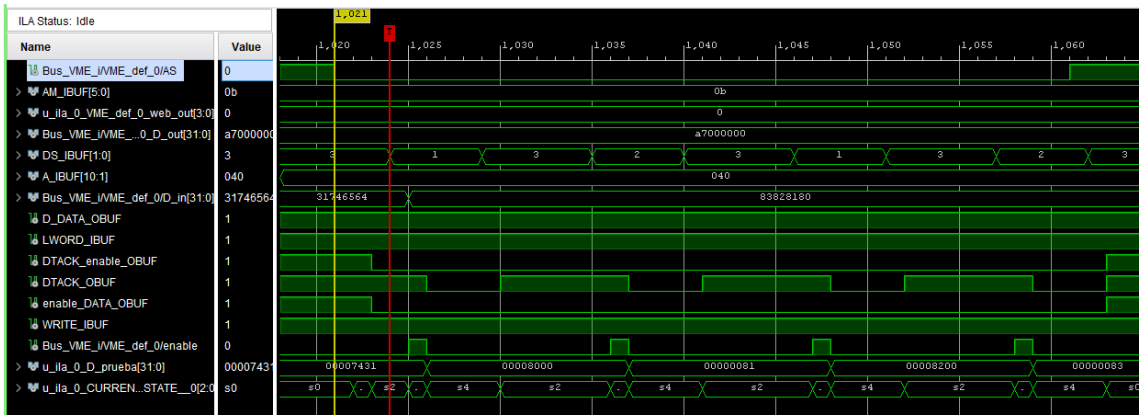


Figura 6.6: Transmisión por bloques de un byte, en lectura.

Para explicar la figura 6.6 comenzare explicando las señales de control. El AS se pondrá a ‘0’ para iniciar la transmisión. Y este se volverá a poner a ‘1’ cuando esta se termine, como ocurría en la verificación. Lo DS presentan el comportamiento que requieren, es decir, estos van cambiando del estado de reposo, al estado en el que se transmite un byte, y además oscilan entre “01” y “10” en función del byte que se quiera transmitir.

Con respecto a las señales de información, el AM está a 0x0b, que es la transmisión que se corresponde. El bus A indica la dirección de inicio, menos el ultimo bit a la derecha, de la transición, que sería 0x40, la cual corresponde con la dirección 0x80 dentro de la memoria. Esta dirección ira cambiando cada dos transmisiones, para poder mandar el dato correcto. No se puede ver este cambio en la dirección, pero si nos fijamos en la figura 6.4, que se corresponde con los valores que presenta la memoria al inicio, y nos fijamos en D_prueba, se puede ver perfectamente como los datos son los correctos, por lo que se tiene certeza de que esta suma en la dirección interna se realiza correctamente. Además de ser los datos correctos, también se puede observar que se sitúan en la posición correcta dentro del bus D.

Con respecto a las señales de control para los chips, se puede observar cómo D_DATA cambian en función del WRITE, y las otras dos señales, enable_DATA y DTACK enable, se mantienen activas durante todo el proceso de comunicación.

Una vez explicada la transmisión por bloques de un byte, voy a mostrar la transmisión por bloques de dos bytes. En la figura 6.7, se puede observar esta transmisión:

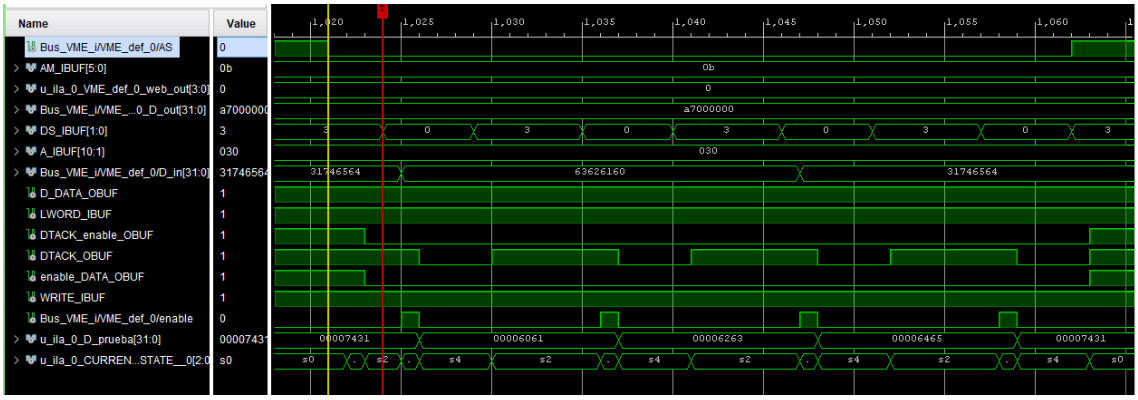


Figura 6.7: Transmisión por bloques de dos bytes, en lectura.

En primer lugar, el AS presenta el mismo comportamiento que en la transmisión anterior, pero los DS no. Así, esto pasaran del estado de reposo, al estado “00” cada vez que se realice una transmisión, y, este no ira cambiando de valor.

La dirección en este caso es diferente. Así en el bus A tenemos 0x30, lo cual se traduce como la dirección de memoria 0x60. De esta manera, esta dirección ira aumentando su valor de uno en uno en cada transmisión. Como he dicho antes, no se puede ver el buffer que cambia este valor, peor si se pueden ver los datos, por lo que, si nos fijamos en el D_prueba, y lo comparamos con la figura 6.4, se puede observar claramente como los datos que se toman son correctos.

Con respecto a las señales de los chips, funcionan exactamente igual que en la transmisión anterior.

De esta manera, queda demostrado que, si enfrentamos el proyecto con un maestro VME (en este caso es otro prototipo), funciona de manera correcta, y se transmite bien la información. En cuanto al proyecto real se refiere, quedaría por evaluar de nuevo los chips de adaptación de tensión, incluyendo los condensadores, y aplicar el VME dentro de la tarjeta que se quiere diseñar, lo cual se va completamente de este proyecto.

7. Conclusiones

Se ha podido observar a lo largo de lo expuesto en esta memoria, que se han conseguido los objetivos marcados en su inicio y que fueron comentados en la introducción.

Se ha estudiado el protocolo del bus VME, analizando el funcionamiento de sus entradas, tanto de datos como de control, viendo qué deben realizar éstas en las diferentes transmisiones del bus, así, y también se ha estudiado cómo debe ser la salida. Además, se ha realizado un profundo estudio sobre los diferentes modos de transmisión, indagando más en aquellos que se han desarrollado dentro del diseño realizado en el ámbito de este Trabajo Fin de Grado.

Se ha observado que, para poder realizar el prototipo, era necesario una parte hardware y otra parte software. La parte hardware se encarga de realizar la comunicación del bus VME. La parte software ha sido imprescindible para poder tratar los datos que se obtienen de la comunicación del bus de la parte hardware y también ha servido para poder inicializar la memoria y realizar así las diferentes pruebas.

Se ha realizado en la parte hardware el diseño en VHDL de un esclavo VME, empleando una máquina de estado, la cual ha sido explicada en el capítulo tres. Esta parte también incluye las conexiones necesarias en el diseño de bloques para poder comunicar la parte hardware (implementada en FPGA) y la parte software (implementada en un microprocesador). Se ha visto cómo se ha de usar el bloque de memoria, necesario para poder almacenar los datos del bus, y se ha solventado el problema de que la memoria del bus es big endian, y la de Vivado Little endian.

Se ha verificado, mediante simulaciones, que los diferentes tipos de comunicaciones funcionan correctamente y sin ningún inconveniente. Para ello se ha verificado una comunicación tanto de lectura, como de escritura, de los tres modos que hay desarrollados, que son transmisión de ciclo único de datos alineados, transmisión de ciclo único de datos desalineados y transmisión por bloques. Para complementar las verificaciones expuestas en esta memoria, se ha añadido el ANEXO I en el cual, aparte de poder ver la explicación de las transmisiones comentadas anteriormente, se incluyen capturas de la simulación de todas las transmisiones que se han implementado en el bus.

Se han utilizado adaptadores de tensión y se ha diseñado una placa PCB, necesaria para probar el esclavo con un maestro real. De esta manera, se ha probado su funcionamiento. Aunque no se han obtenido los resultados esperados, sí se han realizado una propuesta de mejora.

