Modelo de Comunicación Basado en el Estándar AMBA AHB

Memoria del Proyecto de Fin de Carrera
de Ingeniería en Informatica
realizado por
Manuel García Cazalilla
i dirigido por
Antonio Portero Trujillo
Bellaterra, 12 de Setiembre de 2006

Tabla de contenidos

Artículo. Modelo de comunicación basado en el protocolo AMBA AHB

En el siguiente artículo se estudiara la implementación de un modelo de comunicación basado en el protocolo ARM AMBA AHB. El objetivo del modelo es la implementación de dos módulos, un master y un esclavo, que posibiliten el intercambio de información mediante un bus AMBA AHB. La finalidad del código resultante es que pueda ser integrado en cualquier sistema para resolver sus necesidades de comunicación de una forma estándar.

Modelo de comunicación basado en el protocolo AMBA AHB

Manuel García Cazalilla, Proyecto de Fin de Carrera de Ingeniería en Informatica
Escuela Técnica Superior de Ingeniería – Universidad Autónoma de Barcelona

Resumen. En este documento se presenta un modelo de comunicación basado en el estándar AMBA AHB y consistente en la interconexión por bus de un módulo master con otro esclavo. Dicho modelo se describe en SystemC y se prueba en un código para el cálculo de la transposición de matrices.

I. INTRODUCCIÓN

La era digital unida a la sociedad de consumo, hace que a diario surjan nuevas y variadas aplicaciones que requieren el uso de tecnología. Por ello, la industria de desarrollo no para de producir y utiliza herramientas de todo tipo, a veces propias, otras bajo licencia y ocasionalmente gratuitas y de libre uso. Por otro lado, la tecnología ARM *(Advanced Risc Machines)* [1] tiene aplicación en multitud de campos de desarrollo como en automoción, entretenimiento y telefonía móvil entre otros. Esta tecnología, que cubre aproximadamente el 75% del mercado de procesadores empotrados, esta basada en principios RISC y se caracteriza por un bajo consumo, un bajo coste y por el uso del estándar AMBA *(Advanced Microcontroller Bus Architecture)* [2] como interface de comunicación a través de un wrapper.

El uso de modelos puede facilitar el desarrollar de forma rápida y fiable, lo cual hace de estos una herramienta de gran utilidad, interés y potencia. De ahí que el desarrollo en SystemC [3] de un modelo de comunicación simple basado en AMBA AHB tenga un buen grado de interés y una utilidad más que razonable. El modelo propuesto abarca gran parte del estándar aunque deja algunos aspectos a parte debido a la complejidad de este.

II. ARQUITECTURA AMBA

El estándar AMBA define tres tipos de bus:

- Advanced High-performance Bus (AHB)
- Advanced System Bus (ASB)
- Advanced Peripheral Bus (APB)

A grandes rasgos, el primero esta diseñado para comunicar sistemas de alto rendimiento como procesadores y módulos con altas frecuencias de reloj. ASB también se utiliza para sistemas de alto rendimiento pero que no necesitan las prestaciones que ofrece AHB. En cambio, APB esta pensado para comunicar periféricos de bajo consumo y de menores prestaciones.

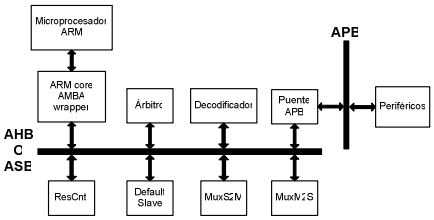

Figura1. Típico sistema conectado por AMBA

Los componentes básicos [2] de un típico sistema AMBA (Fig. 1) son el controlador de reset, el árbitro, el decodificador, los multiplexores y el esclavo por defecto, siendo el puente de APB necesario solamente si el sistema requiere interconectar el bus AHB o ASB con el APB.

Centrando la materia en el AMBA AHB [4], en éste pueden coexistir hasta un máximo de 16 módulos master, capacidad limitada por la señal HMASTER de 4 bits generada por el árbitro para indicar qué Master tiene el control del bus en cada momento. Uno de estos 16 posibles master suele ser el master por defecto que es quien toma el control del bus cuando ningún otro master lo va a hacer. Puede realizar alguna acción para la que se le haya programado o permanecer en estado IDLE (parado) y no hacer nada.
En cuanto a los esclavos, no existe un número máximo de ellos, pero sí una limitación dada por la memoria del sistema ya que no puede haber esclavos que no estén mapeados en memoria. En el AMBA se utiliza un total de 32 bits para las direcciones, pero generalmente un esclavo ocupará más memoria que 1 bit, por lo que su número será muy inferior a 2^{32}. Sin embargo, en este caso la inclusión de un esclavo por defecto es obligatorio ya que su principal función es la de alertar a los master

que pretenden acceder a una dirección de memoria inválida.

Respecto a las transferencias por el bus [5], todas ellas se consideran como ráfagas, ya sean de 1, 4, 8 o 16 datos, además de existir la posibilidad de ráfagas de longitud indeterminada. Para cada dato que se recibe o envía a través del bus, previamente se ha de haber especificado una dirección de memoria de procedencia o de destino, con la peculiaridad de que para las ráfagas de longitud indeterminada, el rango entre la primera y la última posición de memoria no puede exceder de 1k Byte.

III. Obtención del AMBA AHB

Para construir el modelo, consistente en un módulo master y otro esclavo, es necesario utilizar un sistema AMBA AHB completo para poder así comprobar que funciona correctamente acorde al protocolo AMBA. Para ello existen dos alternativas posibles, construir el sistema de principio a fin u obtener uno que ya funcione y probar en él los módulos. Visto que la primera opción es inviable dadas las características del presente trabajo, hubo que buscar cómo realizar la segunda alternativa.

El Departamento de Microelectrónica y Sistemas Electrónicos (MiSE) de la Universidad Autónoma de Barcelona dispone de un AMBA AHB escrito en Verilog cuya jerarquía de componentes [6] se muestra en la Figura 2 y posibilita la utilización de la segunda vía comentada. Pero dicho bus tiene el problema de estar escrito en un lenguaje distinto a SystemC, y puesto que se pretende implementar el modelo en éste lenguaje, la conectividad entre ambos se ha de resolver previamente a cualquier otra acción. Para ello, se ha de buscar una herramienta que la posibilite de un modo transparente.

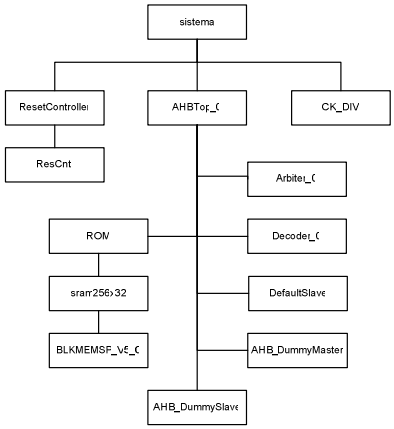

Figura 2. Sistema AMBA en Verilog

Una primera alternativa es utilizar un software gratuito llamado Verilator [7] que funciona bajo sistemas like unix y que es capaz de compilar ambos código con-juntamente. Pero tras algunos ensayos, se decidió desechar esta opción principal-mente porque el tiempo de adecuación, ***set up,*** para poner en marcha la herramienta es demasiado alto

Otra alternativa, es utilizar el software ModelSim [8]. Este programa permite simular un sistema cuyos módulos se implementan en distintos lenguajes, entre los cuales se encuentran Verilog y SystemC. De ese modo, no hay más que substituir los módulos master y esclavo contenidos en el bus Verilog por sus análogos desarrollados en SystemC y que implementan el modelo. Luego de esto, se ha de crear una serie de señales de estímulos “test bench” que estimule las señales básicas de reloj y de reset y añadir un segundo módulo que instancie todo el sistema [9].

IV. Fijando límites al modelo

La complejidad de un sistema AMBA es lo suficientemente elevada como para que sea difícil desarrollar un modelo completo en un proyecto de final de carrera. De ahí que haya que limitar el modelo marcando unos objetivos y mirando de ceñirse a ellos.

En este caso, la primera restricción importante es la de que el modelo escogido se basa en el AMBA AHB, por lo que cualquier cuestión

referente a los buses ASB y APB queda totalmente aparcada.

Respecto al AMBA AHB, en el momento en que un master toma el control del bus e inicia una transferencia, todo el sistema queda reducido a un master y al esclavo que éste selecciona para transmitir. Puesto que únicamente un master puede ejercer el control del bus simultáneamente, esto permite una segunda restricción que limita el modelo a un único master y a un solo esclavo, ya que por otra parte se pretende que el modelo sirva para una comunicación entre dos extremos. Ésta no es una limitación demasiado restrictiva dado que si se modela bien la comunica-ción, se puede extrapolar fácilmente el modelo e incluir varios módulos master y esclavos únicamente configurando el árbitro, el decodificador y los multiplexores para que admitan tal redimensión.

La siguiente restricción es la no utilización de la señal HSPLIT. Aparte de no estar implementada en dicho bus con lo que no se puede probar su correcto funcionamiento, puesto que únicamente habrá un master y un esclavo, la utilización de HSPLIT no es necesaria ya que el master, al ser el único utilitario del bus, siempre lo tendrá disponible y no será necesario desalojarlo al no perjudicar a ningún otro master. Por este mismo motivo, tampoco es necesario incluir la petición de bus bloqueante mediante el uso de la señal HLOCK que habilita a un master a pedir el bus de forma que no lo pierda hasta concluir su operación

V. CONSTRUCCIÓN DEL MODELO

El objetivo del presente trabajo es el de obtener dos módulos escritos en SystemC, un master y un esclavo del bus AMBA AHB. Una vez implementados [10] y probados [11], estos respetarán y cumplirán el protocolo AMBA AHB por lo que para su correcto funcionamiento ya no será necesaria toda la infraestructura del bus cedido por el MiSE, si no que añadiendo un árbitro, un decodificador y un esclavo por defecto simples creados específicamente para cada nuevo sistema, será suficiente. El árbitro únicamente ha de conceder el bus al master cada vez que éste se lo pida, mientras que el decodificador solo ha de activar al esclavo, si se programa su dirección, y al esclavo por defecto en cualquier otro caso. Todo ello conformará un sistema (Fig. 3) abierto a modificaciones que se integrará en el código de su utilitario y que realizará las comunicaciones que se le demanden.

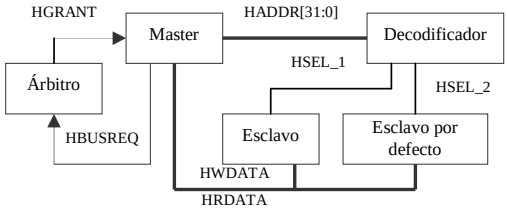

Figura 3. Esquema del modelo

Para dicha integración (Fig. 4), se propone un código esqueleto con la implementación del modelo y que el utilitario del mismo ha de rellenar con su código. Por tanto, se construyen dos módulos, uno con el proceso master, MasterAMBA_AHB, y otro con el proceso esclavo, EsclavoAMBA_AHB.
La idea es que se lancen los procesos de un mismo módulo como procesos hilados *"threads"* que se ejecutarán concurrentemente y que podrán comunicarse por medio de variables compartidas al pertenecer al mismo módulo. De ese modo, los procesos utilitarios podrán programar a los del modelo para que éstos ejecuten las comunicaciones requeridas.

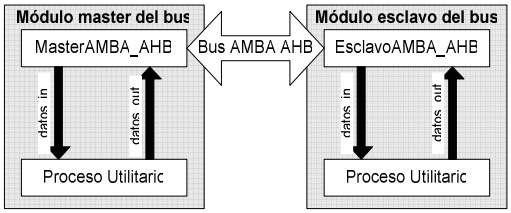

Figura 4. Esquema del modelo

Las variables compartidas a utilizar serán distintas para cada caso, master [12] y esclavo [13], pero ambos utilizarán dos colas, sc_fifo de SystemC, para intercambiar los datos con los procesos utilitarios. De ese modo, en la

cola datos_in el proceso del AMBA escribirá los datos que reciba a través del bus y en datos_out, el proceso utilitario pondrá los datos que desea que se envíen por el AMBA AHB.

VI. Construcción del Master

Las operaciones que un master realiza en un bus AMBA AHB son fácilmente diferenciables. Por tanto, una forma sencilla y elegante de implementar su funcionalidad [14] es mediante una máquina de estados (Fig.5).

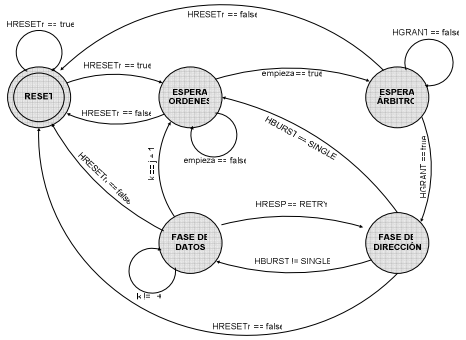

Figura 5. Máquina de estados del master

En ésta máquina de estados pueden verse los estados derivados del protocolo AMBA AHB y un estado adicional, ESPERA MASTER, donde master queda a la espera de recibir órdenes del proceso utilitario por medio de las variables compartidas. Dicha máquina de estados se implementa [14] fácilmente en el código mediante una estructura switch donde cada caso corresponde un estado distinto:

```
switch(estado) {
  case RESET: /*código*/ break;
  case ESPERA_ORDENES: /*código*/ break;
  case ESPERA_ARBITRO: /*código*/ break;
  case FASE_DE_DIRECCION: /*código*/ break;
  case FASE_DE_DATOS: /*código*/
}
```

En el estado RESET, se inicializan las señales y las variables locales y de sincronización entre procesos, además de programar el nuevo estado.

ESPERA_ORDENES es el estado en el que permanece el master hasta recibir una orden del proceso utilitario. Cuando esto sucede, comprueba que las variables de sincronización hayan sido programadas correctamente e inicia la comunicación por el AMBA pidiéndole el bus al árbitro y programando el siguiente estado.

En ESPERA_ARBITRO se retiene el inicio de la comunicación hasta que el árbitro concede el bus mediante la activación de la señal HGRANT del master. Una vez activada dicha señal, se programa el nuevo estado para iniciar la transmisión de datos.

En FASE_DE_DIRECCION se inicia la comunicación con la programación de las señales de control y de dirección relativas al primer dato y que provocarán la activación del esclavo por parte del decodificador.

Finalmente en FASE_DE_DATOS se transmiten los datos por los buses HWDATA o HRDATA según si se envía información o se recibe. Además, si se trata de una transmisión en ráfaga de múltiples datos, se solapan en el mismo ciclo de reloj la programación de las señales de control y de dirección relativas al dato X+1 y el envío del dato X.
Terminada la operación se programa en el bus de direcciones HADDR la dirección donde se mapea al master. De este modo, puesto que no hay ningún otro master en el bus que entre a operar y cambie la dirección, se desactiva al esclavo sin que se active el esclavo por defecto. Así, el esclavo se da cuenta de que ha concluido la operación y puede presentar los resultados a su proceso utilitario.

VII. Construcción del Esclavo

De igual modo que ocurre con el master, la funcionalidad del esclavo [15] es fácilmente diferenciable, por lo que la implementación de una máquina de estados (Fig. 6) supone la mejor opción para estructurar el código del proceso.

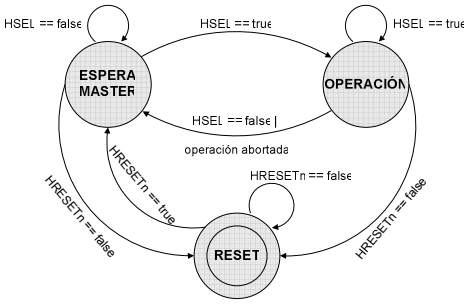

Figura 5. Máquina de estados del esclavo

El estado de partida, RESET, inicializa las variables locales y de sincronización, además de las señales básicas. Después programa el siguiente estado a ejecutar.

En ESPERA_MASTER queda retenido el código del proceso hasta que se produce la activación por parte del decodificador de la señal HSEL. Cuando esto sucede, se revisa el tipo de operación demandada y en caso de ser de lectura, envío de datos de esclavo a master, se coloca el primer dato contenido en la cola datos_out en el bus HRDATA. Hecho esto se programa la respuesta del esclavo, OKAY, RETRY, ABORT o SPLIT, en función del estado de las colas. Finalmente se programa el nuevo estado ha ejecutar al siguiente ciclo.

Por último, en OPERACION se atiende al master recibiendo o enviando datos, según sea el caso, hasta que el decodificador desactiva HSEL. La pérdida de HSEL supone que el master ha colocado una dirección en HADDR no perteneciente al rango de direcciones del esclavo. Por tanto, la operación se da por concluida y se reporta su resultado al proceso utilitario para finalmente programar el nuevo estado a ESPERA ORDENES.

VIII. Gestión de las colas

Como se ha visto (Fig. 4), se utiliza una estructura de datos nativa a SystemC, sc_fifio, para implementar mediante colas el intercambio de datos entre los procesos del mismo módulo. Dichas colas se crean por defecto con una capacidad de 16 datos y se deben de gestionar conjuntamente por los procesos del modelo y los utilitarios del mismo.

Los procesos master y esclavo utilizan la variable datos_requeridos para notificar a los procesos utilitarios la necesidad de que estos introduzcan datos para enviar en la cola datos_out. De igual modo, mediante la variable buffer_lleno se notifica que la cola de entrada de datos, datos_in, está llena y necesita liberarse para poder seguir recibiendo información.
Esta funcionalidad esta prevista para transferencias mayores de 16 datos y, puesto que no se implementa ningún mecanismo de concurrencia sobre dichas variables, es posible que se produzcan pequeñas descoordinaciones en su utilización. Un ejemplo de descoordinación puede darse, por ejemplo, al pedir liberación de espacio para leer dos nuevos datos. Es posible que por dicha descoordinación, en vez de liberarse dos datos se liberen tres. Por tanto, el buen funcionamiento de las colas y la prevención de situaciones como la descrita es íntegramente responsabilidad del utilitario del modelo, siendo el mecanismo de las variables datos_requeridos y buffer_lleno una ayuda a tal cometido.

IX. Prueba en un código real

Una vez construido y probado [10] el correcto funcionamiento del modelo en el bus AMBA AHB cedido por el MiSE, es momento de probar su funcionamiento fuera del bus para lo cual se integra junto con un código encargado de calcular la transposición de matrices [16]. Esta es una operación sencilla, pero como el objetivo es probar que la comunicación funciona, es indiferente que sea más o menos complicada. Además, dicha prueba presenta una peculiaridad destacable ya que el código utilitario consta de dos módulos esclavo, aparte del esclavo por defecto, y uno master (Fig. 6). Así, también se probará si el sistema es capaz de adaptarse y funcionar en un bus AMBA con varios esclavos. Por tanto, se crea un nuevo proyecto, fuera del entorno ModelSim, donde se añade el código utilitario

junto con el del modelo, además de un árbitro, un decodificador y un esclavo por defecto sencillos creados específicamente para este proyecto. Todo ello conforma el sistema de la figura 6 y su funcionamiento consiste en que Matriz pide 4 matrices de 8x8 a Source, calcula su transposición y las envía a Sink quién las muestra por pantalla. Dichas matrices son inicializadas en Source con los valores consecutivos de 0 a 255, por lo que ver que la transposición y, por tanto, el sistema funcionan correcta-mente, es inmediato.

X. Conclusiones y futuro

La conclusión [17] más importante es la de que, dentro de las limitaciones impuestas, el modelo funciona. Así pues, el siguiente paso a dar es el de eliminar dichas limitaciones y utilizar el modelo para resolver problemas de cierta complejidad que requieran de una solución efectiva para sus comunicaciones. Además, también se propone una extensión de funcionalidad [18] que en teoría debería de habilitar al modelo para que, sin saltarse el estándar AMBA AHB, posibilitara que el esclavo del bus pudiera iniciar operaciones en el mismo. De hecho, solamente es una propuesta teórica que no se ha probado y que sería interesante desarrollar en un proyecto futuro.

Finalmente, el código fuente [19] queda a disposición de todo aquel que quiera utilizarlo o perfeccionarlo.

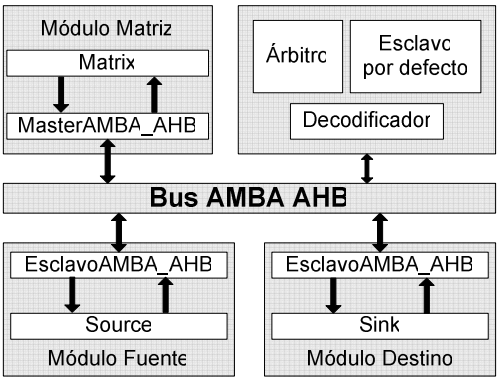

Figura 6. Sistema que calcula la transposición de matrices

Referencias

[1] www.arm.com
[2] Apéndice 4
[3] Apéndice 3
[4] Apéndice 4, apartado 4.2.
[5] Apéndice 4, apartado 4.3.
[6] Apéndice 5
[7] Apéndice 6, apartado 6.1
[8] Apéndice 6, apartado 6.2
[9] Apéndice 1, apartado 1.4.3.
[10] Apéndice 1
[11] Apéndice 1, apartado 1.5.
[12] Apéndice 1, apartado 1.2.2.
[13] Apéndice 1, apartado 1.3.2.
[14] Apéndice 1, apartado 1.2.
[15] Apéndice 1, apartado 1.3.
[16] Apéndice 2
[17] Apéndice 7, apartado 7.2.
[18] Apéndice 7, apartado 7.1.
[19] Apéndice 8

Apéndice 1. Construcción del Modelo

Comentario [L1]: Apartado Desarrollo del Proyecto en SystemC totalmente eliminado. Queda suficientemente claro en la explicación de SystemC del apéndice A

1.1. Introducción

En primer lugar, se entiende como modelo de comunicación el código que un programador podrá utilizar para realizar sus comunicaciones de forma estándar. De ahí que surja el primer inconveniente, ¿cómo integrar el código del modelo dentro del código del programador? Si se tratase de una aplicación software, se podría implementar el modelo como una o varias librerías que el programador incluiría para poder utilizar su funcionalidad, pero puesto que se trata de modelar hardware hay que buscar otros métodos.

Una buena forma de resolver dicho problema es utilizar la potencia de SystemC e implementar el código en una función que pueda ser lanzada como un hilo (thread). De ese modo, si el código de la comunicación y el del programador se implementan como dos funciones separadas para ser lanzadas como diferentes hilos, puede existir una comunicación entre ambos por medio de variables compartidas si los dos son declarados en el mismo módulo. Si a esto le sumamos la capacidad de SystemC de utilizar procesos hilados mediante el constructor SC_CTHREAD, y el hecho de que existen herramientas capaces de sintetizar código SystemC de tales características, todo parece indicar que se ha encontrado una buena solución al problema.

Una transmisión de información a través del bus AMBA AHB siempre se efectúa entre un dispositivo master y uno esclavo, siendo el master el encargado de iniciar la operación. Por ello, parece lógico que se requieran ambos dispositivos en el modelo para que pueda existir una comunicación. De ese modo, uno de los extremos comunicantes del código que implemente el programador ha de utilizar el código del módulo master, MasterAMBA_AHB, mientras que el otro ha de hacer lo propio con el código del esclavo, EsclavoAMBA_AHB. Para ello, se han de seguir ciertas pautas (Fig.1) a la hora de conformar los módulos, además de incluir el archivo constantes.h que contiene la declaración de las constantes necesarias en el código del modelo.

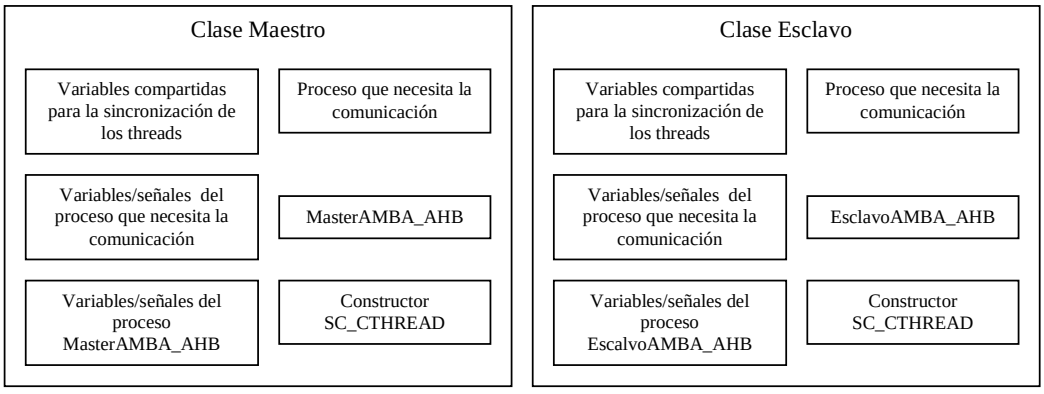

Figura 1. Estructura general de los módulos después de incluir el código del modelo de comunicación

constantes.h

```
//Ancho del bus HWDATA
#define BUS_LENGTH          32
//Dirección del Master
#define MASTER_ADRESS 0x00002000
//finaliza en 0x000020FF

//Direccion del Esclavo
#define SLAVE_ADRESS 0x00003000
//finaliza en 0x000030FF

//Maquina de estados del Master
#define RESET                   0
#define ESPERA_ORDENES          1
#define ESPERA_ARBITRO          2
#define FASE_DE_DIRECCION       3
#define FASE_DE_DATOS           4

//Maquina de estados del Esclavo
#define ESPERA_MASTER           1
#define ESCRITURA               2
#define LECTURA                 3

//Señal HSIZE (tamaño datos)
#define BYTE        0       //8 bits
#define HALFWORD    1       //16 bits
#define WORD        2       //32 bits
#define WORD2       3       //64 bits
#define WORD4       4       //128 bits
#define WORD8       5       //256 bits
#define WORD16      6       //512 bits
#define WORD32      7       //1024 bits

//Señal HRESP (respuesta esclavo)
#define OKAY        0
#define eRROR       1
#define RETRY       2
#define SPLIT       3

//Señal HBURST (tipo de ráfaga)
#define SINGLE      0
#define INCR        1
#define WRAP4       2
#define INCR4       3
#define WRAP8       4
#define INCR8       5
#define WRAP16      6
#define INCR16      7

//Señal HTRANS (tipo de operación)
#define IDLE        0
#define BUSY        1
#define NONSEQ      2
#define SEQ         3
```

1.2. Master

1.2.1. Estructura del código

El punto de partida para la implementación del módulo master del ARM AMBA AHB es conocer cómo se ha de estructurar. Dicha estructura consiste en un archivo .h donde se implementa un esqueleto para la creación del módulo y un archivo .cpp que contiene la implementación del código del master del AMBA AHB. Éste esqueleto será rellenado por los utilitarios a fin de utilizar el modelo. Véase un esquema de dichos archivos:

Archivo .h:

```
#include “systemc.h”
Inclusión de demás librerías
SC_MODULE( nombre del módulo ) //Declaración del módulo
{
        Declaración de las variables del master
        Declaración de las variables del proceso del programador
        Declaración de las variables compartidas utilizadas para la sincronización
        Declaración del reloj del sistema
        Declaración de las señales del master
        Declaración de las señales del proceso del programador
        SC_CTOR( nombre del módulo ) //Constructor
        {
                //Constructor del MasterAMBA_AHB
```

```
                SC_CTHREAD( MasterAMBA_AHB, reloj del sistema.pos() );
                        watching( señal de reset.delayed() == false );
                Resto de constructores de la parte de funcionamiento
        }
        //Declaración de la función que trabajará como master del bus
        void MasterAMBA_AHB();
        Declaración del resto de funciones del programador
};
```

Archivo .cpp

```
#include “Archivo.h”
//Inclusión de demás librerías
//Implementación del master del ARM AMBA AHB
void nombre del módulo::MasterAMBA_AHB()
{
        //Código de la función
}
//Implementación del resto de funciones del programador
void nombre del módulo::nombre de la función()
{
        //Código de la función
}
```

1.2.2. Sincronización de los hilos mediante variables compartidas

En primer lugar, véase una descripción de las variables compartidas:

Tipo	Nombre	Descripción
int	cod_operacion	Notifica el resultado obtenido en la operación demandada
bool	empieza	Indica el inicio/fin de una operación
bool	escritura	Poner a 1 para enviar datos, a 2 para recivirlos
bool	buffer_lleno	Indica que la cola de entrada de datos esta llena
bool	datos_requeridos	Notifica que en la cola de salida no hay datos para enviar
sc_bv<3>	tipo	Especifica el tipo de ráfaga con la que operar
sc_bv<2>	ancho	Especifica el ancho de palabra a utilizar
sc_bv<32>	dirección_base	Especifica la dirección inicial donde se encuentra mapeado el esclavo con el que trabajar
sc_int<BUS_LENGTH>	datos_in	Cola de datos de entrada
sc_int<BUS_LENGTH>	datos_out	Cola de datos de salida

Algunas de estas variables se utilizan para programar al proceso MasterAMBA_AHB para que ejecute una operación a través del bus, mientras que otras proporcionan al programador información del estado de la operación demandada. Así, previamente al inicio de cualquier operación, el programador ha de inicializar ciertas variables con valores concretos para especificar qué desea hacer, para después leer el valor de otras variables y poder así controlar el transcurso de la operación solicitada. Véase en un ejemplo:

Supóngase que se desea enviar una ráfaga de 5 datos a un esclavo, para lo cual, primeramente hay que especificar qué tipo de operación se va a realizar. Necesariamente dicha operación será de escritura pues se desea emitir datos, por tanto, el primer paso es colocar la variable **escritura** a cierto. Si por el contrario se quisiera

leer datos, bastaría con inicializar **escritura** a falso. Después, se ha de especificar de qué manera se pretende enviar los datos puesto que hay varias maneras de enviar 5 datos. La forma más rápida y directa es programando una única ráfaga de 5 datos que los envíe todos en una misma operación. Para ello, inicializa la variable **tipo** a INCR y coloca 5 en **num_datos_incr**.

Hecho esto, se ha de indicar a qué dirección de memoria se va a enviar la información, para lo cual se utiliza **direccion_base**. En este caso, la dirección base ha de corresponder con la dirección del esclavo del AMBA, EsclavoAMBA_AHB, que es quién atenderá la petición. La dirección del mismo la fija el decodificador del bus que en este caso utiliza la 0x00003000 que es referenciada como SLAVE_ADRESS en el archivo "contantes.h".

El siguiente paso es el de especificar el tamaño de los datos [1] a enviar (tamaño de palabra). Por ejemplo, si se desea enviar datos formados por 8 bits (1 byte), se ha de asignar a la variable **ancho** la constante byte.

Finalmente se han de suministrar los datos a enviar, operación que se realiza a través de una cola fifo, **datos_out** en este caso. Para añadir un dato a una cola se ha de utilizar el método write() que proporciona la estructura de datos sc_fifo, por lo que se ejecuta la instrucción datos_out.write(dato a enviar).

Terminada la programación de señales, se da comienzo a la operación activando a valor cierto la señal **empieza.** Véanse los comandos que realizan todo el proceso descrito:

```
escritura = cierto;
tipo = INCR;
num_datos_incr = 5;
direccion_base = SLAVE_ADRESS;
datos_out.write( "dato a enviar" );
empieza = true;
```

Llegado a este punto, se inicia automáticamente la comunicación por el bus AMBA y, aunque todo el proceso queda en manos del código del modelo, es importante no descuidar las colas, para lo cual se han de utilizar las variables de información de estado.

Puesto que únicamente se ha colocado un dato en la cola de salida y sin embargo se ha programado una transferencia de 5 datos, en el momento en el que se inicie el envío la variable **datos_requeridos** será activada a cierto por el código del modelo. Por tanto, al ver este cambio en la variable se deberían de colocar datos en la cola de salida **datos_out**. De igual modo, si se tratase de una transferencia de lectura y la cola de entrada de datos, **datos_in**, estuviera llena, se activaría la variable **buffer_lleno** a cierto notificando tal evento. Se debería entonces vaciar dicha cola mediante la instrucción datos_in.read() que extrae un dato de la cola cada vez que se ejecuta.

Datos_requeridos y buffer_lleno son un mecanismo de ayuda para gestionar las colas de datos, pero es el utilitario del modelo quién realmente ha de controlar su estado en todo momento, aparte de comprobar el valor de **cod_operacion** tras recibir la notificación del fin de la operación, variable **empieza** a valor falso.
A continuación, véanse las variables para la sincronización con más detalle:

Variable cod_operación:

Código	Significado
0	La operación ha concluido satisfactoriamente.
1	Error en la programación del master. El tipo de ráfaga proporcionado es incorrecto.
2	Error en la programación del master. El ancho de palabra proporcionado es incorrecto.

Variable empieza:

Valor	Significado
Cierto	Lo escribe el código del programador para dar comienzo a la operación.
Falso	Lo escribe el código del modelo para notificar que ha finalizado la operación demandada.

Variable escritura:

Valor	Significado
Cierto	Programa una operación de escritura que envía datos del Master al Esclavo del bus.
Falso	Programa una operación de lectura donde el Master recibe datos del Esclavo.

Variable tipo:

Valor	Constante	Significado
0	SINGLE	Operación de ráfaga unaria. Se lee/escribe un único dato.
1	INCR	Operación de ráfaga incremental de duración no especificada. Puede ser todo lo grande que se quiera siempre y cuando no se exceda de 1 Kbyte en el rango de direcciones.
3	INCR4	Operación de ráfaga incremental de 4 datos.
5	INCR8	Operación de ráfaga incremental de 8 datos.
7	INCR16	Operación de ráfaga incremental de 16 datos.

Variable ancho*:

Valor	Constante	Significado
0	BYTE	Ancho de palabra de 8 bits.
1	HALFWORD	Ancho de palabra de 16 bits.
2	WORD	Ancho de palabra de 32 bits.
3	WORD2	Ancho de palabra de 64 bits.
4	WORD4	Ancho de palabra de 128 bits.
5	WORD8	Ancho de palabra de 256 bits.
6	WORD16	Ancho de palabra de 512 bits.
7	WORD32	Ancho de palabra de 1024 bits.

*El ancho de palabra máximo depende de la constante BUS_LENGTH definida en “constantes.h” y que por defecto tiene el valor WORD, 32 bits.