Por último, se ha comprobado que el diseño funciona correctamente, enfrentándolo al prototipo de un maestro real. Se ha explicado el inconveniente que se ha producido cuando se ha ido a usar el chip de adaptación de tensión, y por qué algunas transmisiones se han tenido que demostrar sin chip. También se ha explicado la solución que se ha propuesto para ello, aunque finalmente, por cuestión de tiempo no ha sido posible implementarla y poder ver a ciencia cierta que es la solución correcta.

De esta manera, en primer lugar, se han demostrado las transmisiones de ciclo único, tanto de lectura como escritura, alineadas. Se ha visto en cuales se ha usado el chip y en cuáles no. Se ha explicado también, por qué no ha podido ser posible demostrar las transmisiones desalineadas, debido a que el bus de datos que se ha diseñado, por problemas de pines bidireccionales de la placa de desarrollo, ha tenido que ser de 16 bits en vez de 32 bits.

Por último, se ha demostrado que las transmisiones por bloques de lectura funcionan correctamente, aunque no han podido ser demostradas estas transmisiones en modo escritura ya que el maestro no ha implementado esta función.

Se ha podido ver tanto en la verificación como en la demostración, el funcionamiento de la memoria, imprescindible dentro del diseño, y cómo se comporta.

En conclusión, aún con los inconvenientes que se han producido al final debido a que los adaptadores de tensión que no han permitido pruebas con un maestro real, ha quedado suficientemente comprobado que el esclavo del bus VME desarrollado en este Trabajo de Fin de Grado ha sido diseñado correctamente y su funcionamiento es también correcto.

8. Bibliografía.

[1] American National Standards Institute, Inc., “ANSI/VITA 1-1994 (S2011), American National Standard for VME64”, Approved 1994, Reaffirmed 2002, Stabilized Maintenance 2011.

[2] Xilinx, Inc., “Block Memory Generator v8.4, LogiCORE IP Product Guide (PG058)”, August 6, 2021.

[3] Xilinx, Inc., “AXI GPIO v2.0, LogiCORE IP Product Guide (PG144)”, October 5, 2016.

[4] Xilinx, Inc., “FMC XM105 Debug Card User Guide (UG537 v1.3)”, June 16, 2011.

[5] AVNET, Inc., “ZedBoard (ZynqTMEvaluation and Development) Hardware User’s Guide v 2.2”, January 27, 2014.

[6] Texas Instruments, Inc., “SN74LXCH8T245 8-bit Translating Transceiver with Configurable Level Shifting”, March, 2021.

[7] Markus Joos, “VME tutorial”, (disponible en http://pen.phys.virginia.edu/daq/vme/vme-tutorial.pdf; última consulta 7/05/2023).

[8] Larry Davis, “VME Bus Versa Module Europa IEEE 1014-1987”, (disponible en http://www.interfacebus.com/Design_Connector_VME.html; última consulta 7/05/2023).

Anexo I.

En este anexo se van a mostrar las diferentes capturas realizadas sobre la simulación de los diferentes modos de transmisión. De esta manera, se van a ver todos los modos que se han implementado en el esclavo, y que, por tanto, se han desarrollado y estudiado.

Las imágenes de este anexo van a estar organizadas en:

- Imágenes de ciclo único, escritura alineada.
- Imágenes de ciclo único, escritura desalineada.
- Imágenes de ciclo único, tanto de que hay escrito en memoria, como de la lectura de esa memoria, alineada.
- Imágenes de ciclo único, tanto de que hay escrito en memoria, como de la lectura de esa memoria, desalineada.
- Imágenes de transferencia de bloques, de escritura.

Las de lectura, como se ven tanto la de cuatro bytes, en verificación, como las de uno y dos bytes en la demostración, no las he recogido aquí, ya que quedan claramente demostradas que funcionan.

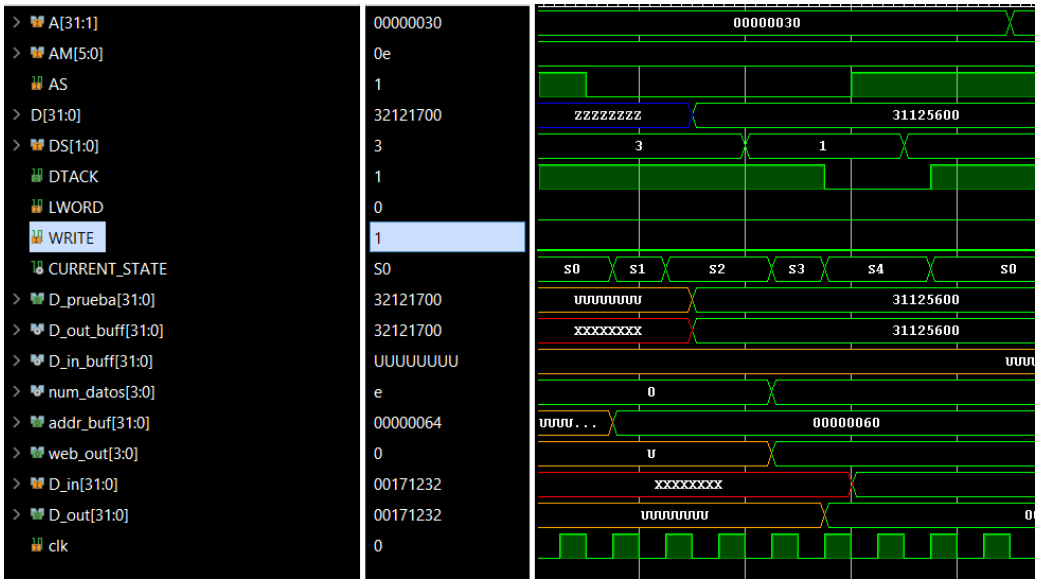


Figura I.1: Simulación de escritura en el modo dos bytes.

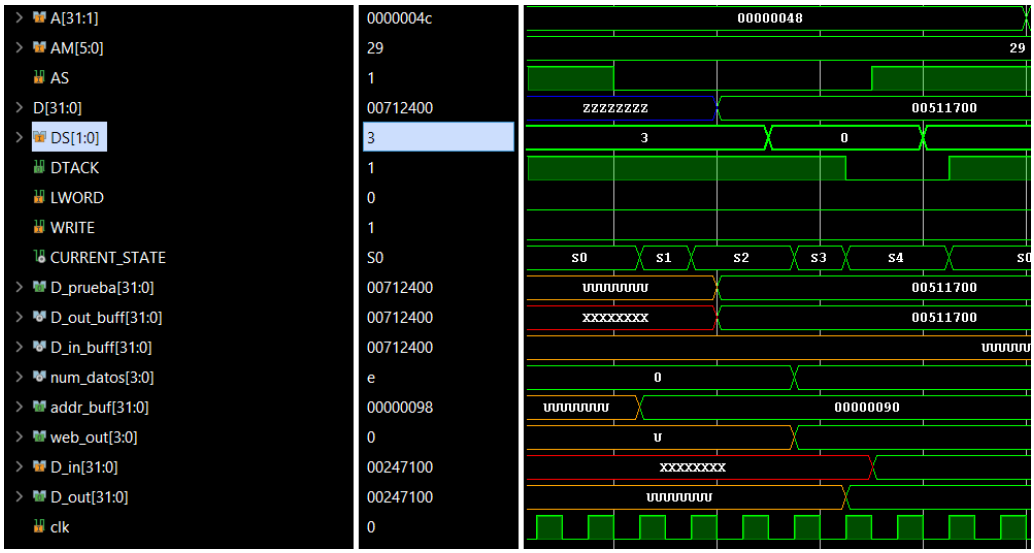


Figura I.2: Simulación de escritura en el modo cuatro bytes.