Variable buffer_lleno:

Valor	Significado
Cierto	La cola datos_in está llena y necesita ser vaciada para poder alojar nuevos datos.
Falso	Es responsabilidad del proceso del programador poner el valor falso una vez vaciada la cola.

Variable datos_requeridos:

Valor	Significado
Cierto	La cola datos_out está vacía y necesita que se depositen nuevos datos que poder enviar.
Falso	Es responsabilidad del proceso del programador poner el valor falso una vez puesto el dato.

1.2.3. Implementación del MasterAMBA_AHB

El master del bus ARM AMBA AHB es siempre el encargado de iniciar cualquier tipo de operación que se realice a través del bus. Para ello, lo primero que ha de hacer es pedir el bus al árbitro y esperar hasta obtenerlo. Cuando el árbitro lo concede, el siguiente paso es el de iniciar la operación que, como se ha visto en capítulos anteriores, se compone de dos fases, la fase de dirección y la de datos. Como se ve, toda esta funcionalidad es fácilmente separable en diferentes partes, más cuando se va a repetir cíclicamente en cada nueva operación con el bus. Por ello, resulta mucho más fácil abordar el código del master si éste se implementa como una máquina de estados (Fig. 2) donde cada operación separable configura un estado diferente y donde los valores de las flechas corresponden a las condiciones de transición entre estados.

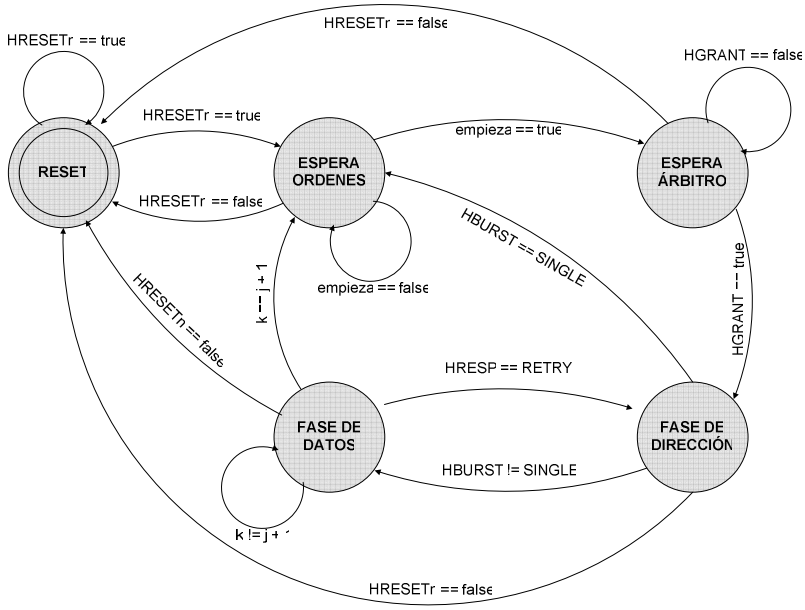

Figura 2. Máquina de estados del MasterAMBA_AHB

Esta máquina de estados es fácilmente representable en SystemC utilizando una estructura del tipo “switch” donde en cada posibilidad o caso “case” se implementa un estado distinto. Luego el esqueleto del código incluido en el fichero .cpp y perteneciente a la función MasterAMBA_AHB presenta la siguiente estructura Donde estado es una variable de tipo entero y los distintos casos son constantes numéricas definidas en “constantes.h”:

```
switch( estado )
{
        case RESET:                     /*código*/      break;
        case ESPERA_ORDENES:            /*código*/      break;
        case ESPERA_ARBITRO:            /*código*/      break;
        case FASE_DE_DIRECCION:         /*código*/      break;
        case FASE_DE_DATOS:             /*código*/      //break no necesario
}
```

Antes de ver detalladamente cada uno de estos estados, se presenta una tabla con una breve descripción de todas las variables locales al proceso MasterAMBA_AHB.

Tipo	Nombre	Descripción
int	k	Contador utilizado para conocer la fase en la que se encuentra la operación
int	j	Contador de datos a enviar/recibir
int	estado	Contiene el estado de la máquina de estados a ejecutar
int	incr_direccion	Se programa con el valor del incremento a aplicar a la dirección base
bool	retry	Se activa si se produce una respuesta retry por parte de un esclavo
bool	leido	Se pone a cierto si se ha podido leer un dato y a falso si no
bool	enviado	Se pone a cierto si se ha podido enviar un dato y a falso si no.
sc_int<*>	datos_out_tmp	Registro temporal donde se alojan los datos extraídos de la cola de salida antes de ser enviados por el bus AMBA.

*El tamaño de la variable viene dado por la constante BUS_LENGTH definida en "constantes.h" y que por defecto vale 32.

A continuación se explicará detalladamente el código fuente del master del bus [2].

RESET

En todo circuito electrónico es necesario incluir una señal de reset que permita inicializar los distintos componentes del circuito a un estado conocido. Ésta es la función que desempeña el estado de RESET, por lo que siempre que se active la señal HRESETn se volverá a este estado para reiniciar el proceso.
Concretamente la inicialización requerida por el master del bus se reduce a inicializar las señales de control y de dirección a estados conocidos según especifica el protocolo del ARM AMBA AHB [1], además de la programación del siguiente estado. Hecho esto, solo queda esperar hasta que la señal de reset se desactive y se pueda seguir con la ejecución del resto del código. De manera que el estado de RESET presenta el siguiente aspecto:

case RESET:
    Inicializa señales de control y dirección a valores válidos
    Programa el siguiente estado
    Espera hasta fin de reset --> HRESETn = 1
break;

ESPERA_ORDENES

Una vez inicializadas las señales en el RESET, se programa automáticamente para que se entre en este estado. Aquí el master queda a la espera de que se le demanden operaciones de transferencia de datos por el bus, para lo cual se utilizan las variables compartidas para la sincronización como ya se ha explicado anteriormente. De modo que el master cada ciclo consulta estas señales de sincronización en busca de cambios de valor que indiquen una petición de operación. Si no se produce tal cambio, se ha de continuar esperando. Véase el esquema de funcionamiento:

case ESPERA_ORDENES
    Si empieza operación:
        Comprobación de las variables para la sincronización
        Inicialización de las variables locales y de sincronización
        Petición del bus
        Programación del siguiente estado
    Si no:
        Espera
    Fin Si
break;

La comprobación de las variables para la sincronización consiste en aseverar que éstas han sido correctamente programadas acorde con el protocolo AMBA AHB [1]. Para ello, primero se comprueba que el tipo de ráfaga escogida sea SINGLE, INCR, INCR4, INCR8 o INCR16, notificándose error en cualquier otro caso y abortando la operación. Así mismo, también se aprovecha para inicializar la variable j con el valor del número de datos a enviar/recibir como muestra la siguiente tabla:

Operación	Valor de j
SINGLE	1
INCR	*
INCR4	4
INCR8	8

*Como INCR es una operación de ráfaga de tamaño indeterminado, el valor de j se ha de especificar explícitamente a través de la variable num_datos_incr

También es necesario comprobar que el ancho de palabra especificado concuerde con alguno de los valores permitidos. Por ello, si éste es distinto a HALFWORD o a WORDx, donde x puede ser 2, 4, 8, 16 o 32, se notifica el error y se aborta la operación.

Finalmente, se inicializan las variables locales restantes, k y retry, con los valores 0 y falso respectivamente. Luego se pide el bus activando la señal HBUSREQ a cierto y se programa el siguiente estado a ejecutar, ESPERA_ARBITRO.

ESPERA_ARBITRO

Se llega a este estado cada vez que se hace una petición de bus y hay que esperar hasta que el árbitro lo conceda. Por tanto, lo único que hay que hacer es consultar HGRANT cada ciclo de reloj para ver si ya se ha concedido el bus o si aún hay que esperar un poco más. Cuando se consigue el bus, se restablece HBUSREQ a falso y se programa el nuevo estado para continuar con la operación.

case ESPERA_ARBITRO:
    Si árbitro concede el bus:
        Deja de pedir el bus pues ya lo tiene
        Programa el nuevo estado
    Si no
        Espera
    Fin Si
break;

FASE_DE_DIRECCION

Una vez conseguido el bus, el maestro ya esta habilitado para iniciar la transferencia de datos que se inicia con la fase de dirección. Por tanto, el cometido de este estado es el de inicializar las señales de control y de dirección del primer dato a enviar/recibir., tal y como se muestra a continuación:

```
//Control
HSIZE.write( ancho );
HBURST.write( tipo );
HTRANS.write( NONSEQ );
HWRITE.write( escritura );
//Dirección
HADDR.write( direccion_base.to_int() + (incr_direccion * k) );
k++;
```

Nótese cómo se asigna la dirección. Se coloca la dirección base del esclavo con el que operar y se le suma el incremento correspondiente. Este incremento se calcula multiplicando el incremento básico por la variable k que inicialmente vale 0 y que se incrementa en uno cada vez que se asigna una nueva dirección. Véase el siguiente ejemplo para aclarar dudas:

```
direccion_base = 0;
incr_direccion = 2;
```

Valor de k:	0	1	2	3
Dirección en HADDR:	0	2	4	6

Esquema del funcionamiento del estado:

```
case FASE_DE_DIRECCION:
        Programa señales de control
        Programa señales de dirección
        Programa el siguiente estado
break;
```

FASE_DE_DATOS

Éste es sin duda el estado más complejo de toda la máquina de estados del proceso master. Dentro del bus AMBA AHB, en las transferencias de ráfagas mayores de un dato, la fase de dirección y la fase de datos se solapan de manera que en el mismo ciclo se ha de colocar el dato x y la dirección del dato x+1. Además de esto, la fase de datos ha de comprobar en que punto de la transferencia se encuentra en cada momento, así como la respuesta que el esclavo emite para poder actuar en consecuencia. Para ello, se ha de tener muy presente el funcionamiento de los procesos SC_CTHREAD [3] donde los cambios efectuados en una señal no se reflejan hasta el ciclo siguiente. Finalmente, para facilitar la comprensión del funcionamiento véase el siguiente ejemplo :

En la figura 3, inicialmente k adquiere el valor 0 y luego se incrementa en uno cada vez que se coloca una nueva dirección en el bus de direcciones HADDR. De ese modo, la primera vez que se entra en la fase de datos k vale 1, la segunda 2 y la tercera y última 3. Teniendo en cuenta estos valores, nótese que cuando k = 1, no se comprueba la respuesta del esclavo ni se ha de leer ningún dato, pero si que se ha de enviar. Para k = 2, tanto se ha de leer como enviar información, así como comprobar la respuesta del esclavo. Finalmente para k = 3, se ha de leer y comprobar la respuesta, pero no enviar.

De igual modo, nótese que solamente se ha asignado un nuevo valor a las señales de control y de dirección cuando k ha valido 1. Sin embargo, si la ráfaga hubiera sido de 3 datos en vez de 2, se debería de haber asignado nuevos valores de control y de dirección para k = 1 y para k = 2.

Por último y siguiendo por esta misma línea, en una operación INCR de 2 datos, j adquiere el valor 2, mientras que en una de 3, j vale 3.

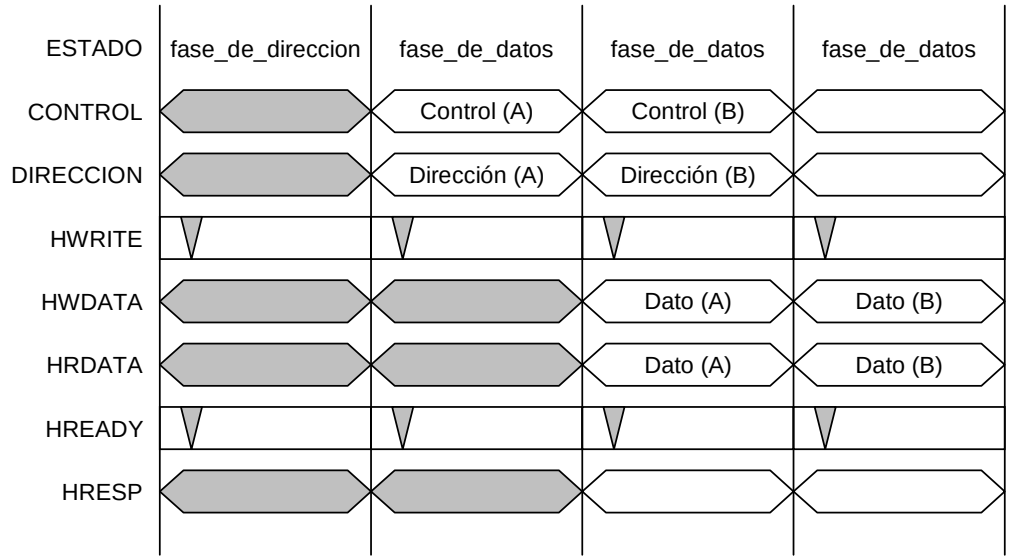

Figura 3. Diagrama de formas de onda de una transferencia INCR de dos.

En conclusión y razonando sobre estas observaciones, se puede extrapolar éste funcionamiento para cualquier tamaño de ráfaga del siguiente modo:

- Si k = 1 → Se escriben datos pero no se leen. Se programan las señales de control y de dirección normalmente. No se comprueba la respuesta del esclavo.
- Si k = j → Se leen y escriben datos, se comprueba la respuesta, pero no se programan las señales de control y de dirección normalmente, sino que se inicializan con los valores por defecto para ir al estado ESPERA_ORDENES.
- Si k = j+1 → Se leen datos pero no se escriben, se comprueba la respuesta del esclavo pero no se tocan las señales de control y de dirección.
- Si k > 1 y k < j → Se leen y escriben datos, se comprueba la respuesta del esclavo y se programan las señales de control y de dirección.

Al margen de esto, se ha visto que en las transferencias de ráfagas mayores de un dato la fase de datos se solapa con la fase de dirección de forma que ambas se ejecutan en el mismo ciclo de reloj. Por ello, una forma razonable de estructurar el código del presente estado es separándolo en éstas dos fases de forma lógica y obteniendo el siguiente esquema de funcionamiento:

```
case FASE_DE_DATOS:
        /***Fase de datos***/
        Si operación de escritura:
                Analizar si se escribe el dato o no
        Si no:
                Analizar si se lee el dato o no
        Fin Si
        /***Fase de dirección***/
        Comprobar la respuesta del esclavo
        Programar las señales de control y de dirección para el dato siguiente
break;
```

Para el análisis de si se ha de escribir/leer un dato o no, se han de tener en cuenta el valor de k y la respuesta del esclavo ya que si éste ha insertado un ciclo de espera, no se han de enviar/recibir datos hasta que HREADY vuelva a valer cierto. De hecho, solamente se escriban datos si se cumple la condición:

Si (HREADY = cierto && HRESP = OKAY && k ≠ j+1) || k = 1

En caso de cumplirse dicha condición, se escribirá el dato si existen datos disponibles para ser enviados y se activará la variable enviado a cierto. En caso contrario, enviado tomará el valor falso.
Respecto a la lectura, la condición es la siguiente:

Si k ≠ 1 && HREADY = cierto && HRESP = OKAY

Una vez cumplida la condición, se leerán los datos solamente si hay espacio disponible para almacenarlos. De hecho, este punto es más crítico que la no existencia de datos que enviar, ya que el no poder almacenar los datos recibidos implica la pérdida de los mismos debido a que realmente no hay implementado un mapeo entre una dirección y un dato concreto. El mecanismo de funcionamiento del presente modelo de comunicación no asocia direcciones con datos, sino que sencillamente pide datos a un esclavo que envía la información que le han proporcionado a través de una cola fifo. Por tanto cuando el esclavo consume un dato de la fifo, este ya no es accesible para él por lo que no se puede reenviar a menos que sea el proceso utilitario del esclavo quién lo coloque de nuevo en la fifo.

En conclusión, queda claro que es muy importante hacer una buena gestión de las colas para una correcta comunicación. Véase como se realiza dicha gestión en el caso de la cola datos_in:

Si capacidad > 1 dato
    Solo se lee el dato si en el ciclo anterior HTRANS no valió BUSY
Sino Si capacidad = 1 dato
    Si en el ciclo anterior HTRANS no valió BUSY
        Lee el dato y programa BUSY para el siguiente ciclo
    Fin Si
Fin Si

Terminada la fase lógica de datos, se prosigue con la de dirección. La comprobación de la respuesta del esclavo y el conocimiento del estado de las colas determinan qué valores asignar a las señales de control para retener la operación o continuar normalmente. Véase el siguiente esquema de funcionamiento:

Si HREADY = cierto
    Si HRESP = OKAY || k = 1
        Si leído y enviado son ciertos
            Programa señales de control y de dirección dependiendo del valor de k
        Si No
            Programa HTRANS = BUSY para retener al esclavo
        Fin Si
    Si No
        Trata los casos HRESP igual a ERROR, RETRY y SPLIT
    Fin Si
Si No
    Si HRESP diferente de OKAY

```
                Programa HTRANS = IDLE y espera a resolver en la siguiente fase de la respuesta
        Fin Si
Fin Si
```

Así, si la respuesta del esclavo es positiva o se trata de la primera vez que se inicia la fase, se pasa a mirar las variables leído y enviado con las que se conocerá la situación de las colas. Las operaciones a efectuar durante toda la transferencia con un esclavo serán siempre o de lectura o de escritura y sin posibilidad de alternancia entre ellas. Por ello, en el estado FASE_DE_DIRECCION se inicializan ambas variables a cierto, de manera que si es una operación de lectura, enviado siempre valdrá cierto y entrar o no en la condición dependerá exclusivamente de la variable leído. De ese modo, cuando una de las dos valga falso se deberá programar HTRANS = BUSY para retener la operación hasta que ambas valgan cierto.

Si por el contrario HRESP vale ERROR, SPLIT o RETRY, se trata de la segunda fase de estas respuestas ya que HREADY es cierto. Por lo tanto, en los dos primeros casos se finaliza la operación devolviendo el código de operación 3 a través de la variable compartida cod_operacion. En cambio, para el caso retry se activa la variable local retry a cierto y se programa el nuevo estado a FASE_DE_DIRECCION para reiniciar la transferencia actual. No se ha de temer la pérdida de datos pues la respuesta retry únicamente se activa por un esclavo al inicio de una operación, nunca entre medio o al final. Por ello, MasterAMBA_AHB utiliza la variable datos_out_tmp a modo de registro temporal donde almacena el último dato enviado. Así si hay un retry, la próxima vez que se envíe un dato se hará desde este registro en vez de desde la cola, con lo que no se perderá ningún dato enviado.

Finalmente, si HREADY tiene el valor falso puede indicar que el esclavo está insertando un ciclo de espera si HRESP = OKAY. Si por el contrario HRESP vale ERROR, SPLIT o RETRY, el esclavo está emitiendo la primera fase de una de estas respuestas. En el primer caso no se hace absolutamente nada, se espera hasta que el esclavo vuelva a poder trabajar con normalidad. En cambio, para los demás casos se programa el estado del master, señal HTRANS, a IDLE y se espera a la segunda fase de dichas operaciones para resolver cada caso. La segunda fase se notifica con la subida de HREADY a cierto y se resuelve como ya se ha explicado.

1.3. Esclavo

1.3.1. Estructura del código

Habiendo visto el código del master, entender el código del esclavo es sencillo. Para empezar, el esqueleto de la definición del módulo contenido en el archivo .h es un calco del esqueleto del master:

```
#include “systemc.h”
Inclusión de demás librerías
SC_MODULE( nombre del módulo ) //Declaración del módulo
{
        Declaración de las variables del esclavo
        Declaración de las variables del proceso del programador
        Declaración de las variables compartidas utilizadas para la sincronización
```

```
        Declaración del reloj del sistema
        Declaración de las señales del master
        Declaración de las señales del proceso del programador
        SC_CTOR( nombre del módulo ) //Constructor
        {
                //Constructor del EsclavoAMBA_AHB
                SC_CTHREAD( EsclavoAMBA_AHB, reloj del sistema.pos() );
                        watching( señal de reset.delayed() == false );
                Resto de constructores de la parte de funcionamiento
        }
        //Declaración de la función que trabajará como esclavo del bus
        void EsclavoAMBA_AHB();
        Declaración del resto de funciones del programador
};
```

Análogamente, el archivo .cpp también tiene una estructura prácticamente idéntica al del módulo master.

```
#include "Archivo.h"
Inclusión de demás librerías
//Implementación del esclavo del ARM AMBA AHB
void nombre del módulo::EsclavoAMBA_AHB()
{
        Código de la función
}
//Implementación del resto de funciones del programador
void nombre del módulo::nombre de la función()
{
        Código de la función
}
```

1.3.2. Sincronización de los hilos mediante variables compartidas

El funcionamiento de la sincronización del esclavo se parece mucho a la del master, salvo porque es más fácil. En este caso no se ha de programar prácticamente nada, si no que sencillamente se ha de estar pendiente de las variables porque en cualquier momento puede haber una notificación de inicio o de final de operación. Para ello, este módulo presenta las siguientes variables de sincronización:

Tipo	Nombre	Descripción
int	cod_operacion	Notifica el resultado obtenido en la operación completada.
bool	Inicio_operacion	Indica el inicio de una operación.
bool	Fin_operacion	Indica el final de una operación.
bool	buffer_lleno	Indica que la cola de entrada de datos esta llena
bool	Datos_requeridos	Notifica que en la cola de salida no hay datos para enviar
bool	No_datos	Permite programar el no envío de datos ante cualquier solicitud.
sc_int<BUS_LENGTH>	Datos_in	Cola de datos de entrada
sc_int<BUS_LENGTH>	Datos_out	Cola de datos de salida

Inicialmente no hay más que asignar el valor a no_datos y esperar hasta que se active **inicio_operacion**. Cuando esto suceda, hay que intervenir en la gestión de las colas para lo cual se puede utiliza la ayuda de las variables **buffer_lleno** y **datos_requerido,** tal y como se ha visto en el caso del master. Ha diferencia de éste, el esclavo esta habilitado para finalizar una operación en el caso de no disponer de información que enviar o de capacidad para almacenar los datos entrantes. Si transcurren 16 ciclos de reloj y el

código del programador no ha atendido una petición de datos requerido o de buffer lleno, el esclavo abortará la operación mediante la respuesta ERROR, activará **fin_operacion** y retornará el código 3 en **cod_operacion**. Con esta medida se pretende evitar una situación de *"dead lock"* donde el Esclavo detiene indefinidamente al master mediante HREADY = cierto y HRESP = falso mientras espera obtener datos que enviar u espacio donde alojar la información entrante.

Véanse a continuación las variables para la sincronización al detalle:

Variable cod_operacion:

Código	Significado
0	Sin valor. Ninguna operación se ha finalizado por el momento.
1	Operación de escritura (Master envía datos a Esclavo) concluida correctamente.
2	Operación de lectura (Esclavo envía datos a Master) concluida correctamente.

Variable inicio_operacion:

Valor	Significado
Cierto	Notifica que se ha iniciado una operación que está por concluir.
Falso	Indica que no hay ninguna operación en curso.

Variable fin_operacion:

Valor	Significado
Cierto	Indica que se ha finalizado una operación.
Falso	Indica que no hay ninguna operación finalizada.

Variable no_datos:

Valor	Significado
Cierto	Programa al EsclavoAMBA_AHB para que aborte todas las operaciones entrantes donde el Master le pida datos.
Falso	Habilita al EsclavoAMB_AHB para que atienda cualquier operación entrante donde el Master le pida datos.

Variable buffer_lleno:

Valor	Significado
Cierto	La cola datos_in está llena y necesita ser vaciada para poder alojar nuevos datos.
Falso	Es responsabilidad del proceso del programador poner el valor falso una vez vaciada la cola.

Variable datos_requeridos:

Valor	Significado
Cierto	La cola datos_out está vacía y necesita que se depositen nuevos datos que poder enviar.
Falso	Es responsabilidad del proceso del programador poner el valor falso una vez puesto el dato.

1.3.3. Implementación del EsclavoAMBA_AHB

De igual modo que el master, la funcionalidad que ha de desempeñar el esclavo es fácilmente diferenciable en distintas partes, por lo que la implementación de una máquina de estados supone la mejor opción. Además, esta máquina de estados es más sencilla y contiene menos estados que la del master, lo que aún simplifica más la tarea. Por otro lado, ahora no existe un estado donde la máquina quede parada a la espera de órdenes del módulo del programador, sino que el movimiento entre los distintos estados depende exclusivamente del comportamiento del bus ARM AMBA AHB.

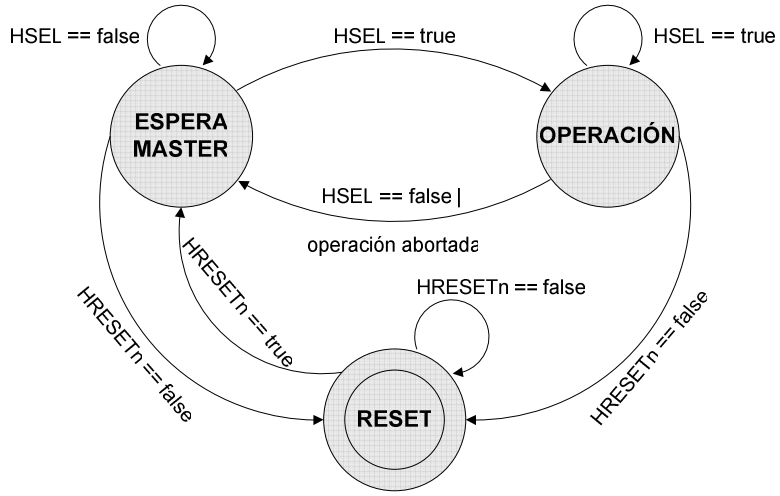

Figura 4. Máquina de estados del EsclavoAMBA_AHB.

Esta máquina de estados (Fig. 4) cuyos valores en las flechas corresponden a las condiciones de transición entre estados, se implementa en una estructura switch dentro del código de la función EsclavoAMBA_AHB en el fichero .cpp. De modo que su esqueleto es el siguiente:

```
switch( estado )
{
        case RESET:                 /*código*/      break;
        case ESPERA_MASTER:         /*código*/      break;
        case OPERACIÓN              /*código*/      //break no necesario
}
```

A continuación se detalla el código fuente del EsclavoAMBA_AHB [4].

RESET

El funcionamiento de este estado es parejo al del módulo master ya que en el se inicializan las señales del bus acorde al protocolo AMBA AHB [1]. Además, RESET también inicializa las variables locales al proceso y las variables utilizadas para su sincronización con los procesos del programador. Después programa el siguiente estado y espera hasta la finalización de la señal de reset, HRESETn = 1. Véase su esquema:

```
case RESET:
        Inicializa señales básicas esclavo según protocolo AMBA
        Inicializa variables locales
        Inicializa variables de comunicación entre los procesos
        Programa el nuevo estado
        Espera hasta finalizar reset --> HRESETn == 1
break;
```

ESPERA_MASTER

En este estado el esclavo permanece a la espera de ser activado por el decodificador, lo que indica que hay algún master del bus que desea trabajar con él. Su esquema de funcionamiento es el siguiente:

```
Si HSEL = cierto
        Si operación de escritura
                Tratamiento de la petición de escritura
        Si no
                Tratamiento de la petición de lectura
        Fin Si
        Programa el nuevo estado
Si no
        Espera hasta ser activado
Fin Si
```

Una vez que se activa al esclavo, éste consulta el tipo de operación a realizar y revisa el estado de la correspondiente cola. Si se trata de una operación de lectura y dispone de información que enviar, pone el primer dato en el bus HRDATA y emite una respuesta OKAY. Si por el contrario no dispone de datos, operación de lectura, o de capacidad de almacenamiento, operación de escritura, programa un ciclo de espera e inicia la petición de datos o de espacio según el caso. En cualquier caso, programa el siguiente estado antes de acabar.

OPERACION

Este estado implementa el código de atención a la petición del master que, a grandes rasgos, consiste en enviar o recibir datos según sea el caso tal y como se aprecia en su esquema de funcionamiento:

```
Si Esclavo seleccionado
        Si HTRANS ≠ BUSY
                Si no_datos || se ha retenido al master durante 16 ciclos
                        Aborta la operación y retorna cod_error = 3
                Fin Si
                Si operación de escritura
                        En función del estado de la cola, lee o no el dato y programa la respuesta
                Si no
                        En función del estado de la cola, envía o no el dato y programa la respuesta .
                Fin Si
        Si no
                No hace nada, solo espera
        Fin Si
Si no
        Si operación de escritura
                Lee el último dato
        Fin Si
        Activa fin_operacion, desactiva inicio_operacion y programa cod_operacion y el nuevo estado
Fin Si
```

Si el master no ha pedido un ciclo de espera, se pasa a mirar el tipo de operación demandada y el estado de la cola asociada. Si la operación es de lectura y se dispone de datos que enviar, se escribe el dato en HRDATA y se programa la respuesta OKAY. Si por el contrario no se dispone de datos que emitir, se programa un ciclo de espera.
Para la operación de escritura, se lee el dato pues en el ciclo anterior ya se aseguro de tener capacidad para almacenar la información entrante. Luego mira si tendrá sitio para el dato que se recibirá al ciclo siguiente y programa la respuesta en función de dicha capacidad. Contrariamente, si el master ha programado un ciclo de espera, no se realiza ninguna acción.

En el caso de no ser seleccionado por el decodificador, se interpreta que la operación se encuentra en su último ciclo, k = j+1, por lo que en caso de escritura se lee el dato entrante a menos que en el anterior ciclo se programase una pausa. En tal caso, el master leerá dicha pausa en el presente ciclo y volverá a programar la dirección del esclavo al ciclo siguiente para concluir la operación. Finalmente, a menos que se de la situación descrita, se programa el nuevo estado y las variables de sincronización con las que notificar el final de la operación y el resultado de la misma.

1.4. Preparación de la simulación en ModelSim

1.4.1 Introducción

Hasta aquí se ha visto cómo se construyen los dos módulos principales del modelo, pero para tener la certeza de que estos funcionan acorde al protocolo ARM AMBA AHB, se tienen que simular. Para la simulación, se necesita el bus AMBA AHB al completo y éste está disponible en Verilog. Por lo tanto, es necesario utilizar el software ModelSim [5] para poder insertar los módulos desarrollados en SystemC en el bus escrito en Verilog y realizar la simulación. Además, también es necesario añadir los procesos utilitarios del modelo de comunicación para que realicen alguna tarea donde se vea implicado el bus.

Para ello, primeramente se han de implementar dos procesos utilitarios, maestro y esclavo, que sirvan para probar la funcionalidad del modelo mediante el uso de este. Seguidamente, se ha de terminar de acoplar bien ambos lenguajes, para lo cual es necesario escribir e incluir algunos archivos extra que permitan estimular las ondas básicas, como el reloj y el reset, de todo el circuito.

Terminado con esto, falta crear un nuevo proyecto en ModelSim, añadir todo el código y crear un archivo .do que no es más que un script con los comandos necesarios para compilar, simular y ver las formas de onda obtenidas.
Al final de todo este proceso, solo queda empezar las simulaciones e ir ajustando el sistema hasta que funcione correctamente, para lo cual se modifica varias veces el código de los procesos utilitarios con el objetivo de probar toda la funcionalidad del bus.

1.4.2. Descripción de los procesos utilitarios

Con el objetivo de probar el código, se añade el proceso maestro a la estructura del módulo master con la finalidad de que maestro actúe como utilitario de ésta parte del modelo. Véase su algoritmo:

Proceso maestro
    Inicializa datos
    Programa tipo de ráfaga
    Programa tipo de operación
    Inicia operación
    Siempre
        Si Operación terminada
            Sal de Siempre
        Si no

```
                    Si datos_requeridos
                        Atiende a la petición de datos para enviar
                    Fin Si
                    Si buffer_lleno
                        Atiende a la petición de liberación de espacio
                    Fin Si
                    Espera
                Fin Si
            Fin Siempre
            Comprobar valor cod_operacion
Fin Proceso
```

El funcionamiento de maestro se limita a programar los datos a enviar y el tipo de operación para luego iniciarla. Mientras la operación se está ejecutando, comprueba a cada ciclo las variables de sincronización entre procesos por si ha de realizar alguna acción en las colas fifo datos_in y datos_out. Después, al finalizar la operación, comprueba qué tal ha ido consultando la variable cod_operación. Todos estos pasos se han ido salvando en un fichero de texto para su posterior revisión y verificar así el correcto funcionamiento del modelo.

Utilizando éste algoritmo como base, se pueden hacen cambios en el tipo de operación a realizar y en el estado de las colas para probar toda la funcionalidad del master del bus.

Respecto a la parte del esclavo del bus, para probar su funcionamiento se añade el proceso esclavo exactamente igual a como se ha hecho en el caso del master. En este caso, su algoritmo es el siguiente:

```
Proceso esclavo
        Prepara datos para enviar
        Programa variable no_datos
        Siempre
            Si inicio_operacion
                Si datos_requeridos
                    Atiende a la petición de datos para enviar
                Fin Si
                Si buffer_lleno
                    Atiende a la petición de liberación de espacio
                Fin Si
            Fin Si
            Si fin_operacion
                Consulta cod_operacion y actúa en consecuencia
            Fin Si
        Fin Siempre
Fin Proceso
```

En este caso, el único juego que ofrece esclavo es el de modificar el estado de las colas y la variable no_datos. Para probar las respuestas retry, error y split, se deberá de modificar el código de EsclavoAMBA_AHB.

1.4.3. Acoplamiento de SystemC y Verilog

Una vez que todo el código operativo está listo, solo queda implantarlo en el bus. Para ello se hace necesario incluir nuevos módulos en la parte superior de la jerarquía del bus [6] para que estimulen las señales básicas del sistema y pueda de ese modo iniciarse la

simulación. Así pues, la jerarquía del bus ha de modificarse (Fig. 5) para que ambos lenguajes, Verilog y SystemC, puedan simularse conjuntamente.

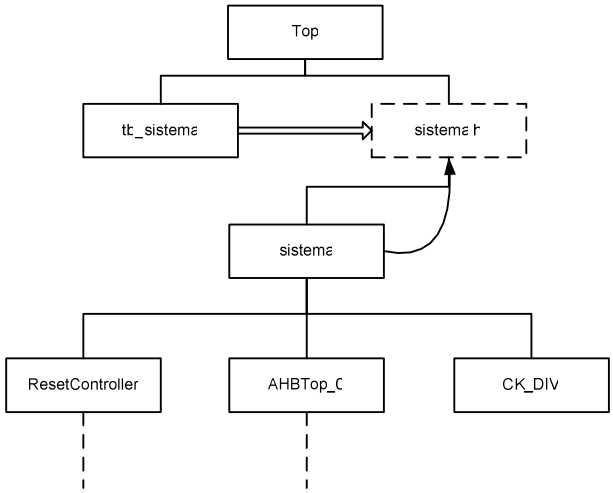

Figura 5. Jerarquía de módulos final después de añadir el código SystemC al bus Verilog.

De ese modo se ve como sistema ya no se encuentra en la parte más alta de la jerarquía si no que ésta pasa a estar ocupada por Top, quien instancia a sistema.h y a tb_sistema. El funcionamiento es el siguiente:

- ModelSim genera automáticamente sistema.h a partir de sistema.v.
- Top instancia a tb_sistema y a sistema.
- tb_sistema estimula las señales de entrada, reloj y reset, de sistema.
- sistema.h instancia a sistema.v y le pasa los estímulos que recibe de tb_sistema.
- sistema.v instancia el resto de componentes del bus donde ahora están AHB_DummyMaster y AHB_DummySlave escritos en SystemC.

A continuación se repasará de forma más detallada cada uno de estos nuevos módulos.

Top

Es el elemento situado más arriba en la jerarquía del sistema y se encarga de instanciar a sistema y a tb_sistema, un *"test bench"* que estimulará las señales de entrada del módulo sistema. Para ello, inicialmente crea un módulo, tbtop, donde declara las señales que se vincularán a tb_sistema e instancia éste módulo asignándole dichas señales. Luego se crea un nuevo módulo, top, donde se declaran las señales correspondientes a el módulo sistema para acto seguido instanciarlo y vincularle dichas señales. Así mismo, también en top se instancia a tbtop y se le vinculan sus señales. De ese modo y en resumen, se obtiene una instancia de tb_sistema y de sistema, luego se les asignan sus señales de entrada y de salida de forma que las que les son comunes a ambos, corresponden a la misma señal. Por tanto, tb_sistema y sistema quedan comunicados con lo que tb_sistema puede estimular las señales de sistema tal y como se pretendía.

Además, en top se crea el reloj del sistema, "sc_clock clk;", y se pone en funcionamiento, "clk("clk",10, 0.5, 0.0)". El código de Top.h es el siguiente:

```
#ifndef TOP_H
#define TOP_H

#include "systemc.h"
#include "sistema.h"
#include "tb_sistema.h"

//Definicio del testBench
SC_MODULE(tbtop)
{
     sc_in<bool>       CK_IN;
     sc_out<bool>      ResetIN_OBM_0;
     sc_out<bool>      HRESETn_OBM_1;

     tb_sistema *test_sistema;

     SC_CTOR( tbtop )
     {
          test_sistema = new tb_sistema("tb_sistema");
          test_sistema->CK_IN(CK_IN);
          test_sistema->ResetIN_OBM_0(ResetIN_OBM_0);
     test_sistema->HRESETn_OBM_1(HRESETn_OBM_1);
     }
};

#ifdef MTI_SYSTEMC
SC_MODULE(top)
{
     sc_clock clk;
     sc_signal<bool>  ResetIN_OBM_0;
     sc_signal<bool>  HRESETn_OBM_1;
     sc_signal<bool>  Led;
     sc_signal<bool>  ck_out;

     //Definició de l'entitat VHDL
     sistema *sistema_;

     //Definicio del testBench en SystemC
     tbtop *test;

     SC_CTOR(top)
          : clk("clk",10, 0.5, 0.0)
     {
          sistema_ = new sistema("sistema", "work.sistema");
          sistema_->CK_IN(clk);
          sistema_->ResetIN_OBM_0(ResetIN_OBM_0);
        sistema_->HRESETn_OBM_1(HRESETn_OBM_1);
          sistema_->LED_ON_0_1(Led);
          sistema_->CK_OUT1_OBM_1(ck_out);

          test = new tbtop("tbtop");
          test->CK_IN(clk);
          test->ResetIN_OBM_0(ResetIN_OBM_0);
          test->HRESETn_OBM_1(HRESETn_OBM_1);

     }
};

#endif
#endif
```

Mientras que el de Top.cpp es:

```
#include "systemc.h"

#include "top.h"

#ifdef MTI_SYSTEMC

SC_MODULE_EXPORT(top);

#endif
```

tb_sistema

tb_sistema es un test bench sobre sistema pensado principalmente para controlar las señales de reset, ResetIN y ResetWDT. Para ello, primeramente en tb_sistema.h se crea el módulo tb_sistema y se vinculan sus señales y el proceso encargado de realizar la funcionalidad. Dicho proceso se implementa en tb_sistema.cpp en lenguaje SystemC y básicamente desactiva las señales de reset anteriores y entra en un bucle de espera infinito. Todo ello queda más claro en el código fuente:

tb_sistema.h

```
#ifndef TB_SISTEMA_H
#define TB_SISTEMA_H

#include "systemc.h"

//Definició de la entitat contador
SC_MODULE(tb_sistema)
{
    //Definició de les senyals d'Entrada/Sortida
    sc_in<bool>         CK_IN;
    sc_out<bool>        ResetIN_OBM_0;
    sc_out<bool>        HRESETn_OBM_1;

    //Constructor de la classe contador
    SC_CTOR(tb_sistema)
    {
            SC_CTHREAD(testBench, CK_IN.pos());
    }

            //Declaració de les funcions membre
            void testBench(void);
};

#endif
```

tb_sistema.cpp

```
#include “tb_sistema.h”

//Implementación de los procesos
Void tb_sistema::testBench()
{
            //Desactiva Reset
            HRESETn_OBM_1.write(true);
            ResetIN_OBM_0.write(true);

            wait(70, SC_NS);

while(true)
{
            wait();
}
}

#endif
```

1.4.4. Script de ModelSim

Para la creación del script, run.do, se utilizó como base uno ya existente, proporcionado por Jaume Joven Murillo del Departamento de Microelectrónica y Sistemas Electrónicos (MiSE) de la Universidad Autónoma de Barcelona. El script resultante de la adaptación al presente proyecto del código de Jaume es el siguiente:

```
onbreak {resume}

# Creación de la librería
if [file exists work] {
    vdel -all
}
vlib work

# Compilación del código Verilog
vlog *.v

# Compilación y lincado de SystemC
# sccom -g uc_mov.cpp
scgenmod -bool sistema > sistema.h

sccom *.cpp
sccom -link

# Inicio y ejecución de la simulación
# vsim -novopt test_ringbuf
vsim -t 10ns top
run 150000ns
```

Ejecutando éste script se compila y sumula todo el sistema, la parte en SystemC junto con la de Verilog, automáticamente. Pero para poder visualizar las formas de onda deseadas, es necesario añadir un nuevo script cuyo proceso de obtención está explicado en [5], y que es el siguiente:

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate -divider RESET&CLOCK
add wave -noupdate -color Green -format Logic -label HRESETn /top/sistema/AHBTop0/AHB_DummyMaster0/HRESETn
```

```
add wave -noupdate -format Logic -label HCLK /top/sistema/AHBTop0/AHB_DummyMaster0/HCLK
add wave -noupdate -divider ARBITER
add wave -noupdate -format Logic -label HLOCK /top/sistema/AHBTop0/AHB_DummyMaster0/HLOCK
add wave -noupdate -format Logic -label HBUSREQ /top/sistema/AHBTop0/AHB_DummyMaster0/HBUSREQ
add wave -noupdate -format Logic -label HGRANT /top/sistema/AHBTop0/AHB_DummyMaster0/HGRANT
add wave -noupdate -format Literal -label HMASTER /top/sistema/AHBTop0/AHB_DummyMaster0/HMASTER
add wave -noupdate -divider CONTROL
add wave -noupdate -format Literal -label ESTADO -radix decimal /top/sistema/AHBTop0/AHB_DummyMaster0/estado
add wave -noupdate -format Literal -label HBURST /top/sistema/AHBTop0/AHB_DummyMaster0/HBURST
add wave -noupdate -format Literal -label HSIZE /top/sistema/AHBTop0/AHB_DummyMaster0/HSIZE
add wave -noupdate -format Literal -label HTRANS /top/sistema/AHBTop0/AHB_DummyMaster0/HTRANS
add wave -noupdate -divider TRANSFER
add wave -noupdate -format Literal -label HADDR -radix hexadecimal /top/sistema/AHBTop0/AHB_DummyMaster0/HADDR
add wave -noupdate -color Red -format Literal -label HWDATA -radix decimal
/top/sistema/AHBTop0/AHB_DummyMaster0/HWDATA
add wave -noupdate -color White -format Logic -label HREADY /top/sistema/AHBTop0/AHB_DummyMaster0/HREADY
add wave -noupdate -color Gold -format Literal -label HRDATA -radix decimal
/top/sistema/AHBTop0/AHB_DummyMaster0/HRDATA
add wave -noupdate -color Green -format Literal -label HRESP /top/sistema/AHBTop0/AHB_DummyMaster0/HRESP
add wave -noupdate -color Magenta -format Logic -label HWRITE /top/sistema/AHBTop0/AHB_DummyMaster0/HWRITE
add wave -noupdate -divider SLAVE
add wave -noupdate -format Literal -label ESTADO /top/sistema/AHBTop0/AHB_DummySlave2/estado
add wave -noupdate -color Blue -format Logic -label HSEL /top/sistema/AHBTop0/AHB_DummySlave2/HSEL
add wave -noupdate -color Magenta -format Logic -label HWRITE /top/sistema/AHBTop0/AHB_DummySlave2/HWRITE
add wave -noupdate -format Literal -label HADDR -radix hexadecimal /top/sistema/AHBTop0/AHB_DummySlave2/HADDR
add wave -noupdate -color Red -format Literal -label HWDATA -radix decimal
/top/sistema/AHBTop0/AHB_DummySlave2/HWDATA
add wave -noupdate -format Literal -label HTRANS /top/sistema/AHBTop0/AHB_DummySlave2/HTRANS
add wave -noupdate -color Gold -format Literal -label HRDATA -radix decimal
/top/sistema/AHBTop0/AHB_DummySlave2/HRDATA
add wave -noupdate -color Green -format Literal -label HRESP /top/sistema/AHBTop0/AHB_DummySlave2/HRESP
add wave -noupdate -color White -format Logic -label HREADY /top/sistema/AHBTop0/AHB_DummySlave2/HREADY
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {11470 ps} 0}
configure wave -namecolwidth 121
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 0
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
update
WaveRestoreZoom {0 ps} {283970 ps}
```

1.5. Probando el modelo

Terminados todos los preparativos previos a la simulación, ya solo queda probar toda la funcionalidad del modelo. Para ello, se ejecutan distintas simulaciones cambiando ciertos parámetros en los procesos utilitarios para de ese modo obligar a los módulos a que realicen las tareas que se desean probar. El uso de ModelSim permite ver las formas de onda que producen las señales del bus, además, el código de los procesos utilitarios generan los archivos “Maestro.txt” y “Esclavo.txt”, cuya consulta es fundamental para la verificación del buen funcionamiento del modelo.

1.5.1. Prueba de Transferencias SINGLE

Operación de escritura

En esta prueba el master enviará un único dato al esclavo quien ha de leerlo correctamente. Para ello, se utiliza el código utilitario [7] sobre el modelo en el entorno

ModelSim con lo que se obtienen los siguientes ficheros de salida, además de las formas de onda de la figura 6:

Archivo Master.txt:

> Operación completada satisfactoriamente

Archivo Esclavo.txt:

> Operación de recepción de datos Completada: Se han recibido 1 datos
>
> Dato/s recibido/s:
> 76

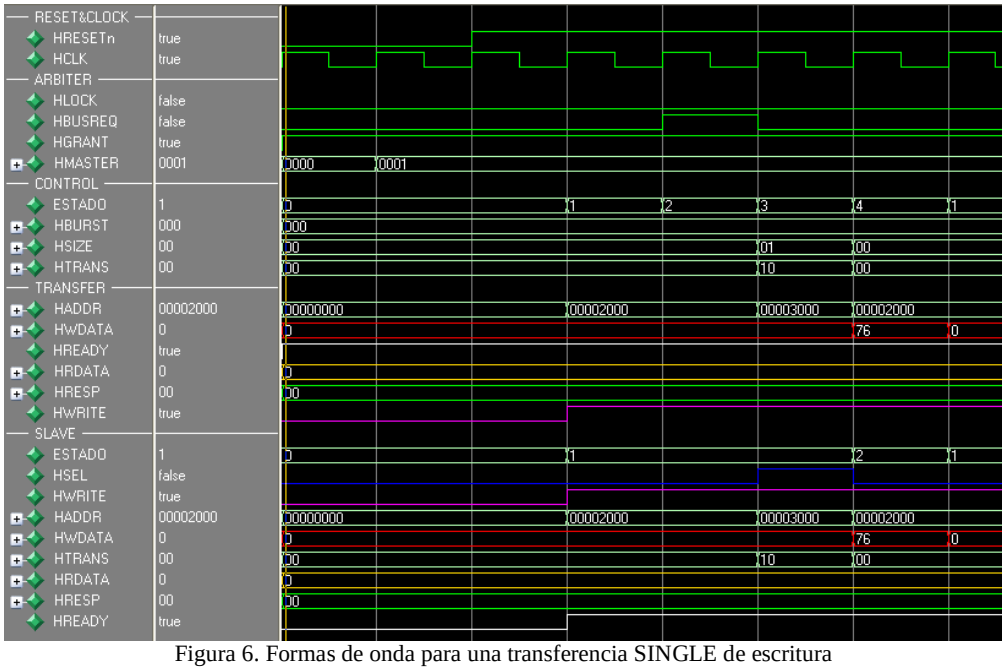

Figura 6. Formas de onda para una transferencia SINGLE de escritura

En la figura 6 pueden observarse todas las señales del bus AMBA AHB que intervienen en la operación demandada. Además, también puede consultarse el valor de la variable estado tanto del master como del esclavo del AMBA. Recuérdese que dicha variable contiene el estado actual de la máquina de estados y que todos los cambios en las señales que se hagan en un ciclo de reloj, no se verán reflejados hasta el siguiente ciclo.

Así pues, puede verse como inicialmente tanto el master como el esclavo del bus inician su ejecución en el estado 0, RESET, donde programan señales cuyo cambio puede apreciarse en el estado 1. En este estado permanece el master hasta recibir la orden de iniciar una operación y, sucedido esto, activa la señal HBUSREQ y pasa al estado 2 donde permanecerá hasta que el árbitro le conceda el bus. Cuando esto sucede, inicia la fase de dirección, estado 3, donde programa las señales de control y de dirección para acto seguido iniciar el estado 4, fase de datos, y enviar la información a través del bus. Finalmente, enviado el dato comprueba la respuesta del esclavo y retorna al estado 1 donde permanecerá hasta nueva orden de operación.

A todo esto, el Esclavo permanece a la escucha de la señal HSEL que le notificará que el Master va a realizar una operación con él. Al activarse dicha señal, el Esclavo consulta el tipo de operación requerida y al ver que es de escritura, mira el estado de su cola de datos de entrada, data_in, para comprobar si tiene capacidad para recibir datos. Puesto que así es, programa la respuesta HRESP = OKAY con HREADY = cierto con lo que está comunicando su disponibilidad para operar. Hecho esto, pasa al estado OPERACION donde recibe el dato 76 y lo almacena en la cola de entrada. Finalmente, al perder HSEL retorna al estado 1 donde aguardará hasta una nueva operación.

Analizada en profundidad la primera prueba, el resto de las pruebas no se detallarán tan minuciosamente.

La siguiente prueba consiste en ejecutar el mismo código pero esta vez con la cola de entrada de datos del esclavo llena. Para llenarla, basta con ejecutar un bucle for al principio donde en cada iteración se entra un dato en la cola y se espera un ciclo. Se modifica también a maestro para que espere 15 ciclos antes de iniciar la operación y le de tiempo así a esclavo a llenar su cola. Véanse los resultados obtenidos tras la prueba:

Archivo “Master.txt”

> Operación completada satisfactoriamente

Archivo “Esclavo.txt”:

> Vaciando cola de entrada... Dato leído: 0
> Vaciando cola de entrada... Dato leído: 1
> Vaciando cola de entrada... Dato leído: 2
>
> Operación de recepción de datos Completada: Se han recibido 14 datos
>
> Dato/s recibido/s:
> 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 76,

La situación descrita en la Figura 2 es consecuencia del hecho de haber forzado la cola a que este llena. En un uso normal del modelo no tiene porque darse esta situación si el código del proceso utilitario hace una buena gestión de las colas.

Respecto a los archivos, se ha de tener en cuenta que no es que se hayan leído 14 datos, sino que al final de la operación hay 14 datos en la cola. El dato recibido durante la operación es el último de la cola, el valor 76.

Nótese como se han vaciado 3 datos de la cola al activarse buffer_lleno. El código de EsclavoAMBA_AHB obliga a que se vacíen 2 datos como mínimo para proseguir la operación una vez que se activa buffer_lleno. El porque se han vaciado 3 en vez de 2 es, con toda seguridad, consecuencia de la no utilización de semáforos ni de otro mecanismo de concurrencia sobre la variable buffer_lleno. Por este motivo su valor no cambia de cierto a falso y viceversa de forma exacta cuando toca y por ello se produce esta pequeña descoordinación. De todos modos, puesto que se ha recibido el dato correctamente se considera que la prueba ha sido satisfactoria, ya que como se viene anunciando, del uso de los datos se encargan los procesos utilitarios y ellos son los responsables de que no se produzcan situaciones comprometidas como la acontecida.

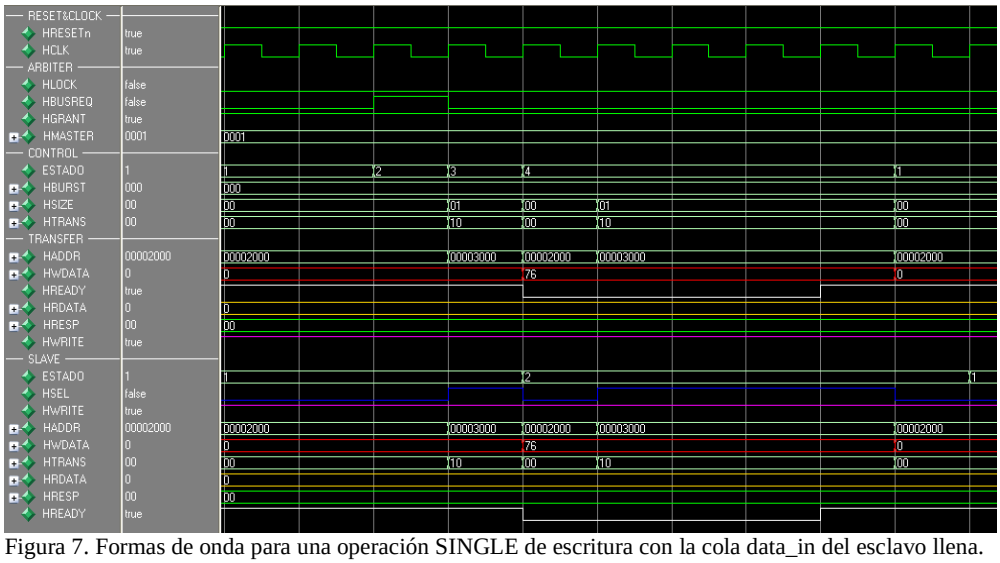

Figura 7. Formas de onda para una operación SINGLE de escritura con la cola data_in del esclavo llena.

Continuando con las pruebas, la siguiente consiste en una transferencia SINGLE de escritura donde la cola de salida de datos del master del código utilitario [8], data_out, está vacía. Véanse los resultados obtenidos:

Archivo “Master.txt”:

> Se requieren datos... Dato proporcionado: 100
>
> Se requieren datos... Dato proporcionado: 100
>
> Operación completada satisfactoriamente

Archivo “Esclavo.txt”:

> Operación de recepción de datos Completada: Se han recibido 1 datos
>
> Dato/s recibido/s:
> 100

La prueba ha sido satisfactoria tal y como se aprecia en la figura 3. Respecto a los archivos, en ellos se reafirma el éxito de la prueba y se aprecia como, de nuevo, cuando únicamente se requería un dato se han proporcionado dos. Las causas de ello vuelven a ser las mismas que anteriormente, pero puesto que no se entiende el iniciar una operación de envío de datos sin disponer de dichos datos, se asume que es el utilitario quien en este caso está haciendo un mal uso del modelo y que éste funciona bien pues se ha enviado la información correctamente.

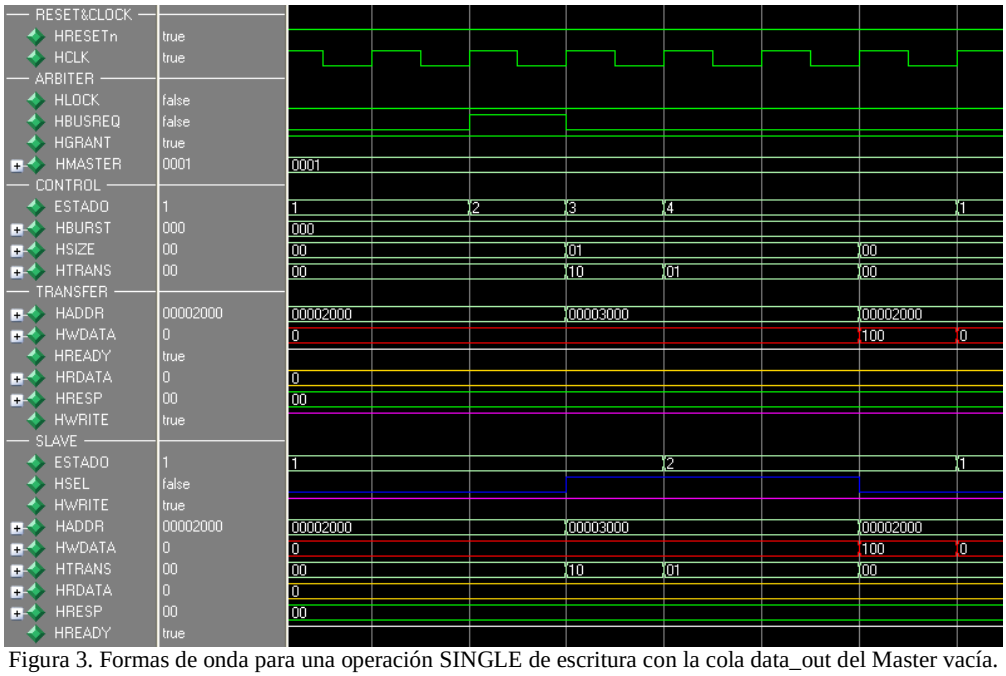

Figura 3. Formas de onda para una operación SINGLE de escritura con la cola data_out del Master vacía.

Operación de lectura

Viendo que la gestión de las colas funciona correctamente, ahora solamente se probará que el modelo funcione trabajando de forma normal sin eventualidades producidas por una mala gestión de las colas por parte de los procesos utilitarios. Véanse pues los resultados de una operación de lectura:

Archivo “Master.txt”:

> Operación completada satisfactoriamente
> Se ha recibido el dato/s: 15

Archivo “Esclavo.txt”:

> Se ha enviado dato/s correctamente

Como cabía esperar, el Master recibe el dato que el Esclavo le proporciona sin ninguna complicación, tal y como se aprecia en la figura 5. En ella, se ven claramente los distintos estados por los que pasan el master y el esclavo del bus y como finalmente el dato 15 circula por el bus HRDATA para ser recogido por el Master.

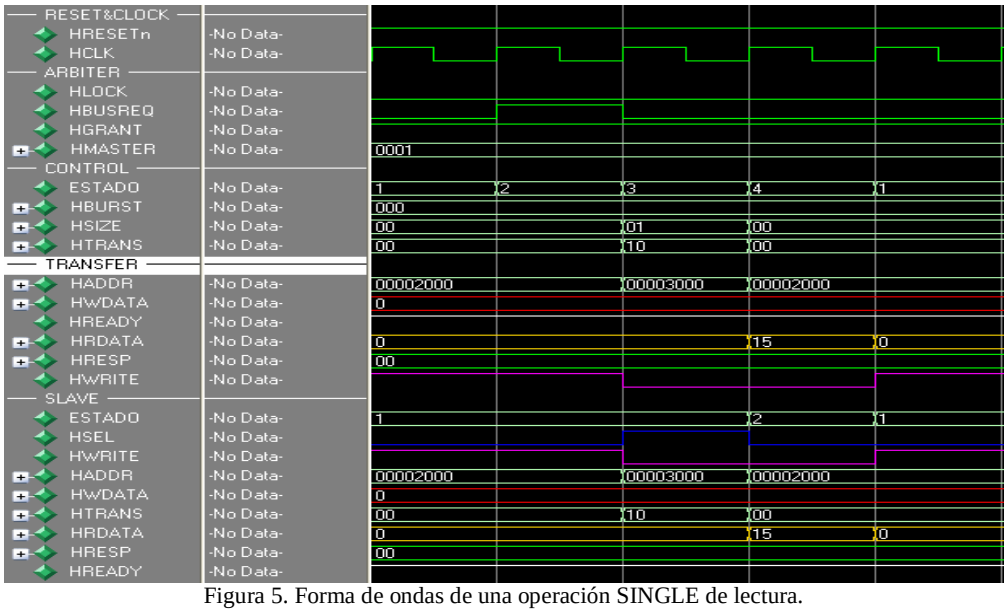

Figura 5. Forma de ondas de una operación SINGLE de lectura.

1.5.2. Prueba de Transferencias BURST (en ráfaga)

Es momento ahora de probar las operaciones de envío y recepción de más de un dato. Como se ha visto en teoría, el bus AMBA soporta distintos tipos y tamaños de ráfaga, pero en el caso del presente modelo, únicamente se trabaja con las ráfagas de tipo incremental. Dentro de éste tipo no será necesario probar exhaustivamente cada tamaño de ráfaga ya que si las colas funcionan correctamente, no habrá entre ellos diferencias más sustanciales que el propio número de datos a enviar recibir.

Operación de escritura

Inicialmente se probará una operación INCR4 de escritura donde se pretende enviar los datos 23, 47, 76 y 101, para lo cual se utilizan los procesos utilitarios [9].Véase la salida en los archivos generados y las formas de onda obtenidas (Fig. 6):

Archivo “Master.txt”:

> Operación completada satisfactoriamente

Archivo “Esclavo.txt”:

> Operación de recepción de datos Completada: Se han recibido 4 datos
>
> Dato/s recibido/s:
> 23, 47, 76, 101

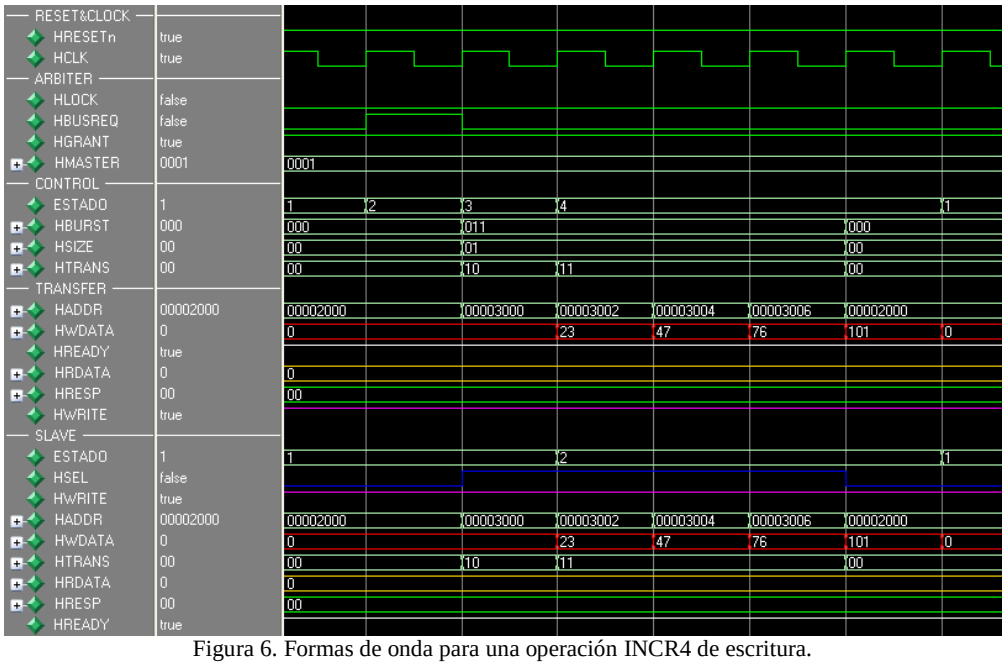

Figura 6. Formas de onda para una operación INCR4 de escritura.

Puede verse en los archivos como la operación se resuelve perfectamente. Si se observa la figura 6 se ve como se incrementa la dirección de HADDR a cada nuevo envío acorde con el tamaño del dato emitido. Además, también se ve como se anidan la fase de dirección y la fase de datos en un mismo ciclo hasta completar el envío. Obsérvese también como al programar la primera dirección HTRANS vale 10 (NONSEQ) con lo que se indica que se trata del primer dato de una ráfaga de uno o varios elementos. Luego, para el resto de datos de la ráfaga HTRANS se programa con el valor 11 (SEQ) que indica que el dato pertenece a una secuencia de datos enviados en ráfaga y que no es el primer dato.

Operación de lectura

Finalmente, la última prueba consistirá en una operación de lectura en ráfaga INCR de tamaño 6 donde se obtienen los siguientes resultados:

Archivo “Master.txt”:

Operación completada satisfactoriamente
Se ha recibido el dato/s: 14
Se ha recibido el dato/s: 81
Se ha recibido el dato/s: 53
Se ha recibido el dato/s: 26
Se ha recibido el dato/s: 48
Se ha recibido el dato/s: 333

Archivo “Esclavo.txt”:

Llenando cola de salida... Dato Volcado: 23

Se ha enviado dato/s correctamente

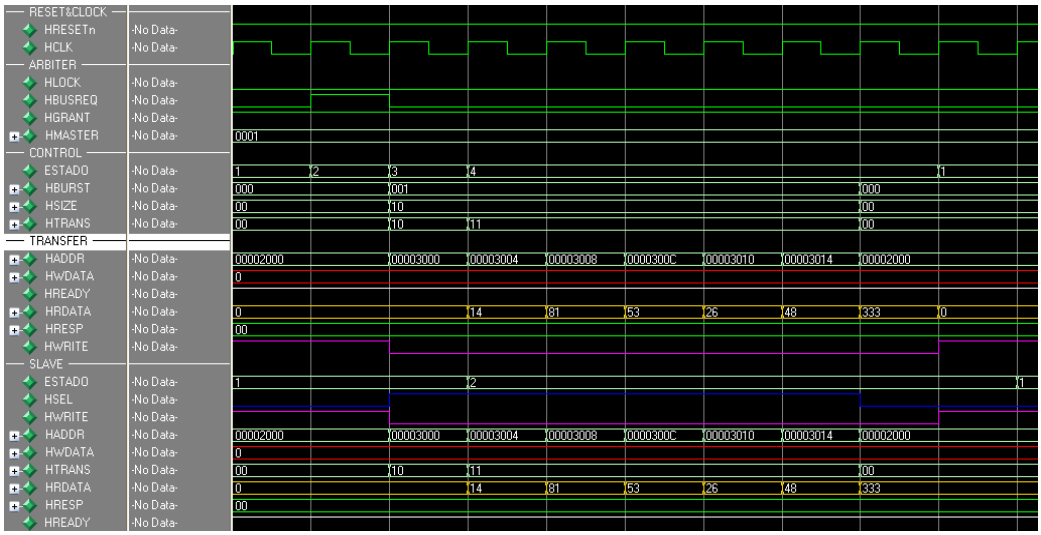

Figura 8. Operación INCR de lectura de 6 datos.

Como muestra la Figura 8, la operación concluye satisfactoriamente.

Referencias

[1] Apéndice 4
[2] Apartado 8.2.1.1.
[3] Apéndice 3
[4] Apartado 8.2.1.2.
[5] Apartado 6.2.
[6] Apartado 5.2.
[7] Apartado 8.2.1.1.
[8] Apartado 8.2.1.2.
[9] Apartado 8.2.2.

Apéndice 2. Montaje y prueba final

2.1. Introducción

Comprobado el correcto funcionamiento del modelo dentro de la estructura ModelSim, es momento ahora de sacarlo fuera y probarlo en un código, que aunque de operación sencilla, requiere de mucha comunicación entre los módulos que lo integran. Para ello, se genera un proyecto nuevo en el Microsoft Visual C++ 6.0 donde se monta toda una infraestructura [1] necesaria para el funcionamiento del modelo y que ilustra la Figura 1.

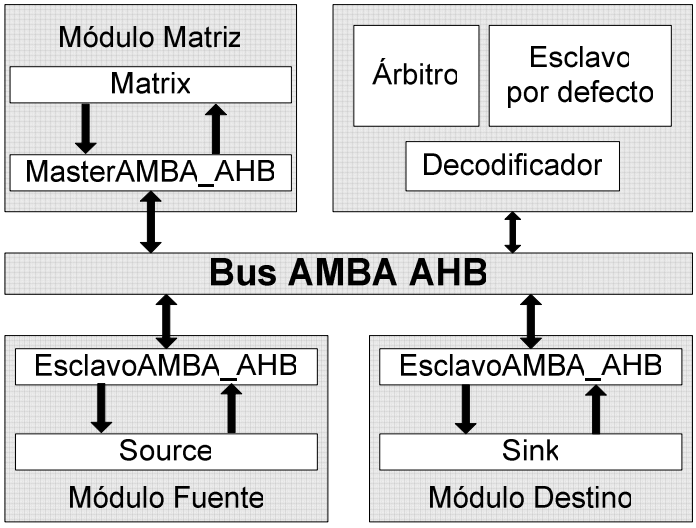

Figura 1. Sistema para el cálculo de la transposición de matrices.

Todo el sistema descrito en la Figura 1 tiene como finalidad el cálculo de la transposición de 4 matrices de 8x8. Como se ve, es necesario implementar a un árbitro, un decodificador y un esclavo por defecto puesto que ya no se cuenta con el bus AMBA AHB escrito en verilog y cedido por el MiSE. Otro aspecto importante es el hecho de utilizar dos esclavos del bus, aparte del esclavo por defecto, situación que no se ha probado aún.

2.2. Funcionamiento

El funcionamiento es sencillo, el proceso Source [2] prepara 4 matrices de 8x8 asignándoles valores del 0 al 255 como se muestra en la Figura 2. Después espera hasta que EsclavoAMBA_AHB inicie una operación para irle suministrando los datos según va siendo necesario.

A todo esto, Matriz [3] programa a MasterAMBA_AHB para que inicie operaciones INCR8 para ir leyendo las matrices fila a fila. Conforme las va recibiendo, las guarda en columnas con lo que va calculando la transpuesta de la matriz. Finalmente, cuando ha recibido las 4 matrices, por lo que ha efectuado 32 operaciones INCR8 de forma correcta a través del modelo, inicia el envío de las matrices transpuestas (Fig. 3) al

proceso Sink [4] a través del bus AMBA AHB, con lo que serán 32 nuevas transacciones correctas a través del modelo.

Sink queda a la espera de que el EsclavoAMBA_AHB de su mismo módulo le notifique el inicio de una operación. Al suceder esto, recoge los datos llegados a través de la cola datos_in y los va almacenando en las cuatro matrices. Al concluir la operación, escribe el resultado obtenido en el archivo de texto “matriz.txt” y finaliza la ejecución del sistema.

Matriz 1:
0 1 2 3 4 5 6 7
8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63

Matriz 2:
64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95
96 97 98 99 100 101 102 103
104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127

Matriz 3:
128 129 130 131 132 133 134 135
136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151
152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183
184 185 186 187 188 189 190 191

Matriz 4:
192 193 194 195 196 197 198 199
200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215
216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231
232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247
248 249 250 251 252 253 254 255

Figura 2. Matrices originales generadas en el proceso Source

Matriz 1:
0 8 16 24 32 40 48 56
1 9 17 25 33 41 49 57
2 10 18 26 34 42 50 58
3 11 19 27 35 43 51 59
4 12 20 28 36 44 52 60
5 13 21 29 37 45 53 61
6 14 22 30 38 46 54 62
7 15 23 31 39 47 55 63

Matriz 2:
64 72 80 88 96 104 112 120
65 73 81 89 97 105 113 121
66 74 82 90 98 106 114 122
67 75 83 91 99 107 115 123
68 76 84 92 100 108 116 124
69 77 85 93 101 109 117 125
70 78 86 94 102 110 118 126
71 79 87 95 103 111 119 127

Matriz 3:
128 136 144 152 160 168 176 184
129 137 145 153 161 169 177 185
130 138 146 154 162 170 178 186
131 139 147 155 163 171 179 187
132 140 148 156 164 172 180 188
133 141 149 157 165 173 181 189
134 142 150 158 166 174 182 190
135 143 151 159 167 175 183 191

Matriz 4:
192 200 208 216 224 232 240 248
193 201 209 217 225 233 241 249
194 202 210 218 226 234 242 250
195 203 211 219 227 235 243 251
196 204 212 220 228 236 244 252
197 205 213 221 229 237 245 253
198 206 214 222 230 238 246 254
199 207 215 223 231 239 247 255

Figura 3. Matrices resultado generadas en el proceso Matrix y mostradas en el Sink

Referencias

[1] Apartados 8.3.1., 8.3.2. y 8.3.3.
[2] Apartado 8.3.4.
[3] Aparrtado 8.3.5.
[4] Apartado 8.3.6.

Apéndice 3. SystemC

Comentario [L2]: Apartado cambiado por completo

3.1. Introducción

SystemC son unas librerías de C++ que permiten modelar hardware. El hecho de que se trate de lenguaje C++, confiere a este sistema una potencia que otros lenguajes similares, como VHDL o Verilog, no pueden conseguir. Además, al ser SystemC una librería de C++, posibilita el co-diseño al poder diseñar partes en hardware y partes en software ya que C++ es un lenguaje software. La especificación [1] junto con las librerías del mismo está disponible de forma gratuita en www.systemc.org, Web donde también se puede participar en el proyecto junto con la comunidad SystemC. Además, también hay muchos libros [2] donde consultar.

A grandes rasgos, el lenguaje se basa en la definición de módulos y de cómo estos se interconectan entre si mediante puertos y señales. Para que todo ello funcione, es necesario un módulo principal que instancie a todos los demás y que estimule las señales de reloj y reset. Dicho módulo será el main que, además de lo ya mencionado, se encargará de iniciar la ejecución del sistema.

Los distintos módulos del sistema se crean como las clases de C++, pero teniendo en cuenta la sintaxis SystemC. De este modo, para cada módulo tendremos un archivo .h dónde se definen las señales del mismo, las funciones y variables que lo conforman y el constructor. Después, en un archivo .cpp se implementan, a excepción del constructor, las funciones y métodos definidos previamente en el .h.

El siguiente ejemplo sencillo permitirá explicar como se utiliza el lenguaje sin entrar en detalles.

3.2. Código ejemplo en C/C++

El código de la operación escrito en lenguaje C/C++ es tan sencillo como el siguiente:

```
#include <stdio.h>
int main()
{
	int A = 10;
	int B = 20;
	int C = 30;
	long D;

	D = A + B + C;
	printf("\nEl resultado de la operación A + B + C es: %ld\n", D);
	return 0;
}
```

El primer paso es incluir las librerías necesarias, stdio.h (estándar input output) en este caso. Después, se implementa la función main que no recibe ningún argumento. En el main, se declaran como enteros (int) las variables A, B y C a las que se asigna un valor al tiempo de la declaración. También se declara la variable D que contendrá el resultado

de la operación y que será de tipo long por si éste valor se sale del rango entero. Finalmente se ejecuta la operación, se muestra el resultado por pantalla con un mensaje informativo mediante el comando printf y se sale del programa con return.

Queda claro que éste es un ejemplo trivial que no presenta dificultad ninguna, cualidad que lo hace idóneo para iniciarse en SystemC y aprender las particularidades de éste.

3.3. Código ejemplo en SystemC

3.3.1. Estructura

Puesto que SystemC es un lenguaje para modelar hardware, el código se ha de estructurar en módulos como si de cajitas se tratase ya que físicamente se organiza de este modo. Para verlo más claro, no hay más que tomar el ejemplo de un sumador. En todo circuito microelectrónico, las entradas de dicho sumador procederán de algún otro elemento, como por ejemplo un registro. De igual modo, el resultado del sumador irá a parar a otro registro u otro elemento, pero siempre externo al sumador. Por tanto, queda claro que se han de separar las entradas y salidas de datos de la operación en si.

Consecuentemente, habrá tres módulos para realizar la operación. Uno de ellos se encargará de realizar la operación en si, mientras que otro suministrará los datos con los que operar. El último recibirá el resultado de la operación y, en este caso, lo mostrará por pantalla. Por tanto, los módulos fuente, proceso y destino, han de trabajar conjunta y coordinadamente para dar con el resultado de la operación. Para posibilitar ésta coordinación, es necesario el uso del módulo main cuya inclusión da lugar a la estructura completa del mismo (Fig. 1).

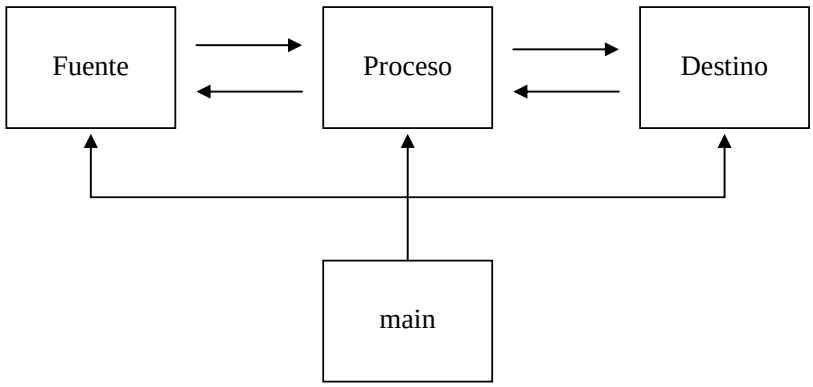

Figura 1. Estructura del código SystemC necesaria para implementar la operación A + B + C

3.3.2. Fuente

El módulo fuente se encarga de suministrar los valores de A, B y C para que proceso los pueda sumar. El archivo .h que contiene la declaración del módulo presenta el siguiente aspecto:

```
#include "systemc.h"
SC_MODULE( fuente )
{
	//Señales
	sc_in_clk		reloj;
	sc_in< bool >		reset;
	sc_in< bool >		data_req;
	sc_out< bool >		data_valid;
	sc_out< sc_int<32> >	data_out;

	//Procesos
	void fuente();

	//Constructor
	SC_CTOR( fuente )
	{
		SC_CTHREAD( fuente, reloj.pos() );
			watching( reset.delayed() == false );
	}
};
```

Como se viene anunciando, la estructura del código es muy similar a la de C/C++. En primer lugar, mediante la instrucción #include se añaden las librerías y archivos cuyo código se desea utilizar, teniendo en cuenta que en todo programa escrito en SystemC, la inclusión de la librería systemc.h es de obligado cumplimiento ya que es en ella donde se define toda la sintaxis y funcionalidad del lenguaje. De ahí que la primera instrucción sea **#include "systemc.h"** y, puesto que no se necesita ninguna más, sea la única librería que se incluya.

El siguiente paso es definir el módulo para lo cual SystemC utiliza una macro que se invoca con SC_MODULE. Ésta recibe por parámetro el nombre que se le asignará al módulo que en el presente ejemplo es fuente. De ahí la instrucción **SC_MODULE(fuente)**. Hecho esto, dentro del módulo se definen las variables y métodos que contendrá, tal y como se hace en una clase en C/C++, pero además, en SystemC también hay que definir nuevos elementos que no son ni variables ni constantes.

Antes se ha visto que hay que dividir el problema de manera que se resuelva utilizando varios módulos entendidos como cajitas interconectadas. Luego esta interconexión es la pista que, por ejemplo, une físicamente la salida de un registro con la entrada de un sumador, lo cual no es bien bien una variable. Por tanto, para modelar las interconexiones entre elementos de un circuito, SystemC utiliza unas estructuras de datos especiales que se conocen como señales y que principalmente se distinguen entre dos grandes tipos, las de entrada y las de salida.

Volviendo al código de ejemplo, dentro del módulo fuente se define una serie de señales exactamente como se definiría una variable en C/C++, primero el tipo y después el nombre, tal y como se ve a continuación:

```
	sc_in_clk		reloj;
	sc_in< bool >		reset;
	sc_in< bool >		data_req;
	sc_out< bool >		data_valid;
	sc_out< sc_int<32> >	data_out;
```

La primera de todas es una señal de entrada pues su declaración empieza por sc_in. Al tratarse de un reloj, tal y como su nombre indica, se define como sc_in_clk. El reloj es un elemento esencial en SystemC pues casi la totalidad de los componentes de un circuito utilizan este elemento para la sincronización y coordinación.
La segunda señal también es muy importante pues se trata del reset del sistema. Se define como sc_in< bool > porque es una señal de entrada boleana, lo cual quiere decir que únicamente tomará los valores cierto (1) o falso (0). Nótese la sintaxis de SystemC, donde para la definición de una señal de entrada, se utiliza sc_in y entre los símbolos < > se especifica el tipo de datos de la misma.
Finalmente, la última señal de entrada es data_req y es de tipo boleano, por lo que se define como sc_in< bool >.

Respecto a las señales de salida, en primer lugar se define data_valid como sc_out< bool >. Al ser una señal de salida su declaración empieza por sc_out y entre los símbolos < > se especifica el tipo de la señal, bool en este caso. Seguidamente se define data_out que es una señal de salida del tipo sc_int<32>. Éste es el tipo de variable entera de SystemC y aunque conceptualmente es igual que el típico int de C/C++, su estructura es totalmente diferente.

Volviendo a los fundamentos físicos, en un circuito solamente existen las pistas de metal o polisilicio para conseguir una interconexión y éstas solo admiten los valores lógicos de 0 y 1 que significan, dependiendo de la tecnología, por ejemplo 1.2V o 0V para una tecnología de 130nm. Por ello, si se desea poder transportar más información, se han de organizar varias pistas formando un bus. Luego por ejemplo, si se forma un bus de 16 líneas, la información transportada pasa a ser 16 ceros y unos que, si se toman como un todo y se leen en binario, son capaces de representar números desde el 0 hasta el $2^{16} - 1$, o incluso números negativos si se utilizan alternativas como el complemento a2. Por lo tanto y volviendo al tipo sc_int<32>, lo que realmente se está definiendo es bus de 32 pistas con la propiedad de que utilizando métodos de la clase sc_int se pueden tratar las pistas por separado y darles el valor 0 o 1 de forma individual. Así mismo, también se pueden hacer asignaciones del tipo variable = número.

Comentario [M3]: Tengo dudas de si lo que digo en este párrafo es cierto. Miratelo por favor.

Terminada la definición de variables, constantes y señales, el siguiente paso es el de definir los procesos del módulo. Éstos son muy parecidos a los métodos de una clase de C/C++ salvo por algunas diferencias. Los procesos en SystemC no son jerárquicos, es decir, no es posible tener procesos dentro de otros procesos, la jerarquía la proporcionan los módulos, no los procesos. Tampoco se pueden llamar, como si de funciones se tratase, directamente desde el código, si no que se invocan de un modo especial que se verá más adelante. Luego en el código de ejemplo, se define el proceso fuente del siguiente modo: **void fuente();**

Una vez están todos los componentes del módulo definidos, se ha de implementar el constructor. Para ello SystemC incorpora una nueva macro que realiza todo el trabajo y que se corresponde con el siguiente código del ejemplo:

```
SC_CTOR( fuente )
{
        SC_CTHREAD( fuente, reloj.pos() );
                watching( reset.delayed() == false );
}
```

En él se puede ver como se invoca a la macro mediante SC_CTOR a quien se le pasa el nombre del módulo que se ha utilizado en el SC_MODULE.. En esta macro se registran uno o varios procesos que se ejecutarán al instanciar el módulo en el main como se verá después. Todos ellos han de pertenecer al módulo, por tanto estar declarados en el mismo, y pueden ser de tres tipos: SC_METOD, SC_THREAD y SC_CTHREAD. En el ejemplo se utiliza un proceso SC_CTHREAD que se registra con la macro **SC_CTHREAD(fuente, reloj.pos())** donde fuente es el nombre del proceso a registrar y reloj.pos() es el reloj que se utilizará para la sincronización del código pues los procesos del tipo CTHREAD siempre requieren de un reloj.

Comentario [M4]: Para qué sirve el método .pos()??

Por último, **watching(reset.delayed()==false)** incluido dentro de la macro SC_CTHREAD hace que el proceso registrado en dicha macro quede a la escucha de la señal reset porque así se le ha dicho al pasarla como parámetro del watching. Así, si se modifica el valor de la señal reset a falso, se reiniciará la ejecución del código del proceso fuente sea cual sea el punto en el que se encuentre.

Con esto se termina el código del archivo .h donde se han definido variables, constantes, procesos y, finalmente, se ha creado el constructor. Ahora, en el archivo .cpp se implementa el código de éstos procesos:

```
#include "fuente.h"
void fuente::fuente()
{
	data_valid.write(false);
	wait_until( reset.delayed() == true );
	while( true )
	{
		wait_until( data_req.delayed() == true );
		data_out.write( 10 );
		data_valid.write( true );
		wait();
		data_out.write( 20 );
		wait();
		data_out.write( 30 );
		wait_until(data_req.delayed() == false);
		data_valid.write( false );
	}
}
```

Igual que en el .h, se inicia el código con la inclusión de las librerías. Luego, exactamente igual que en C++, es necesario definir el ámbito del proceso, de ahí la instrucción **void fuente::fuente(),** donde se notifica que la función fuente, a la derecha de ::, pertenece al módulo fuente, a la izquierda de :: e inmediatamente después del void. Después, se inicia la programación del código del proceso que va a repasar paso a paso:

```
	data_valid.write(false);
	wait_until( reset.delayed() == true );
```

El código resaltado en negrita corresponde con las primeras instrucciones que se ejecutarán. Teniendo en cuenta que en caso de activación de la señal reset, por ser la señal utilizada en el watching, se reiniciará el código des del principio y se retomarán éstas instrucciones, es importante que sirvan para inicializar las señales y variables del módulo a un estado conocido desde el que empezar la ejecución. En este caso, se

inicializa la señal data_valid con el valor falso y se espera hasta que la señal reset adquiera el valor cierto.

Al ser una señal de salida, el valor de data_valid puede ser escrito por el módulo fuente, en cambio, la señal reset al ser de entrada no puede escribirse desde fuente, es de solo lectura. La forma de escribir una señal es por medio del método write. Por ello, al ejecutar **data_valid.write(false)** se está poniendo la señal data_valid a 0, que es lo mismo que falso. Hay que tener presente que en los procesos CTHREAD el nuevo valor de dicha señal no se reflejará hasta el siguiente ciclo de reloj, lo cual es un aspecto importante a tener en cuenta.

Para comprender la siguiente instrucción, **wait_until(reset.delayed()==true)**, es necesario algo de teoría primero. Los procesos SC_CTHREAD y SC_THREAD son procesos hilados (thread process), lo cual quiere decir que una vez iniciados siguen sus propios hilos de ejecución de forma independiente al resto y del mismo modo que lo hacen los hilos (threads) de C/C++. En su ejecución, un proceso hilado es capaz de auto suspenderse de manera que su ejecución queda interrumpida hasta que se produzca un evento o un cambio en alguna señal. Para suspenderse, un proceso de este tipo utiliza la instrucción wait que admite como parámetro el motivo de la espera. De ese modo, la instrucción wait(3) espera hasta que pasen 3 ciclos de reloj durante los cuales la ejecución del proceso queda suspendida. De igual modo, si en vez de un número se pasa como parámetro un evento, la explicación del cual no se dará al quedar fuera de los objetivos del presente documento, wait mantendrá el código en suspensión hasta que se produzca tal evento.

Aparte de wait, los procesos SC_CTHREAD cuentan con una variante llamada wait_until, la cual esperará hasta que se cumpla la condición que recibe como parámetro. Por tanto y volviendo al código de ejemplo, **wait_until(reset.delayed() == true)** mantiene al proceso fuente suspendido hasta que la señal de reset cambie su valor a cierto. Por tanto, wait_until viene a ser lo mismo que esto:

```
while(true)
{
	if( reset.delayed() == true )
	{
		break;
	}
	wait();
}
```

Visto esto, la siguiente instrucción es un bucle infinito, por lo que todo el código interior al **while(true)** se ejecutará siempre y secuencialmente:

```
while( true ) {
	wait_until( data_req.delayed() == true );
	data_out.write( 10 );
	data_valid.write( true );
	wait();
	data_out.write( 20 );
	wait();
	data_out.write( 30 );
	wait_until(data_req.delayed() == false);
	data_valid.write( false );
}
```

De hecho, es muy común utilizar bucles infinitos en SystemC pues el hardware no deja de ser un módulo pensado para realizar una acción muy concreta cuando cambian sus variables de entrada. Por ejemplo, un sumador ha de estar continuamente sumando los valores de sus entradas para ofrecer el resultado en su salida. Por tanto, el código que modela tal comportamiento a la fuerza ha de tener una estructura repetitiva.

A nivel funcional, el código resaltado en negrita espera hasta que el módulo proceso le pida datos. Al recibir ésta petición, envía la información requerida y al finalizar la transmisión, vuelve a quedar a la espera. Dicha espera se implementa con la primera instrucción, **wait_until(data_req.delayed() == true)** que espera hasta que la señal data_req proveniente del módulo proceso tome el valor cierto. Una vez cumplida la condición, escribe en la señal de salida el primer dato, **data_out.write(10)** y lo notifica con **data_valid.write(true)**. Luego espera un ciclo de reloj mediante la instrucción **wait()** y escribe el segundo valor con **data_out.write(20)**. Acto seguido vuelve a esperar otro ciclo con otro **wait()** y envía el último dato, **data_out.write(30)**. Hecho esto espera hasta tener la confirmación de que se han recibido los datos por medio de **wait_until(data_req.delayed() == false)** y finalmente pone la señal data_valid a falso con **data_valid.write(false)**.

De este modo y con un protocolo de comunicación sencillo, fuente suministra tres datos a proceso para que éste pueda operarlos como si de un simple sumador se tratara.

3.3.3. Proceso

La declaración del módulo proceso contenida en el archivo proceso.h es muy similar a la del módulo fuente, tal y como se aprecia en el siguiente código:

```
#include "systemc.h"

SC_MODULE( proceso )
{
	//Señales básicas de entrada
	sc_in_clk		reloj;
	sc_in< bool >		reset;

	//Variables locales
	sc_int<32>		A;
	sc_int<32>		B;
	sc_int<32>		C;
	sc_bigint<64>		D;

	//Comunicación con la fuente
	sc_in< bool >		data_valid_fuente;
	sc_out< bool >		data_req_fuente;
	sc_in< sc_int<32> >	data_in_fuente;

	//Comunicación con el destino
	sc_in< bool >		data_ack_destino;
	sc_out< bool >		data_valid_destino;
	sc_out< sc_int<32> >	data_out_destino;

	//Procesos del módulo
	void proceso();

	//Constructor
```

```
	SC_CTOR( proceso )
	{
		SC_CTHREAD( proceso, reloj.pos() );
			watching( reset.delayed() == false );
	}
};
```

Como se ve, la estructura del código es la misma que en el caso de fuente, salvo porque aquí se definen más señales ya que proceso necesita comunicarse con dos módulos en vez de uno. Habiendo visto una explicación detallada de fuente.h, no es necesario hacer lo propio con proceso.h ya que se puede entender fácilmente con la base ya adquirida. Por tanto, véase el archivo .cpp:

```
#include "proceso.h"

void proceso::proceso()
{
	data_req_fuente.write(false);
	data_valid_destinio.write(false);
	wait_until(reset.delayed() == true);
	while( true )
	{
		 /*** Adquisición de datos ***/
		data_req_fuente.write(true);
		wait_until(data_valid_fuente.delayed() == true);
		A = data_in_fuente.read();
		wait();
		B = data_in_fuente.read();
		wait();
		C = data_in_fuente.read();
		data_req_fuente.write(false);
		wait_until(data_valid_fuente.delayed() == false);

		/*** Procesado de los datos ***/
		D = A + B + C;

		/*** Envío de datos ***/
		data_out_destino.write(D);
		data_valid_destino.write(true);
		wait_until(data_ack_destino.delayed() == true);
		data_valid_destino.write(false);
		//Finalizada la operación, espera indefinidamente
		while(true)
		{
			wait();
		}
	}
```

De nuevo el código se inicia con la inclusión de las librerías y el operador de ámbito. Hecho esto, vuelve a tener instrucciones de programación de variables y señales a valores conocidos y un bucle while(true) donde se implementa toda la funcionalidad. Dicha funcionalidad se puede separar en tres bloques lógicos, la adquisición de datos, el proceso de los mismos y su envío a destino. En el primer bloque, se piden los datos a fuente activando data_req_fuente a cierto mediante la instrucción **data_req_fuente.write(true)**. Luego se espera hasta tener la confirmación de la existencia de datos disponibles en el bus de datos con la sentencia **wait_until(data_valid_fuente.delayed() == true)**. Al recibir dicha confirmación, se

notifica que en el bus de datos existe información válida y útil para el programa, en este caso el dato A. Por tanto, almacena dicho dato en su correspondiente variable con **A = data_in_fuente.read()** ya que el método read() se utiliza en las señales de entrada para leer el valor que contienen. Como se ve, dicho método devuelve el valor contenido en la señal y que puede ser almacenado fácilmente o utilizado de cualquier otro modo. Con este mecanismo y mediando un ciclo de reloj de espera entre dato y dato, ser reciben los datos A, B y C a procesar.

Terminado esto, el bloque lógico de proceso de datos consiste en implementar la operación D = A + B + C, que no necesita ninguna explicación. Finalmente, en el bloque del envío de datos a destino de nuevo se utiliza un protocolo sencillo de comunicación consistente en colocar el valor del resultado en el bus de salida, **data_out_destino.write(D)**, y notificar la existencia de información válida en éste mediante **data_valid_destino.write(true)**. Cumplido esto se espera hasta que destino confirme la recepción de D, **wait_until(data_ack_destino.delayed() == true)**, para después confirmar el fin de operación con **data_vald_destino.write(false)**.

Finalmente se entra en un bucle de espera infinito pues solamente se desea realizar la operación una vez. En caso contrario, la supresión de dicho bucle habilitaría a proceso para realizar continuamente toda la operación.

3.3.4. Destino

El módulo destino es prácticamente idéntico a fuente, salvo porque en vez de enviar datos los recibe. Una vez recibida la información, muestra el resultado de la operación por pantalla, Véanse sus archivos:

Destino.h:

```
#include "systemc.h"
SC_MODULE( destino )
{
	//Señales básicas
	sc_in_clk		reloj;
	sc_in< bool >		reset;

	//Comunicación con proceso
	sc_in< bool >		data_valid;
	sc_out< bool >		data_ack;
	sc_in< sc_int<32> >	data_in;

	//Proceso
	void fuente();

	//Constructor
	SC_CTOR( destino )
	{
		SC_CTHREAD( destino, reloj.pos() );
			watching( reset.delayed() == false );
	}
};
```

Destino.cpp:

```
#include "fuente.h"
```

```
void destino::destino()
{
	data_ack.write(false);
	wait_until( reset.delayed() == true );                                  while( true )

	{
		wait_until( data_valid.delayed() == true );
		cout << endl; << "El resultado de la operación A + B + C
				es: " << data_in.read().to_int() << endl;
		data_ack.write(true);
		wait_until(data_valid.delayed() == false);
		wait();
		sc_stop();
	}
}
```

3.3.5. Main

Main es el fichero donde esta la descripción del sistema. No se trata de un módulo pues no se construye como tal en un archivo .h, si no que se implementa en un archivo .cpp. Además, no presenta la típica estructura de un módulo si no que se define igual que una función de C/C++ en cualquier archivo .cpp tal y como se aprecia en el código:

```
#include "systemc.h"
#include "fuente.h"
#include "proceso.h"
#include "destino.h"

int sc_main(int ac, char* av[])
{
	/***** Definición de señales *****/
		//Señal de reloj
		sc_clock reloj( "CLOCK", 10, 0.5, 0.0 );

		//Señal de reset
		sc_signal< bool > reset

		//Señales entre fuente y proceso
		sc_signal< bool >		data_valid_1;
		sc_signal< bool >		data_req;
		sc_signal< sc_int<32> >		bus_1;

		//Señales entre proceso y destino
		sc_signal< bool >		data_valid_2;
		sc_signal< bool >		data_ack;
		sc_signal< sc_int<32> >		bus_2;

	/***** Instanciación de módulos *****/
		//Fuente
		fuente FUENTE("FUENTE");
		FUENTE.reloj(reloj);
		FUENTE.reset(reset);
		FUENTE.data_req(data_req);
		FUENTE.data_valid(data_valid_1);
		FUENTE.data_out(bus_1);

		//Proceso
```

```
        proceso PROCESO("PROCESO");
        PROCESO.reloj(reloj);
        PROCESO.reset(reset);
        //Comunicación con FUENTE
        PROCESO.data_valid_fuente(data_valid_1);
        PROCESO.data_req_fuente(data_req);
        PROCESO.data_in_fuente(bus_1);
        //Comunicación con DESTINO
        PROCESO.data_ack_destino(data_ack);
        PROCESO.data_valid_destino(data_valid_2);
        PROCESO.data_out_destino(bus_2);

        //Destino
        destino DESTINO("DESTINO");
        DESTINO.reloj(reloj);
        DESTINO.reset(reset);
        DESTINO.data_valid(data_valid_2);
        DESTINO.data_ack(data_ack);
        DESTINO.data_in(bus_2);

    /***** Inicio Simulación *****/
        sc_start(reloj, -1);

    return 0;
}
```

Inicialmente se ha de incluir la librería de SystemC y todos los archivos .h donde están las declaraciones de los módulos que se deseen incorporar al sistema. Hecho esto, la cabecera de la función main, **int sc_main(int ac, char* av[])**, se incorpora tal cual al archivo .cpp y entre sus llaves, { }, se añade todo el código del main.

Primeramente se define el reloj del sistema con **sc_clock reloj("CLOCK", 10, 0.5, 0.0)** sabiendo que sc_clock es una clase especial de SystemC que disponte de varios constructores para su creación. En este caso, los parámetros que se han suministrado son el nombre que se asignará al módulo reloj ("CLOCK"), el periodo del reloj (10), cómo se reparte el periodo entre los valores 0 y 1 (0.5, mitad y mitad) que se denomina dutty cicle (en este caso 50%) y el tiempo inicial del reloj (0.0).

Después se prosigue con la señal de reset que se crea haciendo **sc_signal< bool > reset** donde sc_signal es una señal genérica que no especifica si es de entrada o de salida, solamente es una señal de tipo boleano en este caso. De igual modo, a continuación se definen las señales que intercomunicarán los módulos fuente y destino (Fig. 2).

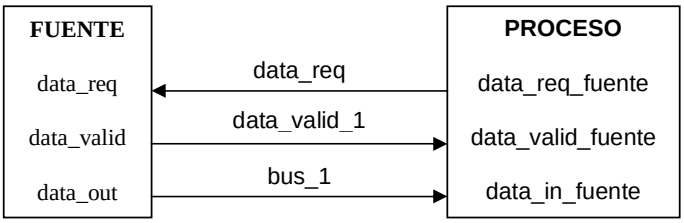

Figura 2. Esquema de interconexión entre los módulos fuente y proceso

Con formato: Fuente: (Predeterminado) Times New Roman

proceso. Fuera de los módulos están las señales que se crean en el main y que son las que realmente efectúan la interconexión. En cambio, dentro de cada módulo están las señales que se declaran en sus correspondientes archivos .h como de entrada o salida,

sc_in o sc_out y que realmente más que como señales pueden verse como descriptores que permiten el acceso a las señales exteriores y posibilitan operaciones como .read o .write.

Volviendo al código del main, en éste se definen todas las señales de interconexión exteriores a los módulos de las figuras 2 y 3 como sc_signal< tipo de señal >.

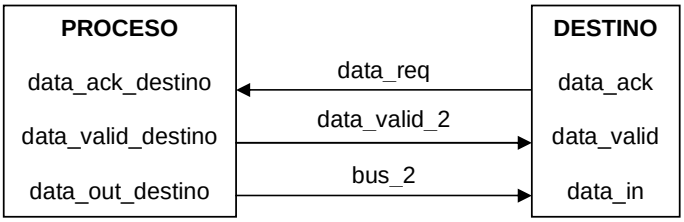

Figura 3. Esquema de interconexión entre los módulos proceso y destino

Hecho esto, se inicia la instanciación de los módulos y la asignación de señales. En primer lugar, se instancia el módulo fuente con la instrucción **fuente FUENTE("FUENTE")**. En ella se crea un módulo del tipo fuente llamado FUENTE y que recibe como parámetro un nombre, "FUENTE" en este caso.

Luego, una vez creado el módulo, se asignan las señales de forma que se correspondan correctamente siguiendo la estructura MODULO.señal interna del módulo(señal exterior creada en el main). Así, en **FUENTE.data_valid(data_valid_1)** se está diciendo que en el módulo FUENTE se vincula su señal data_valid con la señal exterior generada en el main data_valid_1. De ese modo, se han de vincular todas las señales de todos los módulos del sistema, pues en caso contrario el código no funcionará.

Una vez terminados los ajustes de los módulos, se inicia la simulación del sistema con la instrucción **sc_start(reloj, -1).** El parámetro reloj se le pasa para que sea esta señal la que actúe como reloj del sistema. El parámetro -1 indica que la simulación será infinita, sin limite máximo de tiempo, pues si por el contrario en vez de -1 se le pasa un número, éste corresponderá con el número de ciclos a simular.

Finalmente, la última instrucción es para salir del main tal y como se sale de una función que retorna un entero en C/C++.

3.4. Ejemplo de SC_METHOD

Como se ha visto, con el uso de los procesos del tipo THREAD y CTHREAD se puede modelar y simular a un nivel más alto de abstracción utilizando el reloj como elemento sincronizador. Si por el contrario se requiere modelar un circuito a nivel RTL, se han de utilizar los procesos de tipo SC_METHOD con los que se tiene un mayor control combinacional. De nuevo, para ver como funciona se va a estudiar un ejemplo. Que en este caso es muy sencillo puesto que se trata de un pequeño decodificador. Véase su archivo .h:

```
#include "systemc.h"

SC_MODULE(DummyDecoder)
{
	//Señales básicas
	sc_in_clk	HCLK;
	sc_in< bool >	HRESETn;

	//Señales de Entrada
	sc_in< sc_bv<32> >	HADDR;

	//Señales de Salida
	sc_out< bool >		HSEL_Sink;
	sc_out< bool >		HSEL_Source;
	sc_out< bool >		HSEL_DefaultSlave;

	void Decoder();

	SC_CTOR(DummyDecoder)
	{
		SC_METHOD(Decoder);
		sensitive<<HADDR;
	}
};
```

La única diferencia con respecto al proceso de tipo CTHREAD esta en el constructor. Como se ve, en SC_METHOD solamente se ha de pasar como argumento el nombre del proceso ya que no se necesita de ningún reloj para la sincronización. Así mismo, obsérvese como en este caso se utiliza sensitive en el lugar de watching. Sensitive se utiliza para programar al proceso para que se active su ejecución cada vez que se produzca un cambio en las señales añadidas con <<. Véase la funcionalidad de dicho proceso implementada en el archivo .cpp:

```
#include "DummyDecoder.h"

void AHB_DummyDecoder::Decoder()
{
	sc_int<32>	adress;

	adress = HADDR.read();

	if( adress >= SOURCE_ADRESS && adress <= 0x000030FF )
	{
		HSEL_Sink.write(false);
		HSEL_Source.write(true);
		HSEL_DefaultSlave.write(false);
	}
	else if( adress >= SINK_ADRESS && adress <= 0x000040FF )
	{
		HSEL_Sink.write(true);
		HSEL_Source.write(false);
		HSEL_DefaultSlave.write(false);
	}
	else if( adress == MASTER_ADRESS )
	{
		next_trigger(10,SC_NS);
		HSEL_Sink.write(false);
		HSEL_Source.write(false);
		HSEL_DefaultSlave.write(true);
```

```
        }
        else
        {
                HSEL_Sink.write(false);
                HSEL_Source.write(false);
                HSEL_DefaultSlave.write(true);
        }
}
```

En función del valor de la señal de entrada HADDR se habilita una de estas tres señales de salida: SEL_Sink, HSEL_Source y HSEL_DefaultSlave. Los cambios producidos en dichas señales, contrariamente a como pasaba en SC_CTHREAD, son instantáneos ya que no necesitan de un ciclo de reloj para hacerse efectivos.

Cuando se produce un cambio en las señales incluidas en el sensitive, se ejecuta todo el código sin pausa. SC_METHOD no permite la suspensión del proceso pero si la espera por algún motivo. En este caso, la instrucción next_trigger() hace esperar al proceso hasta que acontezca cierto evento o tiempo, pero no suspende al proceso.

No se entrará a estudiar más a fondo este tipo de procesos al no utilizarse para el modelo de comunicación.

3.5. Resumen

Primeramente se ha visto que SystemC es un lenguaje que permite modelar hardware. Para ello estructura el código en módulos y describe las interconexiones entre éstos. Para dichas interconexiones utiliza unas estructuras de datos especiales conocidas como señales. Divide el código de un módulo en dos archivos, un .h y un .cpp. En el primero se declara el módulo con SC_MODULE, se definen las señales, variables, constantes y procesos que se utilizarán y se crea el módulo con el constructor SC_CTOR. Luego de esto se registran uno o varios procesos que se ejecutarán al instanciarse el módulo. Éstos pueden ser de tres tipos, SC_METHOD, SC_THREAD y SC_CTHREAD. En el archivo .cpp únicamente se implementa el código de estos procesos. De ese modo se definen todos los módulos que conformaran el sistema a describir.

Seguidamente, en un archivo a parte se implementa la función main que se encarga de instanciar e interconectar los módulos del sistema. Para ello crea las señales que realmente se utilizarán en la comunicación y las vincula con las señales definidas localmente en cada módulo. Finalmente inicia la simulación del sistema con sc_start(“reloj”, “tiempo de simulación”).

Todo ello se ha visto en un código de ejemplo donde se ha utilizado un proceso del tipo SC_CTHREAD. La principal diferencia entre éste y uno del tipo SC_THREAD es que el primero incorpora la instrucción wait_until mientras que el segundo no. Por tanto, no se verá ningún ejemplo del uso de SC_THREAD al ser muy parecido a SC_CTHREAD.

Para acabar, huelga decir que un código escrito en SystemC se considera sintetizable si su comportamiento es físicamente posible. Por ejemplo, si se tiene un bus de 8 bits, no es posible físicamente enviar 16 bits de información en el mismo ciclo de reloj, aunque en la simulación sí que lo sea. Por ello, se ha de colocar un wait() de manera que se

envíen 8 bits en un ciclo de reloj y 8 más al siguiente ciclo. Por tanto, se ve claro que para que un código sea sintetizable, se ha de programar meticulosamente para que así lo sea.

Finalmente falta decir que SystemC es un compendio de librerías que extienden la funcionalidad del lenguaje C/C++. Construidas a partir de características de C++ como la herencia, incorporan nuevos tipos de datos para expresar mejor el comportamiento del hardware a la vez que posibilitan que el código escrito en SystemC pueda ser mezclado libremente con simple C++.

Referencias

[1] SystemC_2_1_LRM _, disponible con registro gratuito previo necesario en:

https://www.systemc.org/download/5/3/60/104/

[2] SystemC: From The Ground, de David C. Black y Jack Donovan.

Apéndice 4. Microprocesador ARM con bus AMBA

4.1. Descripción del AMBA

AMBA (Advanced Microcontroller Bus Architecture) es un estándar [1] creado por ARM (Advanced Risc Machines) que define tres tipos de bus según las necesidades requeridas por los dispositivos a intercomunicar:

AMBA AHB (Advanced High-performance Bus)

AMBA ASB (Advanced System Bus)

AMBA APB (Advanced Peripheral Bus)

Dónde el primero se utiliza para aquellos dispositivos que requieren altas prestaciones y frecuencias de reloj elevadas y el segundo para módulos de características similares a los primeros, pero con requisitos menos exigentes. El tercer caso está pensado para atender a periféricos de bajo consumo y que no necesitan la potencia de los dos primeros.

AMBA es un bus y, como tal, necesita módulos adicionales a los comunicantes que garanticen el correcto funcionamiento del sistema (Fig. 1). Principalmente estos son un árbitro, un decodificador de direcciones, un controlador de reset, un esclavo por defecto y multiplexores que unan las líneas entre masters y esclavos. Además existe otro elemento importante y que permite la interconexión entre un AMBA AHB o ASB con un APB, el puente APB.

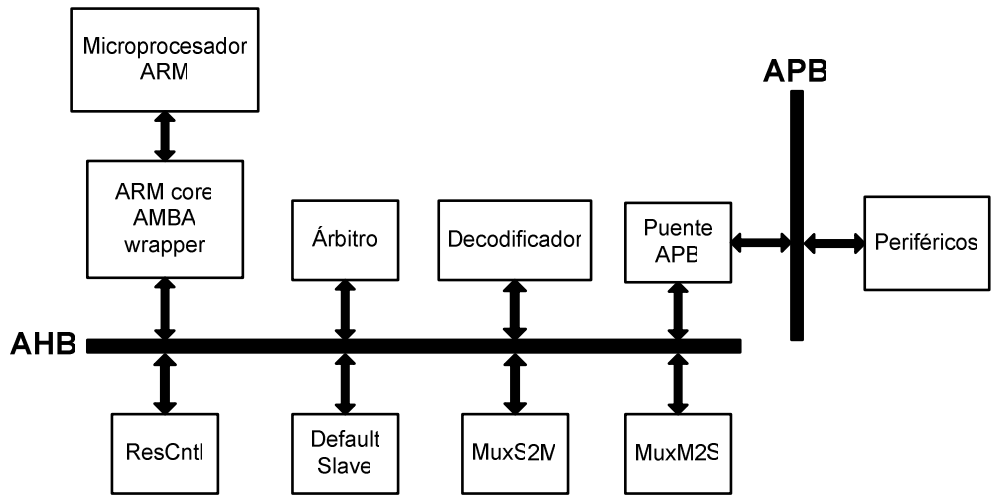

Figura 1. Típico sistema AMBA

La Figura 1 presenta un sistema típico AMBA donde hay interconectado un AHB con un APB a través del puente APB. Este puente actúa como elemento esclavo en el primer bus y como master en el segundo. Del AHB cuelga un microprocesador ARM que utiliza un wrapper (ARM core AMBA wrapper) para comunicarse con el AMBA AHB y que actúa como master del bus. También pueden verse colgados del AHB al árbitro, al decodificador de direcciones, al controlador de reset (ResCntl), al esclavo por defecto (Default Slave) y a los multiplexores que conectan masters con esclavos, MuxS2M y MuxM2S. Al bus APB se conectan periféricos que no requieren grandes prestaciones y a los que el AHB puede acceder a través del puente APB.

En AMBA AHB únicamente puede iniciar la comunicación un dispositivo master, quedando los esclavos relegados a atender las peticiones de éstos. De ese modo, el puente APB al ser esclavo atiende peticiones de los masters en el AHB, mientras que al ser master en el APB, puede iniciar transacciones y, por tanto, hacer peticiones a los periféricos conectados.

Para implementar el protocolo AMBA, existe una serie de señales, distintas para cada bus, cuya interacción posibilita la completitud del estándar y, en definitiva, cómo los dispositivos conectados al bus hacen para transferir datos. A continuación se presentan estas señales de forma resumida mediante tablas para después explicar algunas de ellas con mayor profundidad en los apartados siguientes.

Tabla de Señales AHB

Señal	Descripción
HCLK	Reloj del sistema usado para la sincronización de todos sus elementos.
HRESETnP	Reset del sistema que se activa a la baja.
HADDR[31:0]	Bus de direcciones de 32 bits.
HTRANS[1:0]	Indica el modo de transferencia . Puede ser IDLE, BUSY, NONSEQ y SEQ.
HWRITE	Indica si se desea leer, 0, o escribir, 1.
HSIZE[2:0]	Indica el ancho de banda de la transferencia en bits.
HBURST[2:0]	Indica el tipo de ráfaga actual.
HPROT[3:0]	Añade información de protección y seguridad.
HWDATA[31:0]	Bus de datos donde escribe el master y lee el esclavo.
HRDATA[31:0]	Bus de datos donde lee el master y escribe el esclavo.
HSELx	Selector de esclavo. El esclavo cuya señal HSEL corresponda con HSELx, es notificado con su activación de que un master desea trabajar con él en el momento actual.
HREADY	Señal con la que el esclavo controla la transferencia actual.
HRESP[1:0]	Proporciona información adicional del estado de la transferencia actual.
HBUSREQx	Petición de bus de un master al árbitro.
HLOCKx	Si se activa junto con HBUSREQx, indica que la petición de bus es bloqueante, lo cual quiere decir que el árbitro no le quitará el bus al master hasta que este termine.
HGRANTx	Señal que emplea el árbitro para notificar a un master que le concede el bus.
HMASTER[3:0]	Indica qué master tiene el control del bus en cada momento.
HMASTLOCK	Se activa en caso de conceder una petición de bus bloqueante.
HSPLITx[15:0]	Señal mediante la cual el esclavo indica al árbitro con qué master va a retomar una transacción no finalizada anteriormente.

Tabla de señales ASB

Señal	Descripción
BCLK	Reloj del sistema usado para la sincronización de los elementos de éste.
BnRES	Reset del sistema activo a baja.
AGNTx	Notificación del árbitro al master x de que este dispone del bus.
AREQx	Señal utilizada por los masters para pedir el bus al árbitro.
BA[31:0]	Bus de direcciones del sistema.
BD[31:0]	Bus de datos bidireccional entre maestros y esclavos.
BERROR	Notificación de error en la transferencia actual por parte del esclavo hacia el master con su activación al alta.
BLAST	Indica si la transferencia actual es la última de una ráfaga.
BLOCK	Indica que la transacción que se esta ejecutando es indivisible y que el arbitro va a respetar el control del bus hasta que ésta finalice.
BPROT[1:0]	Señal que aporta información adicional de seguridad a las transacciones.
BSIZE[1:0]	Indica el tamaño de la transferencia que puede ser byte (8 bits), halfword (16 bits) y word (32 bits).
BTRAN[1:0]	Indica el tipo de la próxima transacción que se ejecutará.
BWAIT	Indica si la transferencia actual se ha completado, 1, o si aún quedan ciclos para terminar, 0.
BWRITE	Indica si se desea leer, 0, o escribir, 1.
DSELx	Selector de esclavo. El esclavo cuya señal DSEL corresponda con DSELx, es notificado con su activación de que un master desea trabajar con él en el momento actual.

Tabla de señales APB

Señal	Descripción
PCLK	Reloj del sistema usado para la sincronización de todos los elementos del bus.
PRESETnP	Reset del sistema activo a baja.
PADDR[31:0]	Bus de direcciones de 32 bits.
PSELx	Selector de esclavo del bus de periféricos. El esclavo cuya señal PSEL se corresponda con la señal PSELx del decodificador, es notificado con su activación de que se requieren sus servicios.
PENABLE	
PWRITE	Indica si se desea leer, 0, o escribir, 1.
PRDATA	Bus de datos de lectura.
PWDATA	Bus de datos de escritura.

4.2. El bus AMBA AHB

El bus AMBA AHB está diseñado para comunicar dispositivos de altas prestaciones y rendimiento y de frecuencias de reloj elevadas. Su estructura de bus permite la conexión de hasta 16 módulos maestros, como mínimo ha de haber uno, y de tantos módulos esclavos como espacio de direcciones haya disponible. Aunque dicho espacio de direcciones es de 32 bits, como es de suponer un esclavo abarcará más espacio de memoria que el de un solo bit, por lo que su número será muy inferior a 2^{32}. El resto del sistema (Fig. 3) lo componen el árbitro, el decodificador y los multiplexores necesarios para las conexiones.

Dentro de los 16 posibles módulos master, uno de ellos suele ser el master por defecto. Este dispositivo obtiene el control del bus cuando ningún otro master hace uso de él y se puede configurar su comportamiento, ya sea para realizar tareas específicas dentro del bus o para no hacer nada. De igual modo, también existe un esclavo por defecto, pero en este caso su uso es obligado y su activación se da únicamente cuando algún master intenta acceder a posiciones de memoria erróneas o donde no hay ningún módulo esclavo mapeado. Al detectar este comportamiento, el decodificador activa al esclavo por defecto cuyo funcionamiento se limita a advertir al master de su error colocando la señal HRESP a ERROR si detecta que éste quiere iniciar algún tipo de transferencia (HTRANS igual a SEQ o NONSEQ), o a mantener HRESP a OKAY si la transferencia es IDLE o BUSY.

En lo referente a las transferencias, el protocolo se divide en dos partes (Fig. 2): la fase de direccionamiento y la fase de datos. En la primera se establecen las señales de control y dirección, mientras que en la segunda se ponen los datos en el bus. En las ráfagas mayores de un dato, se inicia la transacción tal y como muestra la figura 2, pero ya en la fase de datos se solapan ambas fases de forma que en cada ciclo se pone el dato k y la dirección relativa al dato k+1 que se enviará en el siguiente ciclo (véase la página 34 para más información).

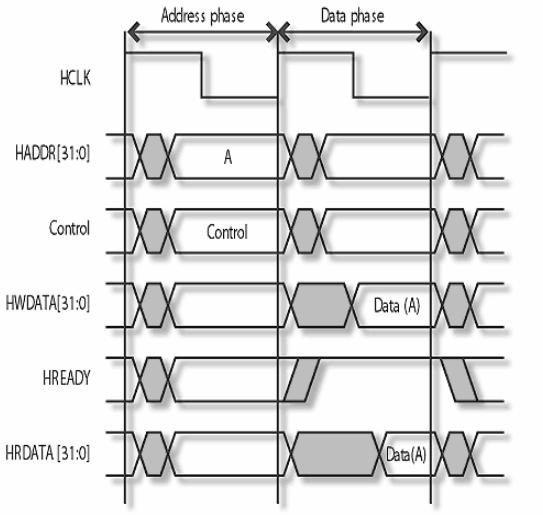

Figura 2. Fases de una transferencia

Las señales de control ayudan al esclavo a discernir qué tipo de transferencia se requiere, el ancho de palabra de ésta y si se utiliza algún tipo de seguridad marcado por la señal HPROT. En cambio, las señales utilizadas en la fase de datos son el bus de datos, las señales de respuesta del esclavo y el bus de direcciones.

La figura 3 muestra el esquema de interconexión de un bus AMBA AHB donde cabe destacar como las señales entre maestros y esclavos no son líneas dedicadas, sino que pasan a través de multiplexores que las reparten en forma de árbol como muestra con mayor claridad la figura 4. Gracias a esto es posible escalar el número de masters y esclavos del bus sin que aumente excesivamente el cableado que los une.

Respecto a las señales de interconexión entre módulos, su interacción hace posible la comunicación entre dispositivos y, por tanto, la funcionalidad del protocolo. En la introducción ya se ha visto una descripción superficial de estas señales, pero ahora se verán en más profundidad aquellas cuyos valores no son únicamente cierto o falso y que pertenecen al AMBA AHB.

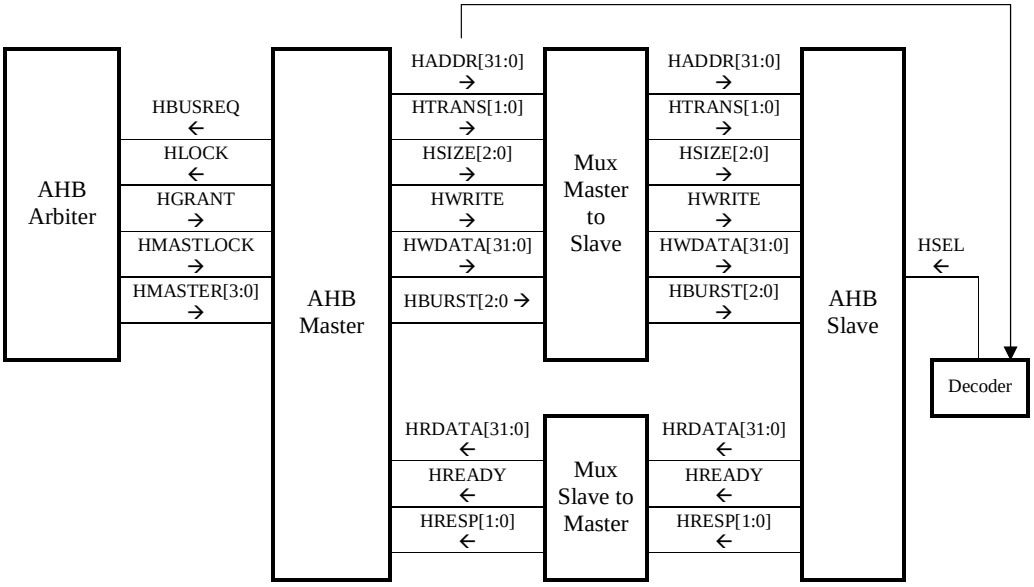

Figura 3. Interconexión de las señales en AMBA AHB con un Master y un Esclavo

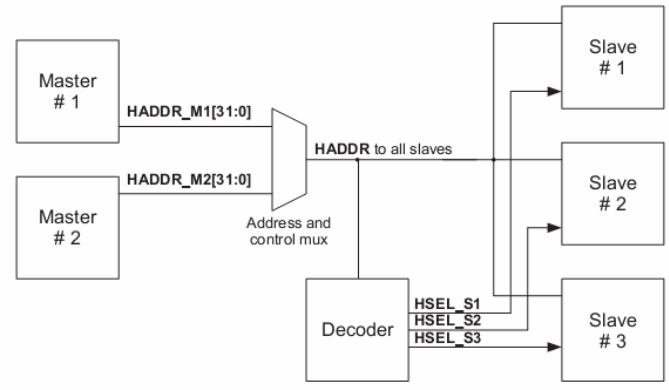

Figura 4. Esquema de la interconexión por multiplexores

HTRANS[1:0]

Señal de dos bits cuyos valores indican el tipo de transferencia que se va a realizar.

Código	Nombre	Descripción
00	IDLE	Indica que no se desea efectuar ninguna transferencia.
01	BUSY	Se utiliza en transferencias de tipo ráfaga para insertar ciclos de espera pudiendo así detener momentáneamente el envío de datos. De ese modo se le notifica al esclavo que se continuará con la ráfaga, pero no inmediatamente, por lo que el esclavo descartará los datos que le lleguen por el bus entendiendo que no son válidos.
10	NONSEQ	Por un lado, indica el inicio de una transferencia de ráfaga, por lo que la dirección y las señales de control no son relativas a transferencias previas. Por el otro lado, se utiliza para indicar que es una transferencia simple, puesto que estas son tratadas como si fuesen una ráfaga de un dato.
11	SEQ	A excepción de la primera, el resto de transferencias que conforman una ráfaga son indicadas con SEQ, de manera que las señales de control son idénticas en toda la ráfaga y las de dirección son relativas a la primera suministrada.

HSIZE[2:0]
Indica el tamaño de los datos que se envían por el bus de datos HWDATA.

Código	Tamaño en bits
000	8
001	16
010	32
011	64

Código	Tamaño en bits
100	128
101	256
110	512
111	1024

HBURST[2:0]
Indica el modo de ráfaga utilizada en la transferencia, teniendo en cuenta que las transferencias simples de un único dato, se consideran ráfagas de uno.

Código	Nombre	Descripción
000	SINGLE	Indica transferencia simple, ráfaga de un dato.
001	INCR	Transferencia en ráfaga de número de datos no especificado.
010	WRAP4	Transferencia en ráfaga de 4 datos.
011	INCR4	Transferencia en ráfaga de 4 datos.
100	WRAP8	Transferencia en ráfaga de 8 datos.
101	INCR8	Transferencia en ráfaga de 8 datos.
110	WRAP16	Transferencia en ráfaga de 16 datos.
111	INCR16	Transferencia en ráfaga de 16 datos.

Para los demás tipos de ráfaga, hay que hacer una distinción entre INCR y WRAP. La primera es una ráfaga incremental con lo que la dirección de memoria se incrementa en cada nuevo envío sumándole a la anterior el tamaño del dato enviado. Respecto a la segunda, el tipo WRAP no se utiliza en el presente modelo por lo que no se explicará su funcionamiento.

HRESP[1:0]
Esta es la señal que los eslavos utilizan para notificar a los masters el estado en el que se encuentran, para lo cual se ayudan con la señal HREADY.

Código	Nombre	Descripción
00	OKAY	Si HREADY = 1 y se activa OKAY, se está notificando que el esclavo esta disponible y que los datos se enviaran o recibirán correctamente. Si por el contrario HREADY vale 0, el esclavo esta introduciendo un ciclo de espera que el master deberá respetar y no enviar datos durante dicho ciclo.
01	ERROR	Notifica que hay un error y que la transferencia no se podrá realizar. Esta respuesta suele darse cuando hay algún fallo crítico en el bus, por lo que el master puede elegir si desea probar suerte y continuar transfiriendo con otros esclavos o si prefiere abortar la operación.
10	RETRY	Indica al master que reinicie la operación puesto que en su momento de inicio el esclavo no estaba disponible, pero ha podido solventar su contratiempo y está listo para transmitir.
11	SPLIT	Notifica que el esclavo no podrá atender la transferencia en ese momento. A fin de no ocupar el bus innecesariamente, el esclavo avisará al árbitro con HSPLITx[15:0] cuando este disponible para que éste conceda el bus al master que transfería cuando se notifico SPLIT, para de ese modo finalizar la transacción pendiente.

Mientras que OKAY se resuelve en un único ciclo de reloj, los casos ERROR, RETRY y SPLIT necesitan dos ciclos para ser tratados. En el caso RETRY (Fig. 5) se ve como el maestro inicia una transferencia, destinada al esclavo mapeado en la posición A, poniendo su dirección en el bus de direcciones HADDR y las señales de control correspondientes. En el siguiente ciclo, envía el dato destinado a A y coloca la dirección del siguiente dato, HADDR = A+4, que pertenece igualmente al esclavo A. Respecto a éste, cuando el master pone la dirección A, él se activa y al no estar listo para transmitir inmediatamente, lo notifica emitiendo HRESP = RETRY y HREADY = falso. Estas señales son vistas por el master un ciclo después de ser emitidas por el esclavo, concretamente cuando el master ya ha enviado el dato A y ha programado la dirección del siguiente dato, HADDR = A+4. Al verlas, el master cancela su actual transferencia y pasa al estado HTRANS = IDLE y queda a la espera de recibir HRESP = RETRY junto con HREADY = cierto. El esclavo puede introducir ciclos de espera haciendo HRESP = OKAY y HREADY = falso hasta que este listo para completar la transferencia, momento en el que activará HRESP = RETRY y HREADY = cierto y se volverá a iniciar la transferencia desde el dato perdido, concretamente el dato.

Para el caso ERROR, el funcionamiento es análogo al de RETRY salvo porque HRESP vale ERROR en vez de RETRY. El master puede elegir entre intentar completar el resto de las transmisiones que tenga con otros esclavos al recibir HREADY = cierto y HRESP = ERROR, o abortar la operación.

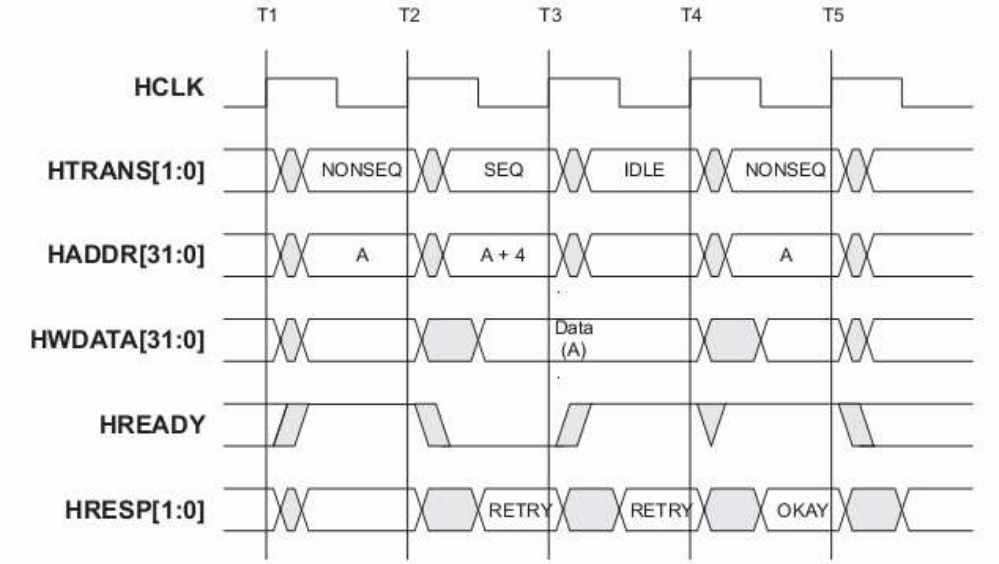

Figura 5. Esquema del funcionamiento de HRESP = RETRY.

Por último (Fig. 6), la señal HRESP = SPLIT presenta un esquema muy parecido a los dos casos anteriores. La primera fase de la respuesta se inicia poniendo HREADY a falso y HRESP = 11 (SPLIT), mientras que la segunda fase concluye la respuesta manteniendo HRESP a 11 y se colocando HREADY a cierto. Cuando el master recibe la segunda fase de la respuesta, deja de trabajar con el esclavo que la ha emitido, bien para continuar con el resto de operaciones que tenga con otros esclavos o para dejar libre el bus a otro dispositivo master. Cuando el esclavo esté preparado para concluir la operación, se lo notifica al arbitro con la señal HSPLITx[15:0] donde activará el bit

correspondiente al master que dejó a medias para que le árbitro le conceda el bus y pueda terminar.

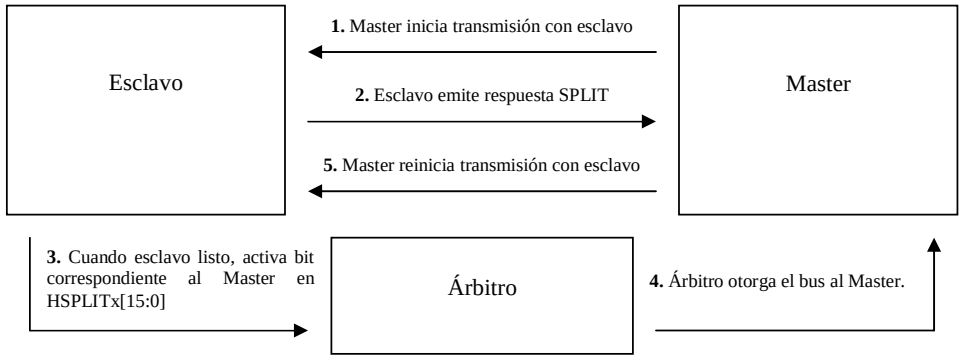

Figura 6. Funcionamiento de una respuesta SPLIT

HMASTER[3:0]
Señal mediante la cual el árbitro notifica al resto del sistema qué master tiene permiso para utilizar el bus. De modo que HMASTER = 001 notifica que el master número uno tiene el bus, HMASTER = 010 notifica que es el master número dos es quien lo tiene, y de igual modo para todos los restantes hasta HMASTER = 111.

HSPLITx[15:0]
Con esta señal el esclavo notifica al árbitro a qué master debe conceder el bus para retomar una transacción interrumpida con HRESP = SPLIT. Cada master se identifica con un único bit, de forma que si se trata del master número tres HSPLIT valdrá 0000000000001000, mientras que si se trata del número cinco valdrá 0000000000100000. De ese modo, el árbitro únicamente ha de aplicar una máscara para saber a qué Master ha de conceder el bus.

4.3. Operaciones en AMBA AHB

4.3.1 Transferencia SINGLE

Una transferencia SINGLE es considerada por AMBA AHB como una transferencia en ráfaga de un único dato. De modo que con este tipo de transferencia se ve claramente la división en fase de direccionamiento y fase de datos, comentada anteriormente (Fig. 2), con la que trabaja el bus AMBA. Su funcionamiento es el siguiente:

En una transferencia SINGLE donde el Master va a enviar un dato, una vez que ha tomado el control del bus inicializa las señales de control y coloca la dirección a dónde desea enviar el dato. Al asignar dicha dirección, el esclavo mapeado en ella es activado por el decodificador y queda capacitado para emitir su estado mediante el uso de las señales HREADY y HRESP. Este estado lo verá el master al siguiente ciclo cuando coloque el dato en el bus HWDATA y podrá así actuar de un modo u otro dependiendo de la respuesta recibida (Fig. 7).

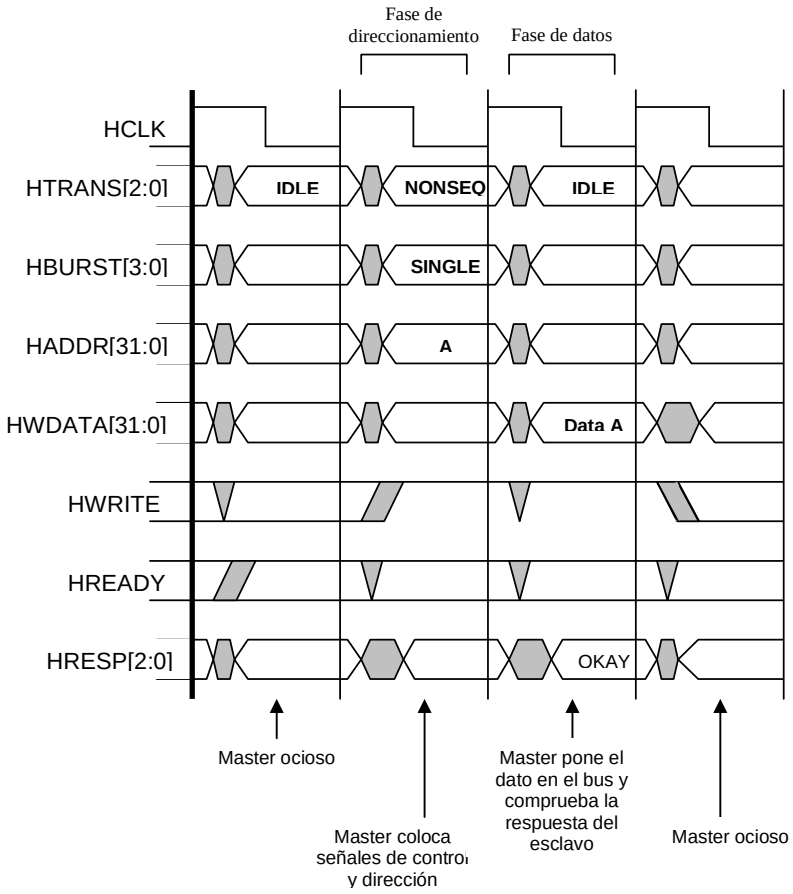

Figura 7. Transferencia SINGLE de escritura

En el caso de una operación de lectura, el funcionamiento es prácticamente análogo al anterior salvo por alguna pequeña diferencia. En primer lugar, el master no pone ningún dato en el bus HWDATA, sino que lee el dato que el esclavo le suministra por HRDATA. En segundo y último lugar, según que respuesta dé el esclavo, el master sabrá si el dato que ha leído es bueno o no (Fig. 8).

En cualquier caso, y como ya se ha comentado anteriormente, el esclavo puede introducir ciclos de espera al master para que este retenga la transacción actual durante otro ciclo (Fig. 9). Además, esta figura muestra también que una transferencia SINGLE necesariamente no significa pedir el bus para transmitir un único dato, sino que se transmite un dato por esclavo.

Como se ve en la figura 9, el master pone las señales de control y de dirección referentes al dato A en la fase de direccionamiento, luego en el siguiente ciclo y ya en la fase de datos, pone o lee, según HWRITE, el dato. Así mismo, se ve como la fase de datos de A queda solapada con la fase de direccionamiento del dato B. En esta segunda fase de direccionamiento se colocan las señales de control y de dirección relativas a B, al tiempo que se lee o se escribe el dato A. Pero en este caso, el esclavo introduce un ciclo de espera, por lo que las señales se mantienen inmutables durante un ciclo, hasta que HREADY vuelve a valer 1 y HRESP[2:0] vale 00, respuesta OKAY. Los ciclos de espera son considerados como parte de la fase de datos.

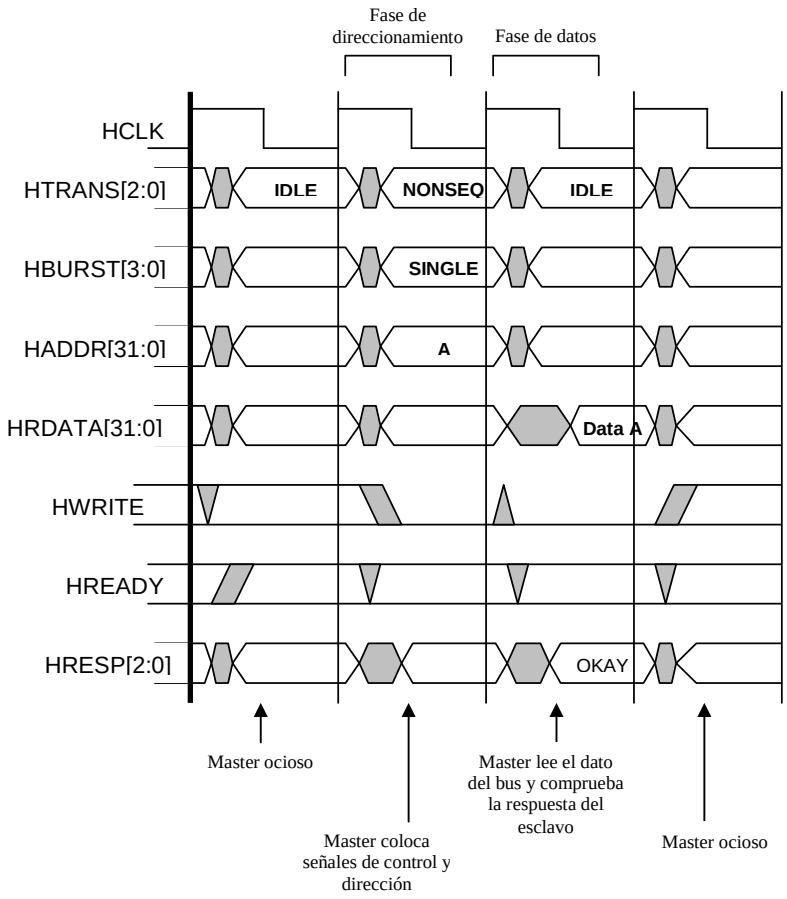

Figura 8. Transferencia SINGLE de lectura

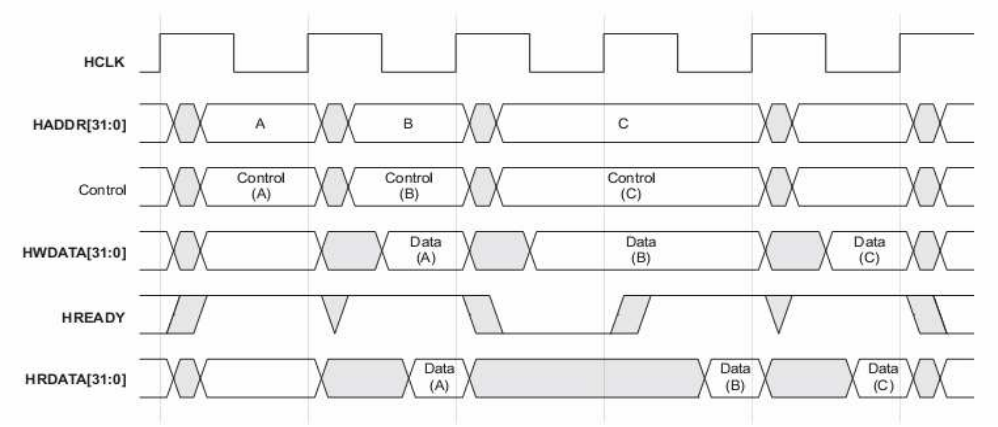

Figura 9. Múltiples transferencias SINGLE con un ciclo de espera introducido por el esclavo en el dato B

Por otro lado, el master también tiene la capacidad de hacer esperar uno o varios ciclos al esclavo. Para hacerlo, activa HTRANS[2:0] = BUSY con lo que el esclavo ha de entender la pausa y esperar hasta que cambie este valor.

4.3.2 Transferencia BURST

Las transferencias en ráfaga, se utilizan cuando hay que enviar o recibir múltiples datos a un mismo esclavo. Existen ocho tipos de ráfaga, contando la ráfaga unaria de tipo SINGLE explicada en el apartado anterior, por lo que se ha de especificar cuál se utiliza en cada momento mediante la señal HBURST[3:0] que puede valer: SINGLE, INCR, WRAP4, INCR4, WRAP8, INCR8, WRAP16 e INCR16.

Dejando de lado el tipo SINGLE, el resto de ráfagas se separan en dos grupos, INCRx y WRAPx. La única diferencia entre estos grupos es cómo hacen el direccionamiento de los datos. En el caso INCRx, se proporciona una primera dirección base que se incrementa según el tamaño del dato enviado. Dicho tamaño está reflejado en la señal HSIZE[3:0] y puesto que todos los datos de la ráfaga tienen las mismas señales de control, el incremento será siempre el mismo para todas las direcciones (Fig. 10). El caso WRAP no se explicará al no formar parte del modelo.

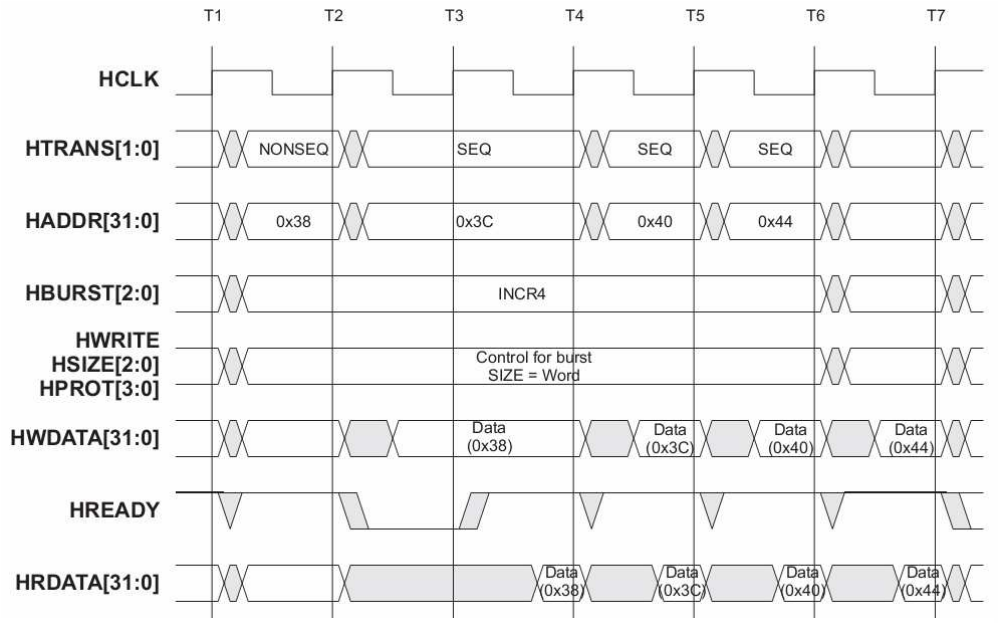

Figura 10. Transferencia en ráfaga incremental de 4 (INCR4).

En la figura 10 puede verse mediante una operación INCR4 como en las transferencias en ráfaga, una vez que se ha enviado el primer dato, las fases de direccionamiento y de datos se solapan. De ese modo, en un mismo ciclo se emite el dato X y la dirección relativa al dato X+1 que se emitirá en el ciclo siguiente. Nótese como, por ejemplo, en T4 se coloca la dirección 0x40 y en T5 se envía o lee el dato relativo a dicha dirección. Así mismo, en T5 a la vez que se envía o recibe el dato se programa la dirección del dato que se leerá o escribirá en T6.

Por otro lado, también puede observarse como el esclavo retiene al master durante un ciclo mediante una respuesta OKAY con HREADY = falso, por lo que los datos enviados el ciclo siguiente por HWDATA para esa dirección, se mantienen en el bus hasta que HREADY vuelva a valer cierto. Contrariamente, el esclavo emitiría el dato válido relativo a la dirección a través del bus HRDATA al volver a activar HREADY a

cierto. Luego es fácilmente deducible que el esclavo se ha encontrado indispuesto y ha parado al master hasta que ha podido volver a estar operativo.

Por último, nótese como en T6 ya no se produce la fase de dirección al haber finalizado la operación. Presumiblemente en dicho tiempo habrá otro master iniciando la fase de dirección de una nueva operación a realizarse a través del bus, tal y como se aprecia en la figura 11. En ella, puede verse cómo se produce una nueva operación justo al finalizar una anterior. El master 2 accede a las señales de control para programar su operación al tiempo que el master 1 finaliza su operación con la lectura o envío del último dato.

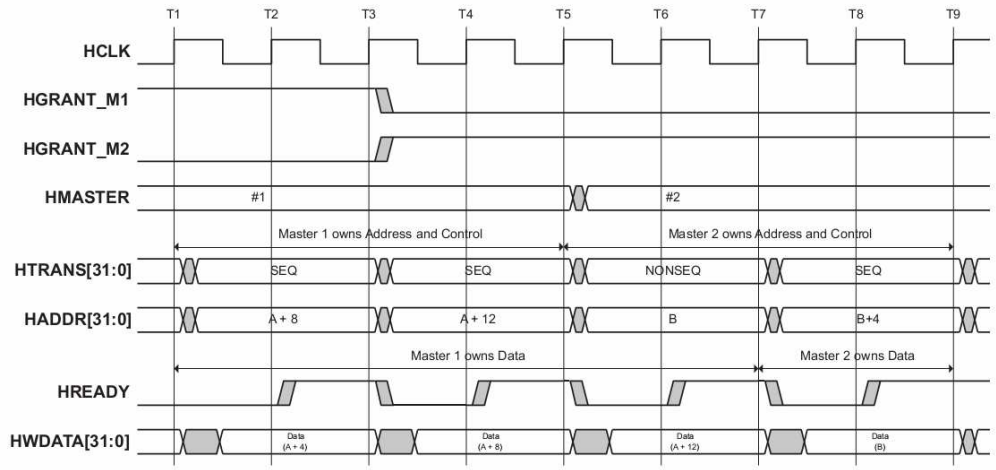

Figura 11. Formas de onda donde se aprecia el cambio de master del bus.

Finalmente queda por decir que todas las operaciones del tipo INCRx presentan el mismo aspecto salvo por el número de datos a emitir.

Referencias

[1] ARM AMBA Specification rev 2.0

Disponible previo registro gratuito en:

http://www.arm.com/securedownloads/user/index.php?subpage=download&action=download&fileid=1&&nomenu=1&notext=1

Apéndice 5. Sistema AMBA AHB en Verilog

5.1. Introducción

Para probar el funcionamiento de los módulos diseñados, el Departamento de Microelectrónica y Sistemas Electrónicos (MiSE) de la Universidad Autónoma de Barcelona facilitó un bus ARM AMBA escrito en Verilog y con la peculiaridad de no implementar la funcionalidad ofrecida por la señal HSPLIT. Por ese motivo no se permiten las transacciones partidas, lo cual implica que cuando un Master adquiere el bus, ostenta su control hasta finalizar la operación. Al margen de esto, el bus incorpora todos los elementos necesarios para su funcionamiento (Fig. 1), además de llevar una ROM añadida. Estos elementos se implementan por separado en archivos .v, extensión utilizada por Verilog.

Este sistema esta preparado para contener un bus ARM AMBA AHB interconectado con un ARM AMBA APB, pero como solamente se requiere el primero, no se incorpora el APB. De hecho, existe toda una estructura jerárquica que compone el sistema (Fig. 2) y que es necesaria para su correcto funcionamiento. Algunos de estos módulos únicamente sirven para instanciar e intercomunicar a los restantes archivos para que puedan trabajar conjunta y coordinadamente.

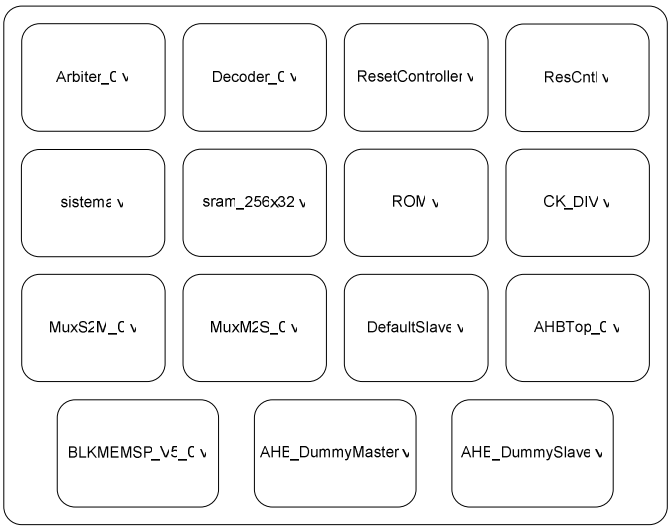

Figura 1. Archivos del sistema escrito en Verilog

La figura 1 muestra todos los archivos .v que componen el bus AMBA. Entre ellos, están los elementos esenciales como el árbitro (Arbiter_0.v), el decodificador (Decoder_0.v), el controlador de reset (ResetController.v), el esclavo por defecto (DefaultSlave.v), un módulo master (AHB_DummyMaster.v) y un módulo esclavo (AHB_DummySlave.v). MuxM2S.v es el multiplexor que transporta las señales de masters a esclavos, mientras que MuxS2M.v hace lo propio en sentido inverso. El resto

de elementos son necesarios para otras funciones como la interconexión e instanciación de módulos.

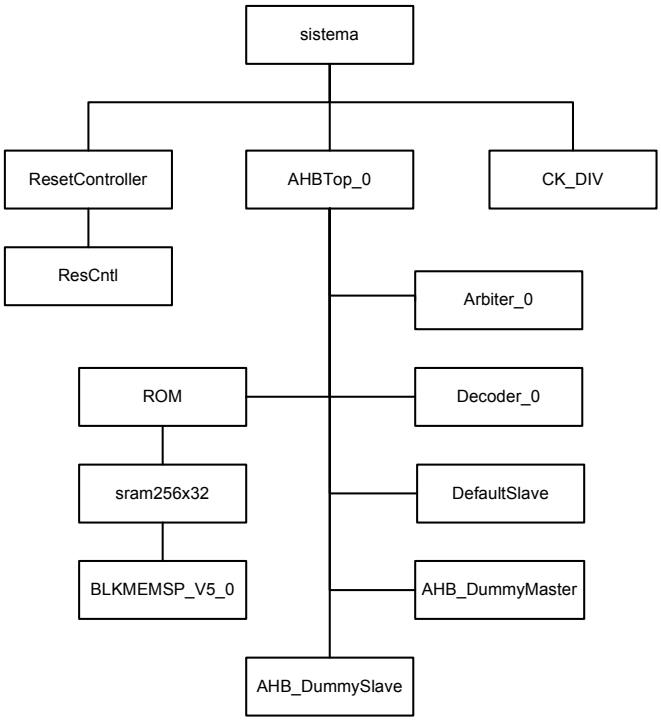