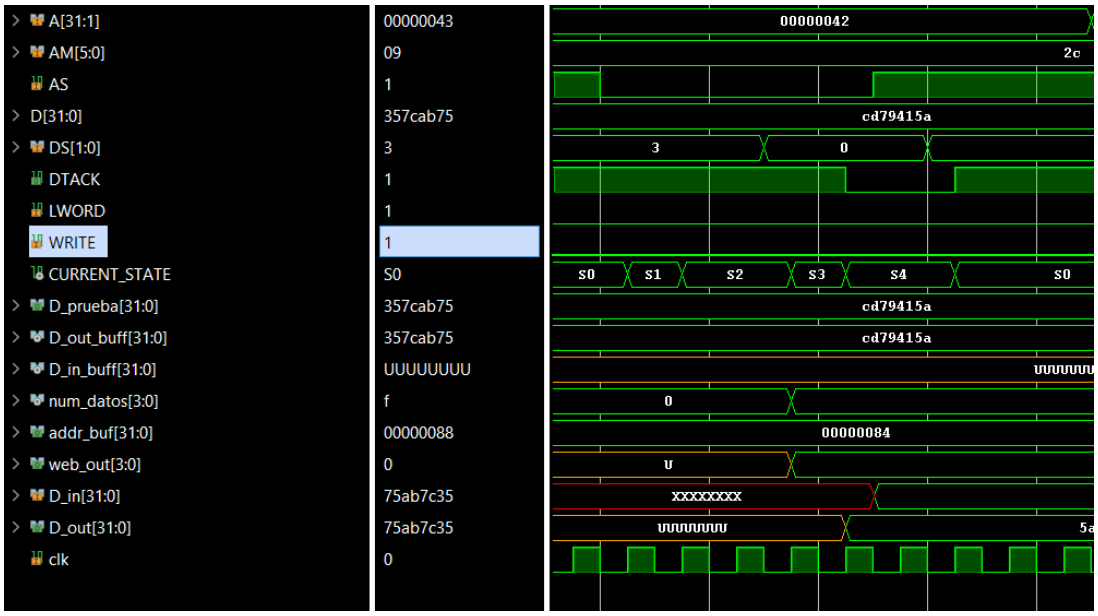


Figura I.3: Simulación de escritura en el modo dos - cero bytes (desalineado)

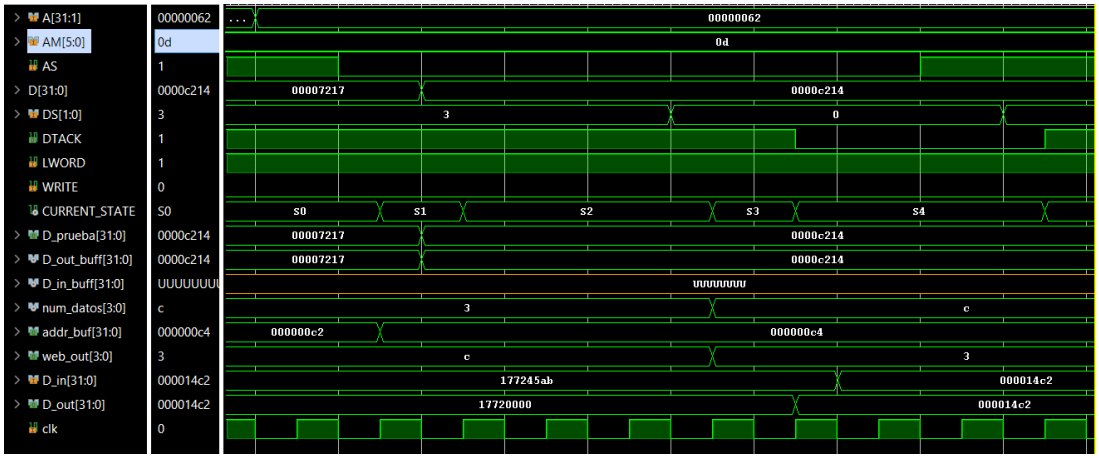


Figura I.4: Simulación de escritura en el modo dos - cero bytes (desalineado)