Figura 2. Jerarquía de módulos del sistema escrito en Verilog

A continuación se verá con más detalle cada uno de los archivos descritos en la Figura 1, pero teniendo en cuenta que, al estar escritos en Verilog, no se entrará en detalles sobre el código en si, si no que se mirarán desde el punto de vista del funcionamiento. Además, señales como remap o pause, al no tener ninguna trascendencia sobre el bus AMBA AHB, no se explicarán.

5.2. Desglose del Sistema

sistema.v

Éste se encuentra en la parte más alta de la jerarquía de componentes (Fig 2.) y, como su nombre indica, se ocupa de instanciar todo el sistema. Para ello, crea una serie de los llamados “wires” en Verilog, que son análogos a los “signal” de SystemC. De ese modo, instancia los tres módulos siguientes en la jerarquía, AHBTop_0, ResetController y CK_DIV, y los comunica entre ellos. Si se hubiera incorporado un bus AMBA APB al sistema, éste también se instanciaría y comunicaría desde aquí, así como cualquier elemento externo a los buses AHB y APB.

Sistema tiene así mismo señales de entrada y de salida (Fig. 3) necesarias para incluirlo en circuitos mayores, de forma que otro módulo superior lo instanciaría y lo pondría a trabajar conjuntamente con otros en un sistema mayor. De ese modo, la jerarquía iría aumentando hasta conformar todo un circuito electrónico dónde sistema sería un módulo más.

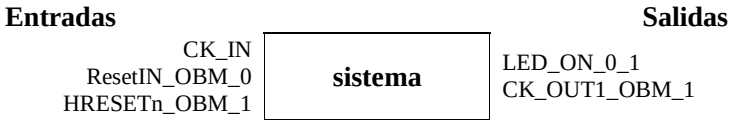

Figura 3. Esquema de las entradas y salidas del módulo sistema

Así, por ejemplo, ResetIN_OBM_0 es una señal de reset exterior que sistema enlaza con la señal ResetIN del módulo ResetController y CK_IN es el reloj del sistema que se propaga a todos los elementos para que funcionen coordinadamente.

CK_DIV.v

Se trata de un divisor de reloj que recibe la señal de reset y un reloj de entrada y proporciona dos relojes de salida (Fig. 4), uno con la misma frecuencia del de entrada y otro con la mitad. El motivo de este módulo es el de proporcionar una frecuencia de reloj elevada con la que trabaja el bus AHB y otra frecuencia más reducida con la que funciona el bus APB.

Figura 4. Esquema de las entradas y salidas del módulo CK_DIV

ResetController.v

El controlador de reset es necesario en el bus para la sincronización de todos sus elementos mediante la señal de reset, que los hará iniciar su funcionamiento de forma simultánea. Luego, como todos tienen un reloj común, a partir del momento en que reciben el reset funcionarán coordinadamente siempre.

Figura 5. Esquema de las entradas y salidas del módulo ResetController

Las entradas que recibe el módulo (Fig. 5) son un reloj, CLK y dos señales de reset, ResetIN y ResetWDT. Estas dos señales, ambas exteriores al sistema, son dos mecanismos diferentes a través de los cuales se produce un reset en el sistema. El primero, ResetIN, es una señal que proviene de un PIN externo del circuito, por lo que podría ser entre otras opciones, el botón de reset del sistema que el usuario puede pulsar. En cambio, ResetWDT se activa por medio del llamado Watch-Dog Timer (WDT) que es un dispositivo hardware que lanza una señal de reset cuando detecta que el circuito ha tenido un fallo. De ese modo, reseteando el circuito pretende reconducir su funcionamiento a un estado normal y correcto.
La salida, ResetOUT, es la señal de reset resultante y, puesto que se activa a la baja, se programa como la operación ResetIN AND ResetWDT.

ResetController realmente no activa ResetOUT dependiendo de las entradas, sino que instancia a ResCntl, que es un wrapper entre el ResetController y el bus AMBA AHB, y

es quién implementa la máquina de estados del reset y propaga su señal al AMBA AHB.

ResCntl.v

Se trata de un wrapper entre el ResetController y el sistema AMBA AHB. ResCntl implementa la máquina de estados de reset que dará lugar a la activación de la señal HRESETn (activa a baja) dependiendo de la entrada POReset (Fig. 6). POReset viene del módulo ResetController quién le da el valor resultante de la operación ResetIN AND ResetWDT, de manera que cuando alguna de estas dos señales adquiere el valor falso, se activa la máquina de estados para resetear el sistema.

Figura 6. Esquema de las entradas y salidas del módulo ResCntl

AHBTop_0

En este archivo se instancia e interconecta entre si el bus AMBA AHB al completo. Por tanto, se generan todos los "wires" de interconexión del bus y se propaga la entrada de reloj hclk y el hreset_n (Fig. 7) a todos los elementos.

Figura 7. Esquema de las entradas y salidas del módulo AHBTop

Arbiter_0.v

Implementa la funcionalidad del árbitro del bus que se basa en una política de prioridades. Aunque incorpore la señal HSPLIT[3:0], este tipo de transacciones no están soportadas por está implementación del bus.

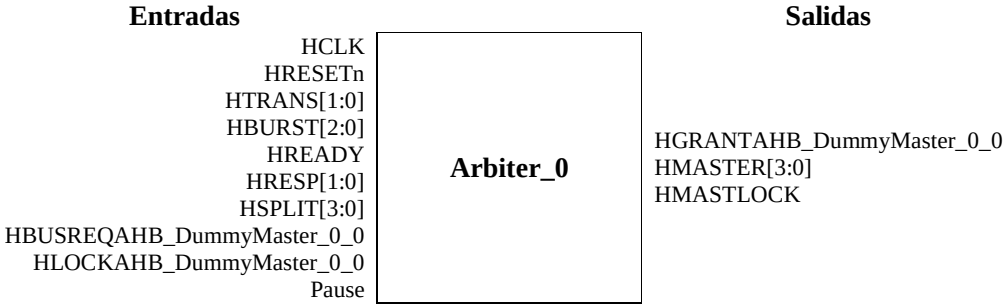

Figura 8. Esquema de las entradas y salidas del módulo Arbiter_0

Su funcionamiento consiste en recibir peticiones de bus por medio de HBUSREQ, procesarlas y una vez escogido quién obtendrá el control del bus, notificárselo activando su HGRANT correspondiente y colocando el número del master en la señal HMASTER[3:0]. Nótese que todos los masters están conectados al árbitro directamente

a través de sus señales HBUSREQ, HLOCK y HGRANT (Fig. 8). De ese modo, habrá una señal de cada por cada master del bus, lo cual es necesario puesto que todos los módulos master deben de poder hacer peticiones de bus cuando lo necesiten y sin tener que pedir permiso al resto, por tanto, simultáneamente. En este caso, hay un único master en el bus y se conecta al árbitro a través de HBUSREQAHB_DummyMaster_0_0, HLOCKAHB_DummyMaster_0_0 y HGRANTAHB_DummyMaster_0_0.

Respecto al resto de señales de entrada, son compartidas por todos los módulos master sin conflicto alguno ya que únicamente el master que controle el bus estará habilitado para escribir en ellas. El árbitro utilizará la información contenida en estas señales para ver en qué estado se encuentra la transferencia actual y poder así decidir cuando otro master pasará a ocupar el bus.

Para el caso de las señales de salida compartidas, HMASTER[3:0] y HMASTLOCK, puesto que únicamente son de lectura para los masters, no hay ningún problema ni necesidad de exclusión mutua.

Decoder_0.v

Este archivo contiene el código que implementa la funcionalidad del decodificador de direcciones del bus, para lo cual crea el módulo Decoder_0 (Fig. 9). Su funcionamiento consiste en leer la dirección de memoria contenida en la señal HADDR y activar el dispositivo esclavo mapeado en dicha dirección activando su señal HSEL correspondiente y desactivando la del resto. En caso de recibir una dirección errónea o donde no hay ningún dispositivo mapeado, el decodificador activa al esclavo por defecto.

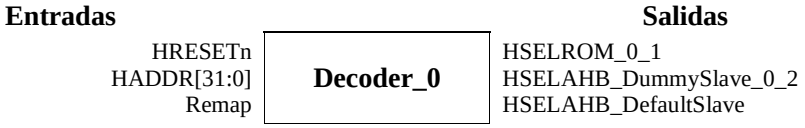

Figura 9. Esquema de las entradas y salidas del módulo Decoder_0

Como solamente el master que controle el bus puede escribir HADDR[31:0], esta señal es compartida por todos los módulos master del bus, mientras que las señales de activación HSEL son individuales para cada dispositivo esclavo. En esta implementación del bus hay una memoria ROM (HSELROM_0_1) y un esclavo sencillo (HSELAHB_DummySlave_0_2), además del esclavo por defecto (HSELAHB_DefaultSlave).

MuxM2S.v

En el presente archivo se implementa el código que actúa como multiplexor de las señales de los módulos master hacía los esclavos, para lo cual utiliza la señal HMASTER como selector del multiplexor para habilitar el paso de las señales de uno u otro master. Además, MuxM2S también implementa la funcionalidad del master por defecto del bus que actúa únicamente cuando ningún otro master tiene su control. Éste master se limita a poner todas las señales a 0 y programar HTRANS = IDLE.

En las entradas (Fig. 10), ha de haber todas las señales *_DummyMaster_* repetidas tantas veces como masters tenga el bus y con numeraciones distintas, *_DummyMaster_0_0, *_DummyMaster_0_1, etcétera… Por tanto, al haber únicamente una copia de estas, la _0_0, queda claro que el presente bus está preparado para un solo master, además del master por defecto

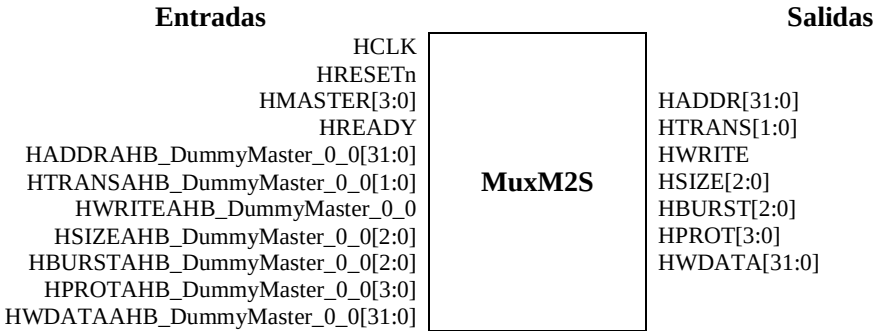

Figura 10. Esquema de las entradas y salidas del módulo MuxM2S

MuxS2M.v

Éste es el multiplexor que conecta las señales de los esclavos a los masters, señales de lectura de los masters y de escritura de los esclavos. Para multiplexar, MuxS2M mira cual es el esclavo seleccionado por el decodificador, aquel que tenga su HSEL = cierto, y propaga sus señales a los masters del bus.

En este caso, se aprecia como sí que hay señales repetidas pero que provienen de distintos esclavos (Fig. 11), lo cual denota la existencia de múltiples de ellos en el bus. Concretamente hay tres esclavos, el AHB_DummySlave, la ROM y el esclavo por defecto DefaultSlave.

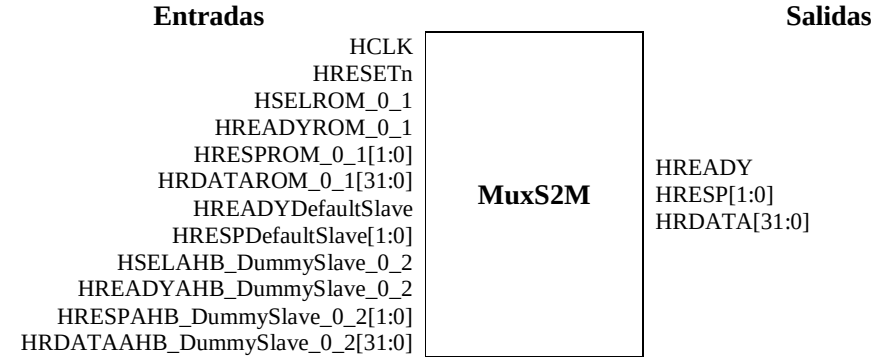

Figura 11. Esquema de las entradas y salidas del módulo MuxS2M

AHB_DummyMaster.v

Éste archivo contiene la implementación de un master sencillo cuyo funcionamiento carece totalmente de importancia para el presente trabajo. La única finalidad que tiene este módulo es la de estar correctamente interconectado en el bus (Fig. 11), para de ese modo poder sustituirlo por el master desarrollado en SystemC.

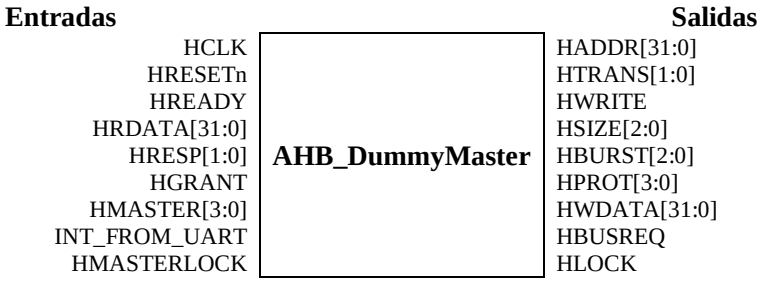

Figura 11. Esquema de las entradas y salidas del módulo AHB_DummyMaster

AHB_DummySlave.v

Su explicación es idéntica al del archivo anterior.

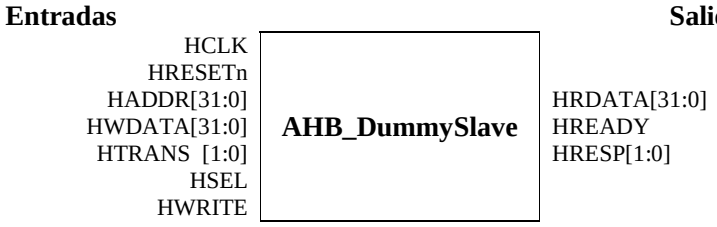

Figura 12. Esquema de las entradas y salidas del módulo AHB_DummySlave

DefaultSlave.v

La estructura de sus señales de entrada y de salida (Fig. 13) es prácticamente la misma que la del AHB_DummySlave, con la diferencia de que no incorpora los buses de datos HWDATA y HRDATA, ni el bus de direcciones.

Figura 13. Esquema de las entradas y salidas del módulo DefaultSlave

sram256x32.v

sram256x32.v es un wrapper que junto con la librería XilinxCoreLib proporciona la posibilidad de utilizar una memoria sram. Pero al tratarse de código propietario de Xilinx y al no utilizarse en el desarrollo del proyecto, no se va a detallar con más profundidad.

ROM.v

Éste archivo Verilog permite integrar el módulo sram256x32 de Xilinx como un esclavo del bus, para lo cual actúa como wrapper entre ambos elementos. Sus señales de entrada y de salida son las mismas que las de los demás esclavos del bus (Fig. 14).

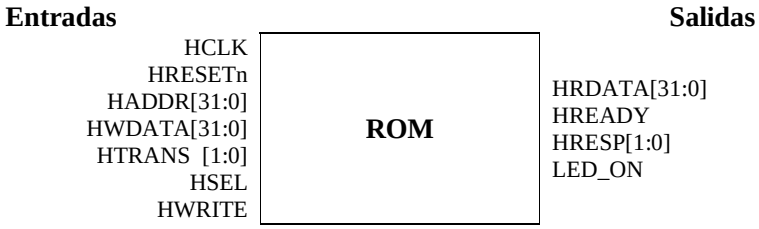

Figura 14. Esquema de las entradas y salidas del módulo ROM

BLKMEMSP_V5_0.v

También éste es un código propietario de Xilinx y que tiene que ver con la memoria sram. De hecho, es el módulo sram256x32 quien instancia a BLKMEMSP_V5_0. Por lo tanto, de nuevo no se entrará en más detalles sobre éste módulo pues tampoco se utiliza en el modelo de comunicación propuesto.

Apéndice 6. SystemC – Verilog

6.1. Verilator: compilador de Verilog en SystemC

Verilator [1] es un software gratuito que, entre otros lenguajes, compila código sintetizable Verilog dentro de código SystemC. Funciona bajo entornos Linux y requiere la instalación de SystemC, SystemPerl, VerilogPerl y Perl 5.6.1 o superior. Puede obtenerse fácilmente en la Web, donde también se encuentra una extensa documentación.

Según la documentación, la utilización de SystemPerl hace que el código final en SystemC sea más rápido de escribir y de ejecutar. Esto es debido a que SystemPerl es un dialecto SystemC que estandariza la interconexión entre las celdas y los pins del circuito descrito. Por otro lado, Verilator optimiza el código para que su compilación con gcc sea notablemente más rápida. De hecho, en la documentación hay un ejemplo donde se pasa de 4 horas de compilación a 7 minutos, claro está utilizando técnicas de compilación paralela entre diferentes máquinas.

Verilator se instaló en una máquina funcionando bajo el sistema operativo Debian GNU Linux Sarge para probar su efectividad y ver si respondía según las expectativas creadas. Inicialmente se probó un código muy sencillo con la intención de comparar las formas de onda obtenidas en la versión original compilada en Verilog y en la versión compilada con Verilator. Así se verificaría de un modo rápido y fiable si el funcionamiento de este software es correcto y el ejecutable resultante es válido desde un punto de vista estrictamente funcional. Puesto que Verilator traduce el código Verilog a SystemC para poder luego compilarlo fácilmente, también se pretendía echar un vistazo al código generado para ver si éste presenta partes redundantes e innecesarias y hacer una valoración más detallada de la efectividad de dicho software. El siguiente paso, sería el de repetir lo hecho anteriormente con un código más complejo para finalmente probar con el código del AMBA AHB.

Finalmente cabe reseñar que las pruebas con el código sencillo no fueron fructíferas, y tras probar con otra distribución de Linux, Fedora Core 4, con la que se obtuvieron idénticos resultados, se decidió desechar la opción Verilator. El código obtenido en la prueba era liado, carente de estructura y presentaba excesiva redundancia. Además, por más vueltas que se le dio, no se pudo conseguir la generación de un archivo ejecutable con el que comparar resultados. Por ello y puesto que los objetivos del presente proyecto no son los de pelear con una herramienta, se abandonó esta opción de raíz y se decidió hacer esta pequeña mención como referencia.

6.2. ModelSim como herramienta para la coverificación

Antes de empezar, cabe reseñar que el entorno de trabajo de ModelSim presenta muchas opciones y posibilidades a los desarrolladores, pero éste escrito no pretende ser un manual completo de dicho software, por lo que solo da una visión superficial de la herramienta y muy encarada a la consecución del presente proyecto. Aquellos que estén

interesados en profundizar la información aquí contenida, pueden visitar la Web del programa [2] donde obtendrán manuales y tutoriales del mismo.

6.2.1. Introducción

ModelSim es un entorno de simulación y verificación para los lenguajes Verilog, VHDL, SystemC y System Verilog, con la característica que permite trabajar con todos ellos a la vez. De ese modo, es capaz de integrar distintos archivos escritos en dichos lenguajes en un mismo proyecto y realizar una simulación conjunta del sistema. Para ello, primero se crea un proyecto y luego se añaden los archivos con el código fuente. Después se crea una librería de trabajo donde se compilarán todos los elementos y que por defecto se llama "work". Acto seguido, se compilan los elementos en dicha librería y por último se simula el sistema al completo. Finalmente, con los resultados obtenidos y el sistema de depuración incluido en ModelSim, un programador puede encontrar los errores y pulir los fallos hasta obtener un código plenamente funcional.

ModelSim puede utilizar las librerías de trabajo de dos formas distintas. Una manera es a modo local con lo que utiliza las librerías para contener el código compilado del sistema, mientras que otra forma es la de utilizar la librería a modo de almacén de recursos estáticos. Así, dicha librería puede contener partes de código que utilice el sistema y que no estén desarrolladas en él, como por ejemplo para utilizar código de un fabricante, como por ejemplo Xilinx, ó para que un programador desarrolle su propia librería de recursos que más tarde podrá utilizar en sus diseños.

La principal arma de depuración de ModelSim es la simulación. Con ella se pueden monitorizar las señales del sistema y ver sus formas de onda en cualquier instante de tiempo. Además se pueden realizar comparaciones entre formas de onda, así como añadir puntos de parada, "breakpoints", en el código fuente y analizar la cobertura de éste. Con todo ello, un programador puede verificar el comportamiento de su sistema y realizar los ajustes necesarios para su correcta sintonización.

6.2.2. Creación de un nuevo Proyecto

En la Figura 1 puede verse cómo es el entorno de desarrollo ModelSim. La ventana inferior es una consola desde donde se introducen comandos al programa como se verá más tarde. En la ventana superior izquierda inicialmente está la pestaña "Library" que contiene las librerías de desarrollo que se pueden utilizar. Por último, en la ventana derecha se abrirán los archivos de código fuente para poder editarlos, así como las formas de onda resultantes de la simulación.

El primer paso, comienza por iniciar un nuevo proyecto mediante la opción File → New Project, tras la cual se obtiene una nueva ventana flotante con varias opciones (Fig. 2). En ésta se asigna un nombre al proyecto y a la librería por defecto, además de especificar la ruta dónde se almacenarán los archivos. Cumplimentado esto, se vuelve al entorno de desarrollo inicial dónde ha aparecido una nueva pestaña en blanco y titulada "Project" en la ventana izquierda. Además, también aparece una nueva ventana flotante (Fig. 3) que permite añadir archivos al proyecto actual. La opción "Create New File" abre una nueva ventan flotante (Fig. 4) que permite especificar el formato del nuevo archivo a crear desde cero. En cambio, "Add Existing File" abre un dialogo de exploración que permite encontrar el archivo a añadir.

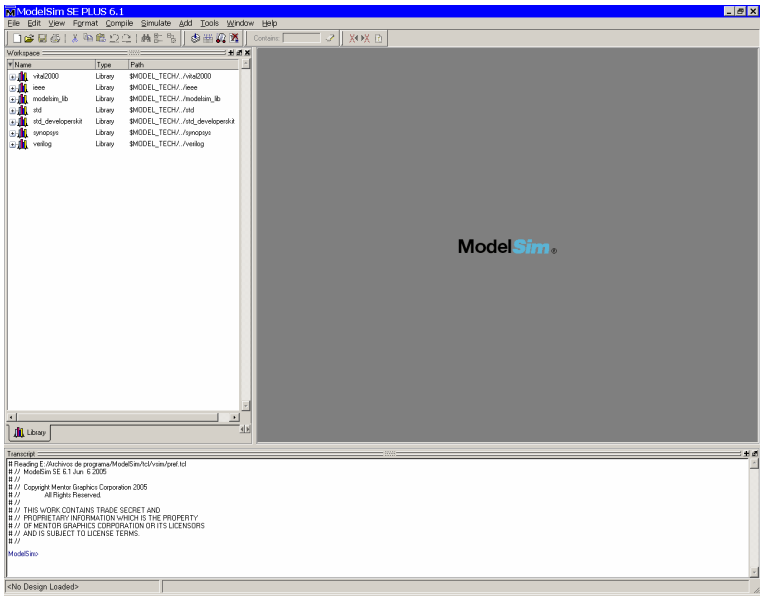

Figura 1. Entorno de desarrollo ModelSim

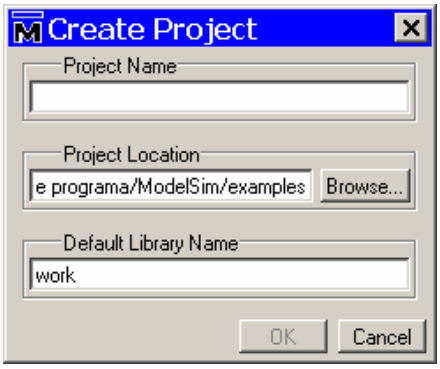

Figura 2. Creación de un nuevo proyecto

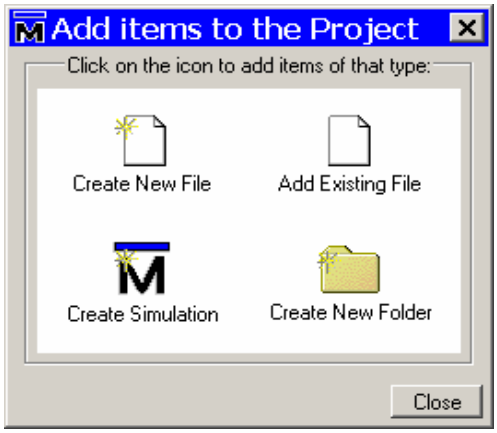

Figura 3. Nueva ventana flotante aparecida tras la creación del proyecto.

Con estas dos opciones se añade o se crea el código fuente que implemente la funcionalidad del sistema hasta obtener la figura 5, donde se observa el proyecto del modelo de comunicación mediante el bus ARM AMBA AHB. En la ventana de la derecha se puede editar el código de los distintos archivos del proyecto contenidos en la ventana izquierda, pestaña "Project", y que conforman el proyecto en si.

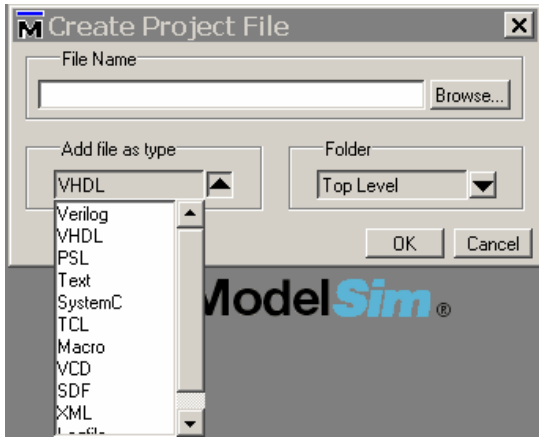

Figura 4. Nuevo archivo, opción "Create New File"

De éste modo, paso a paso se va dando forma al proyecto hasta que todo el código este acabado y solo quede iniciar las pruebas y, por tanto, la simulación.

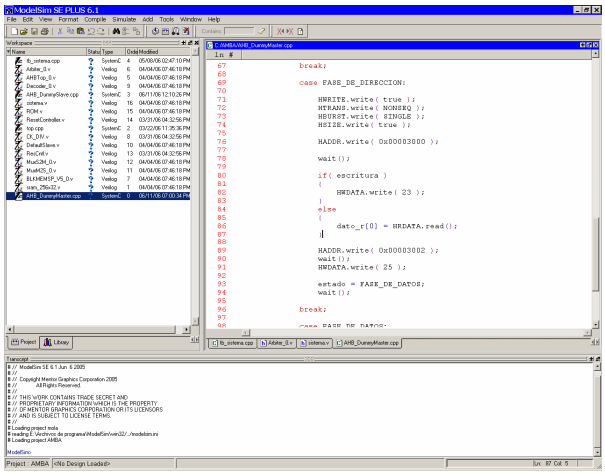

Figura 5. Entorno de desrrollo ModelSim con el proyecyo AMBA cargado

6.2.3. Preparativos para la Simulación

Para poder simular el sistema, antes se ha de compilar el código que lo compone en la librería del proyecto. Para ello, se pueden utilizar comandos introducidos en la ventana de consola o las herramientas gráficas de ModelSim. Si se conocen los comandos o se dispone de un archivo script que los contenga, siempre es mejor optar por la primera vía debido a su eficacia y sencillez. De todos modos, no se va a entrar en detalle en ninguna de las dos opciones al haber utilizado un archivo de script para la compilación el proyecto del modelo de comunicación.

Así pues, una vez compilado el código se puede iniciar la simulación. De hecho, al compilar el código ModelSim lo ejecuta automáticamente y al terminar, aparecen nuevas pestañas en la ventana superior izquierda (Fig. 6). La más importante es la pestaña sim, donde aparece una estructura en árbol del sistema al completo y mediante la cual se puede navegar en la jerarquía hasta encontrar el módulo deseado, en este caso AHB_DummyMaster0 implementado en los archivos AHB_DummyMaster.h y .cpp.

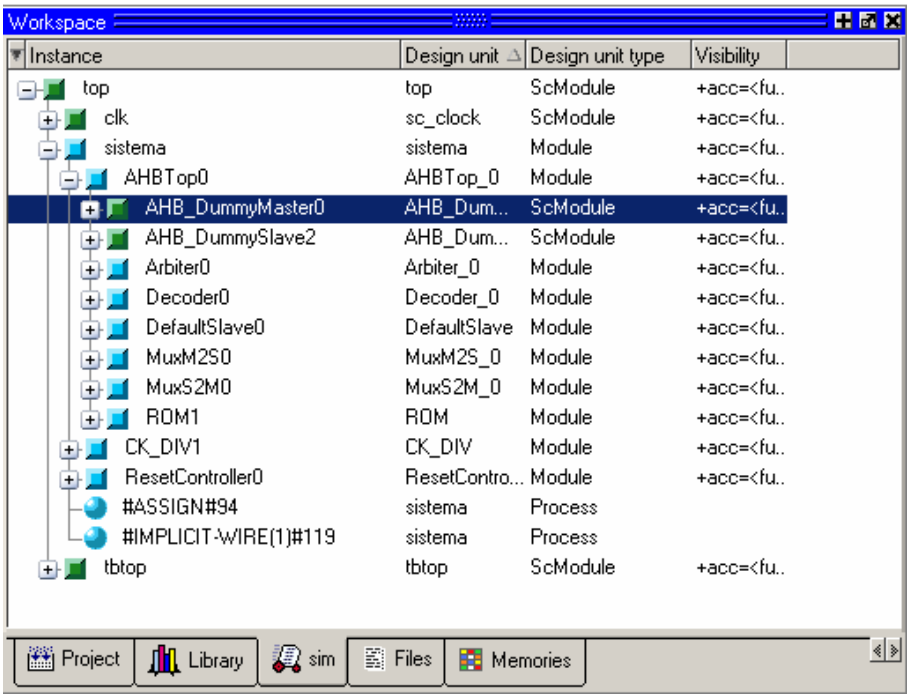

Figura 6. Pestaña sim resultante de la compilación y ejecución del código del proyecto.

Si se hace un clic derecho sobre el módulo, aparece un menú contextual que permite mediante la opción Add → Add to Wave, añadir todas las señales para que sean monitorizadas (Fig. 7). Al añadir dichas señales, aparece una nueva pestaña en la ventana derecha y, como todas las pestañas, se puede desanclar de la ventana derecha para formar una nueva ventana (Fig. 8) y que contiene las señales a monitorizar. De ese modo se pueden añadir todas las señales cuyas formas de onda se desea supervisar.

En la figura 8 se pueden ver las señales añadidas a la monitorización en la primera columna. En la segunda están los valores de éstas en un cierto instante de tiempo, mientras que la tercera es el espacio habilitado para las formas de onda. Cuando se ha terminado de añadir todas las señales deseadas, se pueden personalizar los nombres, el formato de los datos y los colores de las mismas. Para ello, se utiliza el menú contextual que aparece al hacer un clic derecho con el ratón sobre la señal a personalizar. Concretamente, con la opción “Properties…” de éste menú aparece una nueva ventana con múltiples pestañas donde se puede hacer todo esto de forma trivial, al ser las opciones muy sencillas, hasta obtener la presentación deseada. Finalizado esto, es aconsejable grabarlo todo con la opción “File → Save…”. De ese modo, se genera automáticamente un script llamado por defecto wave.do que contiene todos los comandos necesarios para obtener la presentación personalizada de las señales escogidas para ser monitorizadas.

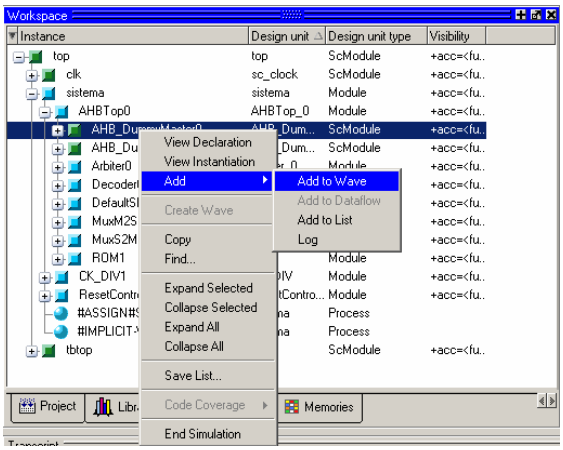

Figura 7. Menú contextual que permite añadir señales para ser monitorizadas

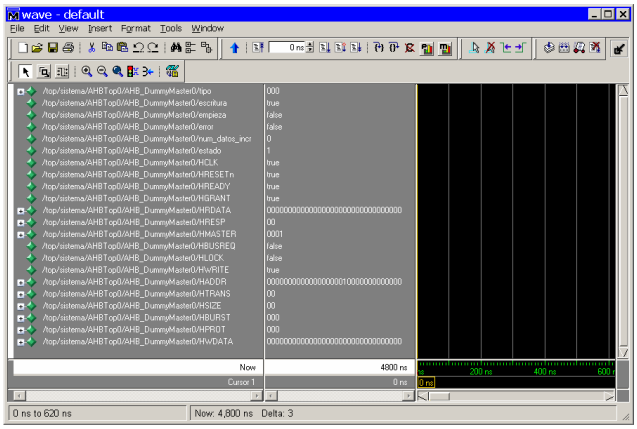

Figura 8. Pestaña de formas de onda desanclada para formar una nueva ventana.

Terminado con todo esto, se ha de reiniciar la simulación para poder ver las formas de onda ya que inicialmente al añadir las nuevas señales no se reflejan sus valores. Para ello, en la ventana principal de ModelSim, se escoge la opción “Simulate → Run → Restart…” con lo que se reinicia la simulación del sistema, pero no será hasta que se haga clic sobre la opción “Simulate → Run → Run -all” que se volverá a repetir la simulación del sistema y se obtendrán las formas de onda.

Por último y a modo de consejo, es recomendable editar el archivo wave.do para copiar todos sus comandos en el script de compilación y ejecución. De ese modo, cada vez que se haga un cambio en el código y se recompile, automáticamente se simulará y se podrá estudiar su salida e incluso compararla con alguna otra gracias a la utilidad de comparación de formas de onda de ModelSim.

Referencias

[1] Sitio Web Oficial: http://www.veripool.com/verilator.html
[2] Sitio Web Oficial: http://www.model.com/resources/resources_manuals.asp

Apéndice 7. Conclusiones

7.1. Extensión de funcionalidad

7.1.1. Introducción

El modelo de comunicación a través del bus ARM AMBA AHB tiene una pega que limita su implantación en cualquier sistema, solamente los dispositivos master tienen la capacidad de iniciar una operación por el bus. Además, un elemento master únicamente puede trabajar con un esclavo, por lo que la operación de master a master no está permitida. Por tanto, si se requiere una comunicación que pueda ser iniciada desde cualquiera de los dos extremos comunicantes, el modelo de comunicación actual no lo permite.

Para resolver dicho inconveniente, se propone de forma teórica como podría el modelo adaptarse a tal eventualidad y posibilitar que se pueda iniciar una comunicación desde cualquiera de los dos extremos.

La principal dificultad que presenta ésta propuesta es la de no salirse del protocolo ARM AMBA AHB. Se ha de hallar el modo de modificar el modelo para obtener la funcionalidad deseada sin que se pierda la compatibilidad con AMBA. Para ello y como se verá después, la solución propuesta es la de aprovechar la capacidad de un dispositivo esclavo de introducir ciclos de espera a un elemento master que solicita una operación con dicho esclavo.

7.1.2. Propuesta de Solución

Como se ha visto, en un bus ARM AMBA AHB es el módulo master quien inicia la comunicación con un dispositivo esclavo, por tanto, solamente queda buscar la manera de que el esclavo pueda también iniciar la comunicación. Para ello, se propone que éste mantenga las señales HREADY a falso y HRESP a OKAY con lo que programa una espera para el master. Esta espera se mantendrá mientras el esclavo no tenga necesidad de comunicar nada. Luego el master estará permanentemente consultando el estado de dichas señales y, si HREADY cambia a cierto, querrá decir que el esclavo desea transmitirle información. Por tanto, viendo el cambio en la señal, el master inicia una transferencia de lectura para obtener los datos, pero dicha transferencia se ha de hacer a ciegas puesto que a priori no se conoce el volumen de datos a recibir. Así pues, el master aguantará la transferencia hasta que el esclavo vuelva a emitir un ciclo de espera, lo cual hará entender al master que ha finalizado el envío de información.

En sentido contrario, si el master no tiene datos que emitir, permanecerá con la señal HWRITE a falso. El esclavo estará constantemente pendiente del valor de ésta señal y si detecta que cambia a cierto, entenderá que el master desea enviarle información. Consecuentemente el esclavo activará HREADY a cierto notificando su disposición a recibir información y el master podrá iniciar con total normalidad una transferencia por el bus.

Nótese que con el mecanismo propuesto no se vulnera el AMBA en ningún momento salvo porque el master y el esclavo monopolizan el bus. Pero ello no es una vulneración del protocolo puesto que el que un esclavo no retenga a un master más de 16 ciclos es únicamente una recomendación del protocolo, no una imposición. Además, esta comunicación está pensada únicamente entre dos elementos, por lo que no tiene sentido que no utilicen constantemente el bus ya que nadie más ha de hacerlo.

Finalmente, en las figuras 1, 2 y 3 puede observarse el esquema de funcionamiento de la propuesta. En la primera se ve el sistema en su estado base o de espera, donde ningún elemento requiere una transmisión y ambos monitorizan las señales del otro esperando un cambio que notifique un inicio de operación.

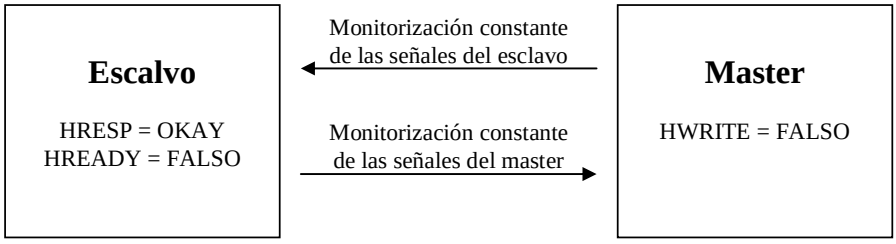

Figura 1. Estado base del sistema. Ningún elemento ha de transmitir por el bus.

En el caso de una transferencia de esclavo a master, el primero cambia el valor de HREADY a cierto para notificar su deseo de emitir. Al ver esto, el master inicia una operación de lectura con el esclavo para recibir la información que éste desea enviarle. Finalmente, el esclavo notifica el fin del envío emitiendo un ciclo de espera, HRESP = OKAY y HREADY = FALSE, con lo que el master finaliza la operación y se retorna al estado base.

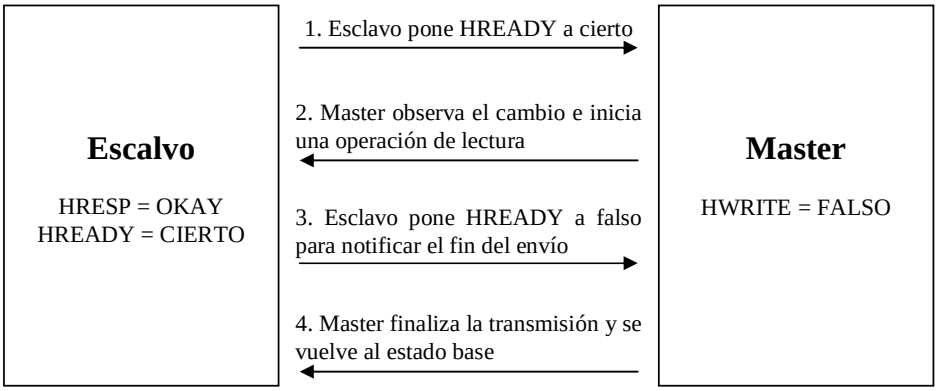

Figura 2. Operación de envío de datos de esclavo a master requerida por el primero.

En última instancia, un master puede operar normalmente con el esclavo al estar habilitado a iniciar la comunicación siempre que desee. Pero para que el esclavo sepa de sus intenciones, primeramente deberá el master colocar su señal HWRITE a cierto para en el siguiente ciclo iniciar la operación. De ese modo, al ver el cambio en la señal, el

esclavo modificará HREADY de falso a cierto para de ese modo notificar su disposición a la operación y cumplir con el protocolo AMBA.

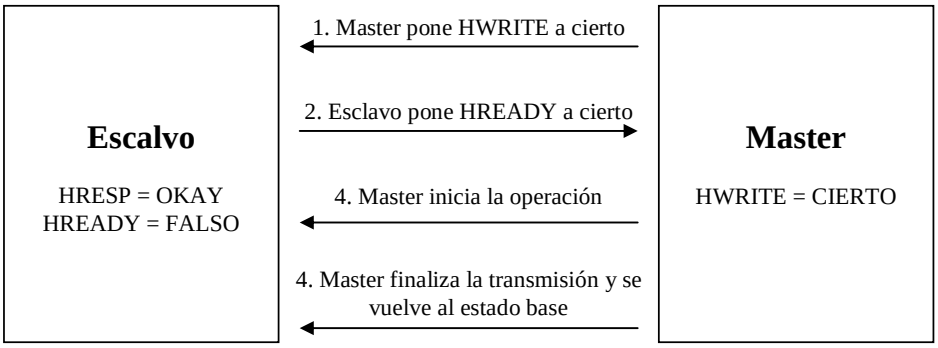

Figura 3. Operación de envío de datos de master a esclavo

7.2. Resultados, Conclusiones y Futuro

Finalmente, gracias a las pruebas en ModelSim, ha quedado probado que el modelo de comunicación funciona perfectamente acorde al protocolo AMBA AHB, siempre teniendo en cuenta las limitaciones del presente trabajo. Después se ha sacado el código del modelo fuera del bus cedido por el MiSE y se ha probado en un sistema diseñado para el cálculo de la transposición de matrices, donde nuevamente las pruebas efectuadas han resultado satisfactorias y se corrobora el buen funcionamiento del modelo.

Vistos los resultados obtenidos, se puede concluir con que le modelo de comunicaciones a través del bus AMBA AHB propuesto en el presente trabajo funciona correctamente.

Respecto al futuro, la supresión de las limitaciones del presente modelo es plausible, así como la implementación de la extensión de funcionalidad propuesta en el apartado anterior. De igual modo, la utilización del modelo para las comunicaciones en un sistema complejo, como por ejemplo el MPEG descrito en la Figura 1, podría ser un buen reto de futuro.

En la Figura 1, el módulo Codificador necesitaría implementar ambos procesos del modelo, master y esclavo. De otro modo, cuando tenga los datos codificados, en vez de poder enviarlos directamente al módulo Destino utilizando al master del bus, tendrá que ser MPEG quién primero lea los datos del Codificador y luego los envíe a Destino.

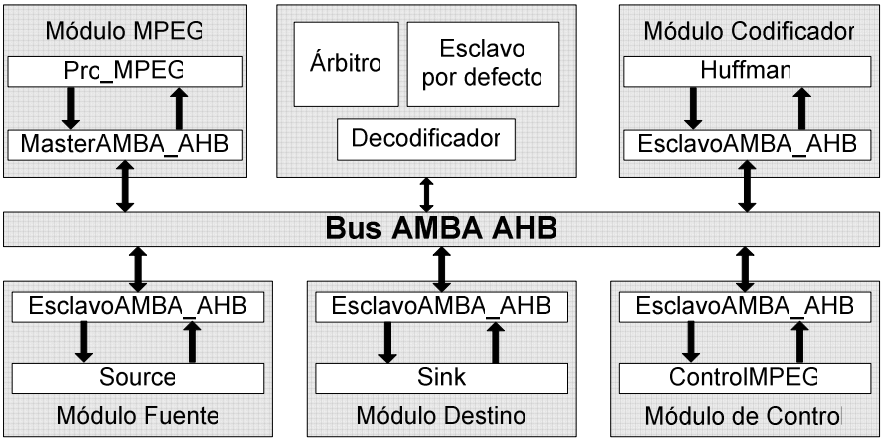

Figura 1. Diseño de las comunicaciones a través del modelo de un sistema MPEG

Apéndice 8. Código fuente

8.1. Model de comunicación

8.1.1. Master del bus

Archivo .h