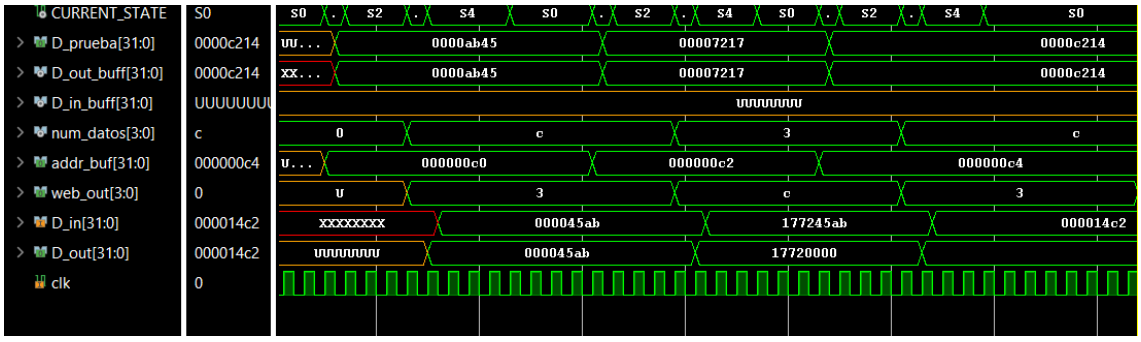


Figura I.5: Que contiene la memoria para la lectura de cuatro bytes (alineada)

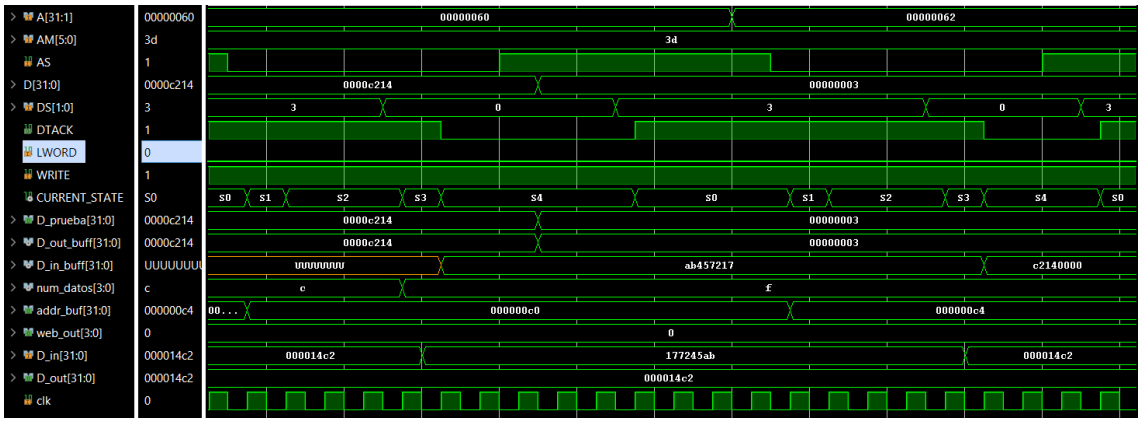


Figura I.6: Simulación de lectura en el modo cuatro bytes

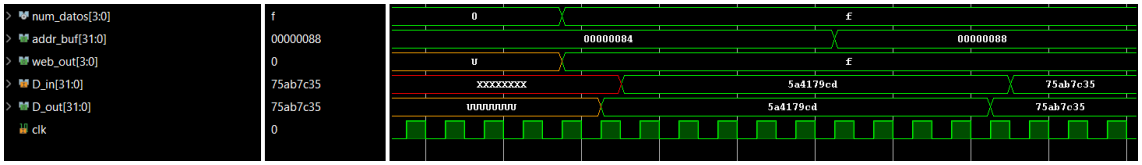


Figura I.7: Que contiene la memoria para la lectura de un byte (alineada)