```
#include "systemc.h"
#include "constantes.h"

SC_MODULE(AHB_DummyMaster) {
	//Variables locales
	int estado;		/Indica el estado actual de la máquina de estados
	int num_datos_incr;	//Número de datos a enviar en una ráfaga INCR

	//Variables para la sincronización entre procesos del módulo
	int cod_operacion;	//Indica la existencia de errores en la operación
	bool empieza;		//Indica el inicio/fin de una operación
	bool escritura;		//Tipo de operación: 1 escritura, 0 lectura
	bool buffer_lleno;	//Indica que datos_in se ha llenado
	bool datos_requeridos;		//Pide datos a los procesos del programador
	sc_bv< 3 >		tipo;		//Indica tipo de ráfaga --> señal HBURST
	sc_bv< 2 >		ancho;		//Indica tamaño de datos --> señal HSIZE
	sc_bv< 32 >		direccion_base;		//Proporciona la dirección base del esclavo con el que operar
	sc_fifo< sc_bv< BUS_LENGTH > > datos_in;		//Cola de datos recibidos a través del bus
	sc_fifo< sc_bv< BUS_LENGTH > > datos_out;		//Cola de datos a enviar a través del bus

	/*** Entrada ***/
	//Señales básicas
	sc_in_clk		HCLK;		//Reloj del sistema
	sc_in< bool >		HRESETn;		//Reset del sistema

	//Señales del árbitro
	sc_in< bool >		HGRANT;		//Si 1 bus concedido, si 0 no
	sc_in< sc_bv<4> >	HMASTER;		//Indica qué master controla el bus

	//Señales del esclavo
	sc_in< bool >		HREADY;		//Indica la disponivilidad del esclavo para operar
	sc_in< sc_bv<BUS_LENGTH> > HRDATA;		//Bus de datos de esclavo a master
	sc_in< sc_bv<2> >	HRESP;		//Indica estado del esclavo
	/**** Fin Entrada ****/

	/**** Salida ****/
	//Señales hacia el árbitro
	sc_out< bool >		HBUSREQ;		//Petición de bus
	sc_out< bool >		HLOCK;		//Si 1 indica petición bloqueante

	//Señales hacia el esclavo
	sc_out< bool >		HWRITE; //Si 1 --> operación de escritura, si 0 --> operación de lectura
	sc_out< sc_bv<BUS_LENGTH> >		HADDR;		//Bus de direcciones
	sc_out< sc_bv<2> > HTRANS;	//Tipo de transferencia: IDLE, BUSY, NONSEQ, SEQ
	sc_out< sc_bv<2> > HSIZE;	//Tamaño de los datos que circulan por el bus
	sc_out< sc_bv<3> > HBURST;	//Tipo de ráfaga: SINGLE, INCR, INCR4, INCR8, INCR16
	sc_out< sc_bv<3> > HPROT;	//Proporciona seguridad en los datos (No se utiliza)
	sc_out< sc_bv<BUS_LENGTH> > HWDATA;		//Bus de datos de master a esclavo

	/**** Fin Salida ****/
	//Constructor
	SC_CTOR(AHB_DummyMaster)
	{
		//Proceso de comunicación por AMBA AHB
		SC_CTHREAD( MasterAMBA_AHB, HCLK.pos() );
			watching( HRESETn.delayed() == true );
	}
	void MasterAMBA_AHB();
};
```

Archivo .cpp

```
void AHB_DummyMaster::MasterAMBA_AHB()
{
  int k;                                //Multiplicador de direcciones
  int j;                                //Contador de la longitud de la transferencia a realizar
  bool retry;                           //Indica si se recive respuesta retry
  bool leido;
  bool enviado;
  int incr_direccion;    //Incremento a aplicar a la dirección base
  sc_int< BUS_LENGTH > datos_out_tmp; //Contenedor temporal de los datos de salida
  sc_int< BUS_LENGTH > datos_in_tmp;   //Contenedor temporal de los datos de entrada

  //Programa el estado inicial
  estado = RESET;

  while( true )
  {
    switch( estado )
    {
      case RESET:
      //Ha de inicializar las señales de control y dirección a valores válidos

        //Control
        HTRANS.write( IDLE );
        HSIZE.write( BYTE );
        HWRITE.write( true );
        HBURST.write( SINGLE );

        //Dirección
        HADDR.write( MASTER_ADRESS );

        //Programa el siguiente estado
        estado = ESPERA_ORDENES;

        //Espera hasta fin de reset --> HRESETn = 1
        wait_until( HRESETn.delayed() == false );

      break;
      //Fin del estado RESET

      case ESPERA_ORDENES:
      //Mantiene el proceso a la espera de peticiones de transmisión

        if( empieza )
        //Si recibe orden de iniciar operación
        {
          //Comprueba si el tipo de operación es correcto
          if( tipo == SINGLE )        j = 1;
          else if( tipo == INCR )     j = num_datos_incr;
          else if( tipo == INCR4 )    j = 4;
          else if( tipo == INCR8 )    j = 8;
          else if( tipo == INCR16 )   j = 16;
          else
          {
            cod_operacion = 1;//Código tipo de ráfaga erronea
            empieza = false;            //Finaliza operación
          }

          if( empieza )
          //Se prosigue con la operación
          //Calcula el incremento a aplicar en cada nuevo dato a la dirección base (HADDR)
          //También comprueba si el ancho de palabra a utilizar es correcto
          {
            if( ancho == BYTE )       incr_direccion = 1;
            else if( ancho == HALFWORD ) incr_direccion = 2;
            else if( ancho == WORD ) incr_direccion = 4;
            else if( ancho == WORD2 ) incr_direccion = 8;
            else if( ancho == WORD4 ) incr_direccion = 16;
            else if( ancho == WORD8 ) incr_direccion = 32;
            else if( ancho == WORD16 ) incr_direccion = 64;
            else if( ancho == WORD32 ) incr_direccion = 128;
            else //Ancho de palabra incorrecto
            {
              cod_operacion = 2;
```

```
      empieza = false;
    }

    if( empieza )
    //Inicializa variables locales y de sincronización
    {
     k = 0;
        retry = false;
        cod_operacion = 0;
        buffer_lleno = false;
        datos_requeridos = false;

        //Pide el bus
        HBUSREQ.write( true );

        //Programa nuevo estado
        estado = ESPERA_ARBITRO;
    }
   }
  }
  wait();

break;
//Fin estado ESPERA_ORDENES

case ESPERA_ARBITRO:
//Espera hasta que el árbitro le concede el control del bus --> HGRANT = 1

  if( HGRANT.read() == true )
//Si árbitro concede el bus
  {
    //Deja de pedir el bus pues ya lo tiene
    HBUSREQ.write( false );

    //Programa el nuevo estado
    estado = FASE_DE_DIRECCION;
  }
  else
  //Sigue la espera hasta obtener el bus
  {
    wait();
  }

break;
//Fin estado ESPERA_ARBITRO

case FASE_DE_DIRECCION:
//Primera fase del protocolo de comunicación consistente en programar
//las señaeles de control y la dirección con la que trabajar

  //Control
  HSIZE.write( ancho );
  HBURST.write( tipo );
  HTRANS.write( NONSEQ );
  HWRITE.write( escritura );

  //Dirección
  HADDR.write( direccion_base.to_int() + (incr_direccion * k) );
  k++;
  wait();
  leido = true;
  enviado = true;
  //Programa el nuevo estado
  estado = FASE_DE_DATOS;

break;
//Fin estado FASE_DE_DIRECCION

case FASE_DE_DATOS:
//Segunda fase del protocolo de comunicación. Se envían/reciven los datos
//En el caso de transferencias de ráfaga > 1, se solapa con la fase de dirección

  /*** ESCRITURA/LECTURA DE DATOS (FASE DE DATOS) ***/
  if( escritura )
  //Envío de datos de Master a Esclavo
  {
```

```
 if( (HREADY.read() == true && HRESP.read() == OKAY && k != j+1) || k == 1 )
 //Solo se envían datos si el esclavo responde HREADY = cierto y HRESP = OKAY,
 //ó si es el primer ciclo donde el esclavo aún no ha tenido tiempo de emitir su estado
 {
  if( datos_out.num_available() > 0 || retry )
  //Si hay datos que enviar
  {
  if(!retry) datos_out_tmp = datos_out.read();
  else retry = false;

  //Envía el dato
  write(datos_out_tmp);
  enviado = true;
 }
 else
 //Si no hay datos que enviar
 {
  //Pide datos
  datos_requeridos = true;
  enviado = false;
  }
 }
}
else
//Envío de datos de Esclavo a Master
{
 if(retry) retry = false;
 if( k != 1 && HREADY.read() == true && HRESP.read() == OKAY )
 //Si k = 1, el esclavo aún no ha tenido tiempo de emitir los datos, luego solo se han de leer si k != 1
 {
  if( k != 1 )
  {
    if( datos_in.num_free() > 1 )
    {
     if(!leido)      leido = true;
     else
         {
          datos_in.write( HRDATA.read() );
          leido = true;
         }
    }
    else if( datos_in.num_free() == 1 )
    {
     if(!leido)      leido = true;
     else
         {
          datos_in.write( HRDATA.read() ); //Lee el dato
          buffer_lleno = true;
          if( k != j+1 )      leido = false;
         }
    }
  }
 }
 else
 {
    if( datos_in.num_free() > 0 )      leido = true;
    else
    {
     leido = false;
     buffer_lleno = true;
    }
  }
 }

/*** FIN ESCRITURA/LECTURA DE DATOS (FASE DE DATOS) ***/

/*** PROGRAMACIÓN SEÑALES DE CONTROL Y DIRECCION (FASE DE DIRECCION ***/

if( HREADY.read() == true || k == 1 )
//Si k = 1, es el primer ciclo y la respuesta del esclavo no se comprueba pues no ha podido emitirla
//Para el resto de casos mira si HREADY vale cierto
{
 if( HRESP.read() == OKAY || k == 1 )
 //Si va todo bien (respuesta OKAY), ó es el primer ciclo y no hay respuesta válida aún
 {
  if( leido && enviado )
```

```
   {
    if( k < j )
    {
     //Control
     HWRITE.write( escritura );
     HTRANS.write( SEQ );
     HBURST.write( tipo );
     HSIZE.write( ancho );

     //Dirección
     HADDR.write( direccion_base.to_int() + (incr_direccion * k) );
    }
    else if( k == j )
    {
        //Control
        HSIZE.write( BYTE );
        HTRANS.write( IDLE );
        HBURST.write( SINGLE );