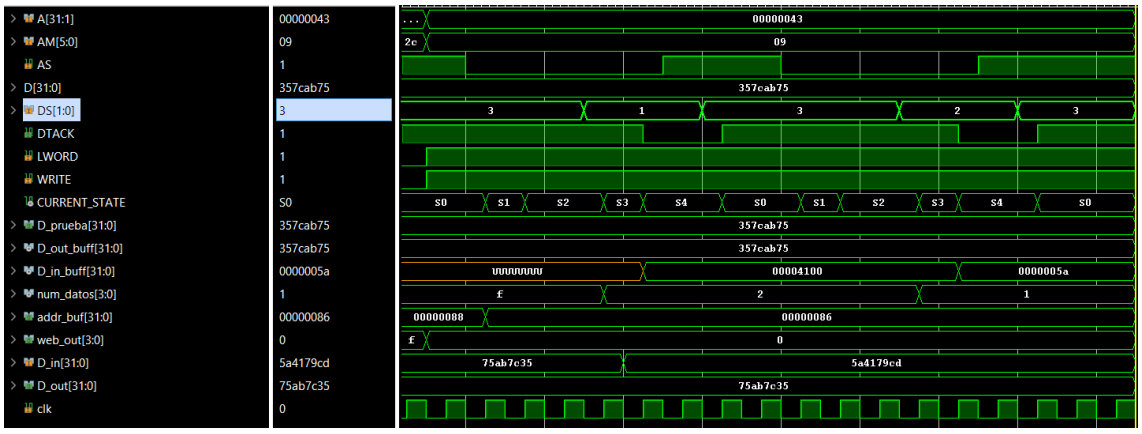


Figura I.8: Simulación de lectura en el modo un byte

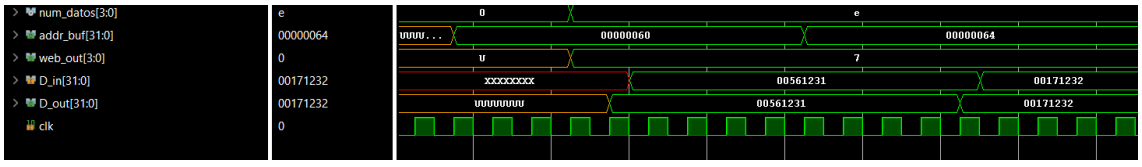


Figura I.9: Que contiene la memoria para la lectura de tres - un bytes (desalineada)

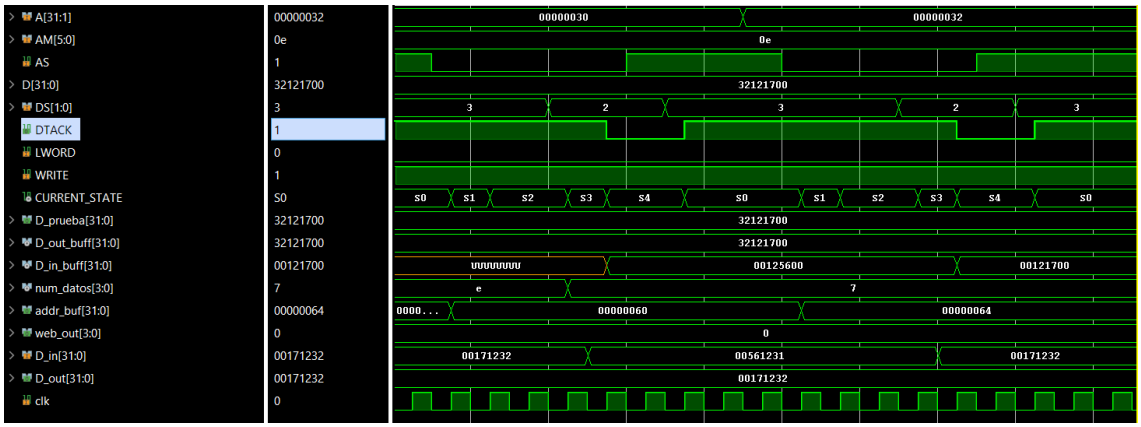


Figura I.10: Simulación de lectura en el modo tres - un byte (desalineado)

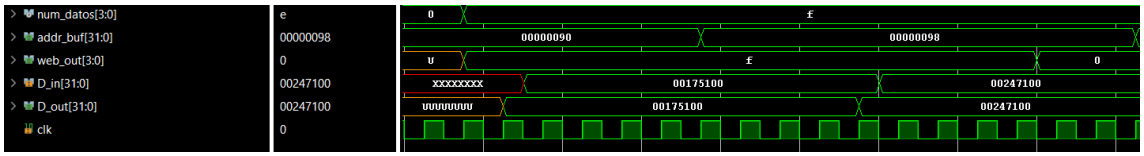


Figura I.11: Que contiene la memoria para la lectura de dos - cero bytes (desalineada)

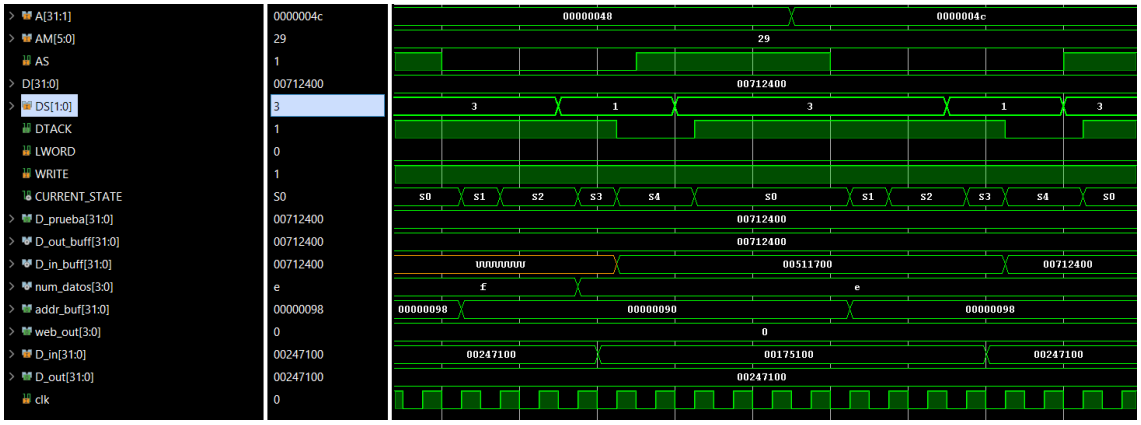


Figura I.12: Simulación de lectura en el modo dos - cero byte (desalineado)

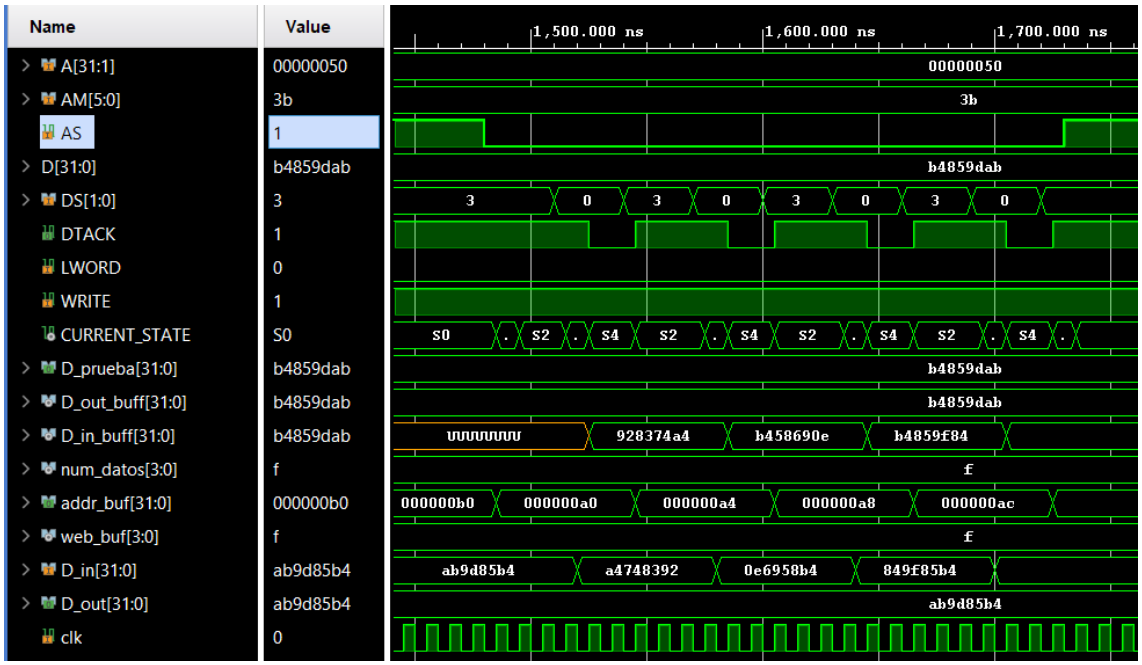


Figura I.13: Simulación de escritura en el modo cuatro byte, en una transmisión por bloque.