        //Dirección
        HADDR.write( MASTER_ADRESS );
    }
    else
    //k == j+1
    {
        //Control
        HSIZE.write( BYTE );
        HTRANS.write( IDLE );
        HBURST.write( SINGLE );

        //Dirección
        HADDR.write( MASTER_ADRESS );

        //Coloca la señal HWRITE a su valor por defecto
        HWRITE.write(true);
        Vacía el bus de datos
        HWDATA.write(0);

        //Programa el siguiente estado
        estado = ESPERA_ORDENES;

        //Notifica que la operación ha concluido satisfactoriamente
        cod_operacion = 0;    //Código de NO error
        empieza = false;      //Finaliza operación
    }
    k++;
 }
  else  HTRANS.write(BUSY);
}
else if( HRESP.read() == eRROR || HRESP.read() == SPLIT )
HRESP == ERROR --> Segunda fase de la respuesta ERROR
//HRESP ==  SPLIT --> Segunda fase de la respuesta SPLIT. Se trata como el caso ERROR
{
    HSIZE.write( BYTE );
    HTRANS.write( IDLE );
    HWRITE.write( true );
    HBURST.write( SINGLE );
    HADDR.write( MASTER_ADRESS );
    HWDATA.write( 0 );

    //Notifica que la operación ha finalizado con error
    cod_operacion = 3;                  //Código de error: abortado
    empieza = false;                                    //Indica final de operacion
    estado = ESPERA_ORDENES;      //Programa el nuevo estado
 }
 else if( HRESP.read() == RETRY )
 //Segunda fase de la respuesta RETRY
 {
    k = 0;
    retry = true;

    //Programa el nuevo estado para reiniciar el envío/recepción de datos
    estado = FASE_DE_DIRECCION;
    break;
 }
} //Fin de if( HREADY.read() == true || k == 1)
```

```
    else
    //HREADY.read() == false
    {
/* HRESP == eRROR --> Primera fase de la respuesta eRROR: el esclavo notifica de un error crítico en el bus o bien es que sencillamente no tiene ni tendrá datos disponibles para el maestro
HRESP == RETRY --> Primera fase de la respuesta RETRY: el esclavo pide que se reinicie la operación al no haber estado preparado para recivir/enviar cuando se le demandó
HRESP == SPLIT --> Primera fase de la respuesta SPLIT: el esclavo notifica al master que no puede trabajar en estos momentos y lo emplaza a realizar uns transacción partida. Al no soportar el modelo este tipo de transacción, se trata como una respuesta eRROR por lo que se abortará la operación
HRESP == OKAY -->Si se trata del último ciclo, se ha puesto en HADDR la dirección del master con lo que el esclavo ha quedado deseleccionado y no podrá trabajar con el bus. Se vuelve a colocar la última dirección válida del esclavo para que pueda proseguir trabajando pués seguramente le falta un último dato que enviar/recivir En todos los casos se ha de esperar a la segunda fase del ciclo de la respuesta para resolver       */

        if( HRESP.read() != OKAY ) HTRANS.write( IDLE );
        if( k == j+1 )
        {
            //Control
            HWRITE.write( escritura );
            HBURST.write( tipo );
            HSIZE.write( ancho );
            if( tipo == SINGLE )     HTRANS.write( NONSEQ );
            else      HTRANS.write( SEQ );

            //Dirección
            HADDR.write( direccion_base.to_int() + (incr_direccion * (k-2)) );
        }
    }

    /* FIN PROGRAMACIÓN SEÑALES DE CONTROL Y DIRECCION (FASE DE DIRECCION) */

     wait();

    break;
    //Fin estado FASE_DE_DATOS
   }
  }
}
```

8.1.2. Esclavo del bus

Archivo .h

```
#include "systemc.h"
#include "constantes.h"

SC_MODULE(AHB_DummySlave)
{
        //Variables locales
        int estado;             //Indica el estado actual de la máquina de estados

        //Variables para la sincronización entre procesos del módulo
        int cod_operacion;      //Indica la operacion terminada
        bool buffer_lleno;      //Notifica que datos_in esta llena
        bool no_datos;                  //Si true se abortarán todas las operaciones de lectura
        bool fin_operacion;             //Notifica el fin de una operación
        bool inicio_operacion;                  //Notifica el comienzo de una operación
        bool datos_requeridos;                  //Petición de datos para una operación de lectura
        sc_fifo< sc_bv<BUS_LENGTH> > datos_in; //Cola de datos recibidos
        sc_fifo< sc_bv<BUS_LENGTH> > datos_out; //Cola de datos a enviar

        /**** Entrada ****/
        //Señales básicas
        sc_in_clk                       HCLK;           // Reloj del sistema
        sc_in< bool >                   HRESETn;        // Reset del sistema

        //Señales del decodificador
        sc_in< bool >                   HSEL;           // Activacion del Esclavo

        //Señales del master
```

```
        sc_in< bool >                   HWRITE;         //Si 1 --> escritura, si 0 --> lectura
        sc_in< sc_bv<BUS_LENGTH> > HADDR;               //Bus de direcciones
        sc_in< sc_bv<BUS_LENGTH> > HWDATA;              //Bus de datos
        sc_in< sc_bv<2> >           HTRANS;             //Tipo de transferencia
        sc_in< sc_bv<3> >           HBURST;             //Tipo de ráfaga
        /**** Fin Entrada ****/

        /**** Salida ****/
        sc_out< bool >                  HREADY;         //Indica la disponivilidad del esclavo
        sc_out< sc_bv<BUS_LENGTH> >HRDATA;              //Bus de datos
        sc_out< sc_bv<2> >          HRESP;              //Indica estado del esclavo
        /**** Fin Salida ****/

        //Constructor
        SC_CTOR(AHB_DummySlave)
        {
                //Proceso de comunicación por AMBA AHB
                SC_CTHREAD( EsclavoAMBA_AHB, HCLK.pos() );
                        watching( HRESETn.delayed() == true );
        }
        void EsclavoAMBA_AHB();
};
```

Archivo .cpp

```
void AHB_DummySlave::EsclavoAMBA_AHB()
{
//Variables locales
int j;
bool buff;
bool no_leas;
bool operacion;
bool sel;

sc_int<BUS_LENGTH> buffer;

//Inicialización de la máquina de estados
estado = RESET;

while( true )
{
   switch( estado )
   //Implementación de la máquina de estados del esclavo del AMBA
   {
      case RESET:
      //Inicialización de las señales y variables

         //Inicializa señales de respuesta del esclavo
         HRESP.write( OKAY );
         HREADY.write( true );

         //Inicializa variables de comunicación entre los procesos del módulo
         cod_operacion = 0;
         fin_operacion = false;
         inicio_operacion = false;
         datos_requeridos = false;

         //Programa el nuevo estado
         estado = ESPERA_MASTER;

         //Espera hasta finalizar reset --> HRESETn == 1
         wait_until( HRESETn.delayed() == false );

      break;
      //Fin estado RESET

      case ESPERA_MASTER:
      //El esclavo queda a la espera de ser seleccionado por algun master

         if( HSEL.read() == true )
         //Si el decodificador activa HSEL
         {
            //Programa respuesta por defecto
```

```
HRESP.write(OKAY);

inicio_operacion = true;
no_leas = false;
   sel = false;

if( HWRITE.read() == true )
//Operación de escritura --> Master envía datos a esclavo
{
    operacion = true;

    if( datos_in.num_free() >= 1 )
    //Si hay espacio para el dato entrante
    {
        HREADY.write(true);
        buff = true;
        j=0;
    }
    else if( datos_in.num_free() == 0 )
    {
        HREADY.write(false);
        lleno = true;
        buff = false;
        j=1;
    }
    //Programa el nuevo estado
    estado = OPERACION;
}
else
//Operacion de lectura --> Master demanda datos a esclavo
{
    operacion = false;

    if( no_datos )
    //Si es sabido que no habrá datos para enviar
    {
        //Inicia la respuesta ERROR
        HRESP.write( eRROR );
        HREADY.write( false );
        wait();
        HREADY.write( true );
        wait();
        estado = ESPERA_MASTER;
        cod_operacion = 3;
        fin_operacion = true;
        inicio_operacion = false;
    }
    else
    {
        if( datos_out.num_available() > 1 )
        //Si hay datos disponibles para ser enviados
        {
            //Envia el dato
            HRDATA.write(datos_out.read());

            //Programa el siguiente estado
            estado = OPERACION;
            HREADY.write(true);
            j=0;
        }
        else
        //Si no hay datos disponibles para enviar
        {
        //No hay datos aún disponibles, pero no se sabe si los habrá más adelante.
              //Por tanto, se inicia respuesta RETRY al no estar el esclavo aún preparado
              //y se piden datos a los procesos del programador

                //Pide los datos al proceso que controla el esclavo del AMBA AHB
                datos_requeridos = true;

                //Introduce ciclo de espera
                HREADY.write( false );
                j=1;

                //Programa el nuevo estado
                estado = OPERACION;
```

```
                }
            }
        }
    }
    wait();

break;
//Fin estado ESPERA_MASTER

case OPERACION:

    if( HSEL.read() == true )
    {
        sel = false;
        if( HTRANS.read() != BUSY )
        {
            if( j == 16 || no_datos )
            //Si se llega a los 16 ciclos de espera, se aborta operación
            {
                //Programa respuesta ERROR
                HRESP.write( eRROR );
                HREADY.write( false );
                wait();
                HREADY.write( true );
                wait();

                //Programa el nuevo estado para atender nuevas peticiones de masters
                estado = ESPERA_MASTER;
                HRESP.write(OKAY);

                //Programa las variables de sincronización entre procesos
                cod_operacion = 3;              //Código de operacion abortada
                fin_operacion = true;           //Indica el final de una operación
                inicio_operacion = false;
            }
            else
            {
                if( HWRITE.read() == true )
                //Operación de envio de datos de Master a Esclavo
                {
                    if( datos_in.num_free() > 1 )
                    {
                        if(buff)
                        {
                            datos_in.write( HWDATA.read() );
                HREADY.write(true);
                j=0;
                        }
                        else
                        {
                            HREADY.write(true);
                            buff = true;
                            }
                    }
                    else if( datos_in.num_free() == 1 )
                    {
                            if(buff)
                            {
                                datos_in.write( HWDATA.read() );
                                buffer_lleno = true;
                           HREADY.write(false);
                                buff = false;
                                j++;
                            }
                            else        buffer_lleno = true;
                    }
                    else if( datos_in.num_free() == 0 )
                    {
                            buffer_lleno = true;
                            HREADY.write(false);
                            buff = false;
                            j++;
                    }
                }
                else
                //Operación de envío de datos de Esclavo a Master
```

```
        {
                if( datos_out.num_available() > 1 )
                {
                    if(!buff)
                    buff = true;
                    HRDATA.write( datos_out.read() );
                    HREADY.write(true);
                    j=0;
                }
                else if( datos_out.num_available() == 1 )
                {
                    if(buff)
                    {
                        HRDATA.write( datos_out.read() );
                        HREADY.write(true);
                        datos_requeridos = true;
                        buff = true;
                        j++;
                    }
                    else     datos_requeridos = true;
                }
                else if( datos_out.num_available() == 0 )
                {
                    HREADY.write(false);
                    buff = false;
                    datos_requeridos = true;
                    j++;
                }
            }
        }
    }
    wait();
}
else
{
   if( j == 0 )
    {
        if( operacion )
        //Si se reciben datos del Master(HWRITE = true), se ha de leer el último dato
        {
                Lee el último dato
                if( !no_leas )
                datos_in.write( HWDATA.read() );
                //Se notifica al proceso utilitario que se han recibido datos
                cod_operacion = 1;    //Código de operación de escritura de Master a Esclavo
            }
            else
            {
                //Vacia el bus de datos
                HRDATA.write( 0 );

              //Notifica el éxito del envío al módulo utilitario del bus
                cod_operacion = 2;    //Codigo de operación de lectura
            }
            estado = ESPERA_MASTER;          //Se programa el nuevo estado
            fin_operacion = true;            //Indica el final de una operación
            inicio_operacion = false;
            wait();
        }
        else
        {
                if(sel)
                {
                        if(operación) cod_operacion = 1;
                        else          cod_operacion = 2;

                        estado = ESPERA_MASTER;    //Se programa el nuevo estado
                        fin_operacion = true;  //Indica el final de una operación
                        inicio_operacion = false;
                }
                else
                {
                        no_leas = true;
                        sel = true;
                }
                wait();
```

```
                        }

                    break;
                    //Fin de estado OPERACION
                }
                //Fin máquina de estados (Estructura SWITCH)
            }
            //Fin bucle While(true)
}
//Fin Proceso EsclavoAMBA_AHB()
```

8.1.3. Constantes.h

```
#ifndef AR_
#define AR_

        // Referente a la matriz a enviar
        #define BLOCK               8               // Tamaño de la matriz de datos a procesar NxN
        #define BUS_LENGTH          32              // Anchura de los buses HWDATA y HRDATA
        // Posiciones de memoria
        #define MASTER_ADRESS       0x00002000      // Master que finaliza en 0x000020FF
        #define SLAVE_ADRESS        0x00003000      // Elave que finaliza en 0x000030FF
        #define ROM_ADRESS          0xF0000000      // ROM que finaliza en 0xF0001FFF

        // Maquina de estados del AHB_Master
        #define RESET                       0
        #define ESPERA_ORDENES              1
        #define ESPERA_ARBITRO              2
        #define FASE_DE_DIRECCION           3
        #define FASE_DE_DATOS               4

        // Maquina de estados del AHB_Slave
        //#define RESET                     0
        #define     ESPERA_MASTER           1
        #define OPERACION                   2

        // Tamaño de las transferencias HSIZE --> Señal de control
        #define BYTE                        0       //8 bits
        #define HALFWORD                    1       //16 bits
        #define WORD                        2       //32 bits
        #define WORD2                       3       //64 bits
        #define WORD4                       4       //128 bits
        #define WORD8                       5       //256 bits
        #define WORD16                      6       //512 bits
        #define WORD32                      7       //1024 bits

        // Respuesta del slave HRESP
        #define OKAY                        0
        #define eRROR                       1
        #define RETRY                       2
        #define SPLIT                       3

        // Señal HBURST
        #define SINGLE                      0
        #define INCR                        1
        #define INCR4                       3
        #define INCR8                       5
        #define INCR16                      7

        // Señal HTRANS
        #define IDLE                        0
        #define BUSY                        1
        #define NONSEQ                      2
        #define SEQ                         3

#endif
```

8.2. Código para las pruebas en ModelSim

8.2.1. Operación SINGLE

8.2.1.1 Prueba 1

```
void AHB_DummyMaster::maestro()
{
	sc_int<BUS_LENGTH> tmp;

	//Crea un fichero de salida para comprobar el resultado de la operación
	ofstream f("Master.txt", ios::out);
	f.close();
	empieza = false;
	wait_until( HRESETn.delayed() == true );   //Espera por reset

	for( int p = 0; p < 1; p++ ) //Se ejecuta una única vez
	{
		//Programa las variables de sincronización
		tipo = SINGLE;				//Ráfaga de un solo dato
		ancho = WORD8;				//Tamaño de palabra de 256 bits
		escritura = true;			//Operación de escritura
		datos_out.write(76);			//Coloca el dato en la cola de salida
		direccion_base = SLAVE_ADRESS;		//Dirección del esclavo
		wait();
		//Inicia la operación
		empieza = true;
		f.open("Master.txt", ios::app);

		//Espera hasta que finaliza operación
		while(true)
		{
			if( !empieza ) break;   //Si final de operación, sale para comprobar el resultado
			else
			{
				if( datos_requeridos )
				{
					f << "Se requieren datos..." << endl << endl;
					datos_out.nb_write(100+p);
					datos_requeridos = false;
					wait();
				}

				if( buffer_lleno )
				{
					datos_in.nb_read(tmp);
					f << "Vaciando buffer... Dato leído: " << tmp
					  << endl << endl;
					buffer_lleno = false;
					wait();
				}
				if( !buffer_lleno && !datos_requeridos ) wait();
			}
		}
		wait();

		switch( cod_operacion )
		{
			case 0:
				f << "Operación completada satisfactoriamente" << endl;
				while( datos_in.num_available() > 0 )
				{
					f << "Se ha recivido el dato/s: " << datos_in.read() << endl;
					wait();
				}
				f << endl;
			break;
			case 1: f << "Error en el tipo de ráfaga" << endl; break;
			case 2: f << "Error en el ancho de palabra" << endl; break;
			case 3: f << "Operación abortada" << endl;
```

```
                }
                //Cierra el fichero
                f.close();
                wait();
        }
        wait(15); //Espera para dar tiempo al esclavo a presentar sus datos en su fichero.
        sc_stop();//Finaliza la simulación
}

void AHB_DummySlave::esclavo()
{
        int num_datos_recividos;
        sc_int<BUS_LENGTH> tmp;
        //Crea el fichero si no existe
        ofstream f("Esclavo.txt", ios::out);
        f.close();
        //Espera hasta fin de reset
        wait_until( HRESETn.delayed() == true );
        no_datos = false;       //Habilita al esclavo para operaciónes de lectura
        f.open("Esclavo.txt", ios::app);

        while(!inicio_operacion)        //Espera hasta que se inicie una operación
                wait();

        while(true)
        {
                if( fin_operacion )     //Si operación finaliza
                {
                        //Crea un fichero para comprobar el resultado de la operación
                        switch( cod_operacion )
                        {
                                case 0: f << "Cod_Operación sin valor" << endl;
                                case 1:
                                        f << "Operación de recepción de datos Completada:";
                                        num_datos_recividos = datos_in.num_available();
                                        f << " Se han recivido " << num_datos_recividos  << " datos" << endl;
                                        f << endl << "Dato/s recivido/s:" << endl;
                                        for(int i = 0; i < num_datos_recividos; i++)
                                        {
                                                f << datos_in.read() << endl;
                                                wait();
                                        }
                                        f << endl;
                                break;
                                case 2: f << "Se ha enviado dato/s correctamente" << endl; break;
                                case 3: f << "Operación abortada" << endl;
                        }
                        //Restablece el valor del fin de operación
                        wait();
                        fin_operacion = false;
                }
                else
                {
                        if( buffer_lleno )
                        {
                                if( datos_in.nb_read(tmp) )
                                {
                                        f << "Vaciando cola de entrada... Dato leído: " << tmp << endl;
                                        buffer_lleno = false;
                                }
                        }
                        if( datos_requeridos )
                        {
                                if( datos_out.nb_write(23) )
                                {
                                        f << "Llenando cola de salida... Dato Volcado: 23" << endl << endl;
                                        datos_requeridos = false;
                                }
                        }
                        wait();
                }
        }
}
```

8.2.1.2. Prueba 2

```
void AHB_DummyMaster::maestro()
{
	sc_int<BUS_LENGTH> tmp;

	//Crea un fichero de salida para comprobar el resultado de la operación
	ofstream f("Master.txt", ios::out);
	f.close();

	empieza = false;
	wait_until( HRESETn.delayed() == true );   //Espera por reset

	wait(16);

	for( int p = 0; p < 1; p++ ) //Se ejecuta una única vez
	{
		//Programa las variables de sincronización
		tipo = SINGLE;				//Ráfaga de un solo dato
		ancho = WORD8;				//Tamaño de palabra de 256 bits
		escritura = true;			//Operación de escritura
		direccion_base = SLAVE_ADRESS;		//Dirección del esclavo
		wait();

		//Inicia la operación
		empieza = true;

		f.open("Master.txt", ios::app);

		//Espera hasta que finaliza operación
		while(true)
		{
			if( !empieza )		//Si final de operación, sale para comprobar el resultado
				break;
			else
			{
				if( datos_requeridos )
				{
					f << "Se requieren datos..." << endl << endl;
					datos_out.nb_write(100+p);
					datos_requeridos = false;
					wait();
				}

				if( buffer_lleno )
				{
					datos_in.nb_read(tmp);
					f << "Vaciando buffer... Dato leído: " << tmp << endl << endl;
					buffer_lleno = false;
					wait();
				}

				if( !buffer_lleno && !datos_requeridos )
					wait();
			}
		}

		wait();

		switch( cod_operacion )
		{
			case 0:
				f << "Operación completada satisfactoriamente" << endl;
				while( datos_in.num_available() > 0 )
				{
					f << "Se ha recivido el dato/s: " << datos_in.read() << endl;
					wait();
				}
				f << endl;
			break;
			case 1: f << "Error en el tipo de ráfaga" << endl; break;
			case 2: f << "Error en el ancho de palabra" << endl; break;
			case 3: f << "Operación abortada" << endl;
		}
```

```
                //Cierra el fichero
                f.close();

                wait();
        }

        //Finaliza la ejecución del código tras 3 ciclos
        wait(15);   //Espera para dar tiempo al esclavo a presentar sus datos en su fichero.
        sc_stop();
}

void AHB_DummySlave::esclavo()
{
        int num_datos_recividos;
        sc_int<BUS_LENGTH> tmp;
        //Crea el fichero si no existe
        ofstream f("Esclavo.txt", ios::out);
        f.close();
        //Espera hasta fin de reset
        wait_until( HRESETn.delayed() == true );
        no_datos = false;       //Habilita al esclavo para operaciónes de lectura
        f.open("Esclavo.txt", ios::app);
        while(!inicio_operacion)            //Espera hasta que se inicie una operación
                wait();

        while(true)
        {
                if( fin_operacion )     //Si operación finaliza
                {
                        //Crea un fichero para comprobar el resultado de la operación
                        switch( cod_operacion )
                        {
                                case 0: f << "Cod_Operación sin valor" << endl;
                                case 1:
                                        f << "Operación de recepción de datos Completada:";
                                        num_datos_recividos = datos_in.num_available();
                                        f << " Se han recivido " << num_datos_recividos << " datos" << endl;
                                        f << endl << "Dato/s recivido/s:" << endl;
                                        for(int i = 0; i < num_datos_recividos; i++)
                                        {
                                                f << datos_in.read() << endl;
                                                wait();
                                        }
                                        f << endl;
                                break;
                                case 2: f << "Se ha enviado dato/s correctamente" << endl; break;
                                case 3: f << "Operación abortada" << endl;
                        }
                        //Restablece el valor del fin de operación
                        wait();
                        fin_operacion = false;
                        f.close();
                }
                else {
                        if( buffer_lleno )  {
                                if( datos_in.nb_read(tmp) )
                                {
                                        f << "Vaciando cola de entrada... Dato leído: " << tmp << endl;
                                        buffer_lleno = false;
                                }
                                else
                                        f << "No se puede vaciar buffer" << endl;
                        }
                        if( datos_requeridos )
                        {
                                if( datos_out.nb_write(23) )
                                {
                                        f << "Llenando cola de salida... Dato Volcado: 23" << endl << endl;
                                        datos_requeridos = false;
                                }
                        }
                        wait();
                }
        }
}
```

8.2.2. Operación BURST

```
void AHB_DummyMaster::maestro()
{
	sc_int<BUS_LENGTH> tmp;

	//Crea un fichero de salida para comprobar el resultado de la operación
	ofstream f("Master.txt", ios::out);
	f.close();

	empieza = false;
	wait_until( HRESETn.delayed() == true );   //Espera por reset

	wait(16);

	for( int p = 0; p < 1; p++ ) //Se ejecuta una única vez
	{
		//Programa las variables de sincronización
		tipo = INCR;                                    //Ráfaga de un solo dato
		ancho = WORD8;                                  //Tamaño de palabra de 256 bits
		escritura = true;                               //Operación de escritura
		direccion_base = SLAVE_ADRESS;          //Dirección del esclavo
		num_datos_incr = 3;
		datos_out.write(23);
		datos_out.write(47);
		datos_out.write(76);
		datos_out.write(101);
		wait();

		//Inicia la operación
		empieza = true;

		f.open("Master.txt", ios::app);

		//Espera hasta que finaliza operación
		while(true)
		{
			if( !empieza )          //Si final de operación, sale para comprobar el resultado
				break;
			else
			{
				if( datos_requeridos )
				{
					f << "Se requieren datos... Dato proporcionado: 100" << endl << endl;
					datos_out.nb_write(100+p);
					datos_requeridos = false;
					wait();
				}

				if( buffer_lleno )
				{
					datos_in.nb_read(tmp);
					f << "Vaciando buffer... Dato leído: " << tmp << endl << endl;
					buffer_lleno = false;
					wait();
				}

				if( !buffer_lleno && !datos_requeridos )
					wait();
			}
		}
		wait();

		switch( cod_operacion )
		{
			case 0:
				f << "Operación completada satisfactoriamente" << endl;
				while( datos_in.num_available() > 0 )
				{
					f << "Se ha recivido el dato/s: " << datos_in.read() << endl;
				}
				f << endl;
			break;
			case 1: f << "Error en el tipo de ráfaga" << endl; break;
			case 2: f << "Error en el ancho de palabra" << endl; break;
```

```
                        case 3: f << "Operación abortada" << endl;
                }

                //Cierra el fichero
                f.close();

                wait();
        }

        //Finaliza la ejecución del código tras 3 ciclos
        wait(15);   //Espera para dar tiempo al esclavo a presentar sus datos en su fichero.
        sc_stop();
}

void AHB_DummySlave::esclavo()
{
        int num_datos_recividos;
        sc_int<BUS_LENGTH> tmp;
        //Crea el fichero si no existe
        ofstream f("Esclavo.txt", ios::out);
        f.close();
        //datos_out.write(15);
        //Espera hasta fin de reset
        wait_until( HRESETn.delayed() == true );
        no_datos = false;       //Habilita al esclavo para operaciónes de lectura
        f.open("Esclavo.txt", ios::app);
        while(!inicio_operacion)            //Espera hasta que se inicie una operación
                wait();

        while(true)
        {
                if( fin_operacion )     //Si operación finaliza
                {
                        //Crea un fichero para comprobar el resultado de la operación
                        switch( cod_operacion )
                        {
                                case 0: f << "Cod_Operación sin valor" << endl;
                                case 1:
                                        f << "Operación de recepción de datos Completada:";
                                        num_datos_recividos = datos_in.num_available();
                                        f << " Se han recivido " << num_datos_recividos << " datos" << endl;
                                        f << endl << "Dato/s recivido/s:" << endl;
                                        for(int i = 0; i < num_datos_recividos; i++)
                                        {
                                                f << datos_in.read() << endl;
                                                wait();
                                        }
                                        f << endl;
                                break;
                                case 2: f << "Se ha enviado dato/s correctamente" << endl; break;
                                case 3: f << "Operación abortada" << endl;
                        }
                        //Restablece el valor del fin de operación
                        wait();
                        fin_operacion = false;
                        f.close();
                }
                else
                {
                        if( buffer_lleno ){
                                if( datos_in.nb_read(tmp) )         {
                                        f << "Vaciando cola de entrada... Dato leído: " << tmp << endl;
                                        buffer_lleno = false;
                                }
                                else
                                        f << "No se puede vaciar buffer" << endl;
                        }

                        if( datos_requeridos ) {
                                if( datos_out.nb_write(23) )        {
                                        f << "Llenando cola de salida... Dato Volcado: 23" << endl << endl;
                                        datos_requeridos = false;
                                }
                                }               wait();       }               }               }
```

8.3. Código para las pruebas finales

8.3.1. Árbitro

```
#include "systemc.h"
#include "constantes.h"

SC_MODULE(AHB_DummyArbiter)
{
	//Señales básicas
	sc_in_clk		HCLK;		// Reloj del sistema
	sc_in< bool >		HRESETn;	// Reset del sistema

	//Señales de Entrada
	sc_in< bool >		HBUSREQ;	//Petición de bus
	sc_in< bool >		HLOCK;		//Si 1 indica petición bloqueante

	//Señales de Salida
	sc_out< bool >		HGRANT;		//Concede el bus al AHB_DummyMaster
	sc_out< sc_bv<4> >	HMASTER;	//Indica qué master controla el bus

	//Procesos del módulo
	void Arbiter();

	SC_CTOR(AHB_DummyArbiter)
	{
		SC_CTHREAD(Arbiter,HCLK.pos());
			watching(HRESETn.delayed()==true);
	}
};

#include "AHB_DummyArbiter.h"

void AHB_DummyArbiter::Arbiter()
{
	wait_until(HRESETn.delayed() == false);

	while(true)
	{
		if( HBUSREQ.delayed() == true )
			HGRANT.write(true);
		wait();
	}
}
```

8.3.2. Decodificador

```
#include "systemc.h"
#include "constantes.h"

SC_MODULE(AHB_DummyDecoder)
{
	//Señales básicas
	sc_in_clk			HCLK;		// Reloj del sistema
	sc_in< bool >			HRESETn;	// Reset del sistema

	//Señales de Entrada
	sc_in< sc_bv<BUS_LENGTH> >HADDR;		//Bus de direcciones

	//Señales de Salida
	sc_out< bool >		HSEL_Sink;		//Esclavo del modulo sink
	sc_out< bool >		HSEL_Source;		//Esclavo del modulo source
	sc_out< bool >		HSEL_DefaultSlave;	//Activación del Esclavo por defecto

	//Procesos del módulo
	void Decoder();
	SC_CTOR(AHB_DummyDecoder)
	{
		SC_METHOD(Decoder);
		sensitive<<HADDR<<HCLK<<HRESETn;
```

```
	}
};

#include "AHB_DummyDecoder.h"

void AHB_DummyDecoder::Decoder()
{
	sc_int<32>		adress;

	adress = HADDR.read();

	if( adress >= SOURCE_ADRESS && adress <= 0x000030FF )
	{
		HSEL_Sink.write(false);
		HSEL_Source.write(true);
		HSEL_DefaultSlave.write(false);
	}
	else if( adress >= SINK_ADRESS && adress <= 0x000040FF )
	{
		HSEL_Sink.write(true);
		HSEL_Source.write(false);
		HSEL_DefaultSlave.write(false);
	}
	else if( adress == MASTER_ADRESS )
	{
		//next_trigger(10,SC_NS);
		HSEL_Sink.write(false);
		HSEL_Source.write(false);
		HSEL_DefaultSlave.write(true);
	}
	else
	{
		HSEL_Sink.write(false);
		HSEL_Source.write(false);
		HSEL_DefaultSlave.write(true);
	}
}
```

8.3.3. Esclavo por defecto

```
#include "systemc.h"
#include "constantes.h"

SC_MODULE(AHB_DefaultSlave)
{
	//Señales básicas
	sc_in_clk			HCLK;		// Reloj del sistema
	sc_in< bool >			HRESETn;	// Reset del sistema

	//Señales del decodificador
	sc_in< bool >			HSEL;		// Activacion del Slave
	//Señales del master
	sc_in< bool >			HWRITE;
	sc_in< sc_bv<BUS_LENGTH> > HADDR;	//Bus de direcciones
	sc_in< sc_bv<BUS_LENGTH> > HWDATA;	//Bus de datos
	sc_in< sc_bv<2> >		HTRANS;		//Tipo de transferencia
	sc_in< sc_bv<3> >		HBURST;		//Tipo de ráfaga
	sc_out< bool >			HREADY;		//Indica la disponivilidad del esclavo
	sc_out< sc_bv<BUS_LENGTH> > HRDATA;	//Bus de datos
	sc_out< sc_bv<2> >		HRESP;		//Indica estado del esclavo

	//Procesos del modulo
	void DefaultSlave();

	//Constructor
	SC_CTOR(AHB_DefaultSlave)
	{
		SC_CTHREAD( DefaultSlave, HCLK.pos() );
			watching( HRESETn.delayed() == true );
	}
};
```

```
#include "AHB_DefaultSlave.h"

void AHB_DefaultSlave::DefaultSlave()
{
	HRESP.write(OKAY);
	HREADY.write(true);

	//wait_until(HRESETn.delayed() == true);
	wait();

	while(true)
	{
		if(HSEL.delayed() == true && HTRANS.read() != IDLE && HTRANS.read() != BUSY)
		//Si detecta que el master quiere transmitir
		{
			//Programa respuesta ERROR
			HREADY.write(false);
			HRESP.write(eRROR);
			wait();
			HREADY.write(true);
			wait();
			HRESP.write(OKAY);
		}
		wait();
	}
}
```

8.3.4. Fuente

```
SC_MODULE(Fuente)
{
	//Variables locales
	int estado;			//Indica el estado actual de la máquina de estados

	//Variables para la sincronización entre procesos del módulo
	int cod_operacion;
	bool buffer_lleno;		//Indica que la cola datos_in esta llena
	bool no_datos;			//Programa al esclavo para que aborte todas las operaciones de lectura
	bool fin_operacion;		//Notifica el fin de una operación
	bool inicio_operacion;		//Notifica el comienzo de una operación
	bool datos_requeridos;		//Petición de datos para una operación de lectura
	sc_fifo< sc_bv<BUS_LENGTH> > datos_in;		//Cola de datos recibidos a través del bus
	sc_fifo< sc_bv<BUS_LENGTH> > datos_out;		//Cola de datos a enviar a través del bus

	/**** Entrada ****/

	//Señales básicas
	sc_in_clk			HCLK;		// Reloj del sistema
	sc_in< bool >			HRESETn;	// Reset del sistema

	//Señales del decodificador
	sc_in< bool >			HSEL;		// Activacion del Esclavo

	//Señales del master
	sc_in< bool >			HWRITE;		//Si 1 --> escritura, si 0 --> lectura
	sc_in< sc_bv<BUS_LENGTH> > HADDR;		//Bus de direcciones
	sc_in< sc_bv<BUS_LENGTH> > HWDATA;		//Bus de datos de master a esclavo
	sc_in< sc_bv<2> >		HTRANS;		//Tipo de transferencia
	sc_in< sc_bv<3> >		HBURST;		//Tipo de ráfaga
	/**** Fin Entrada ****/

	/**** Salida ****/
	sc_out< bool >			HREADY;		//Indica la disponivilidad del esclavo para opera
	sc_out< sc_bv<BUS_LENGTH> > HRDATA;		//Bus de datos de esclavo a master
	sc_out< sc_bv<2> >		HRESP;		//Indica estado del esclavo

	/**** Fin Salida ****/
	SC_CTOR(Fuente)
	{
		//Proceso de comunicación por AMBA AHB
		SC_CTHREAD( EsclavoAMBA_AHB, HCLK.pos() );
			watching( HRESETn.delayed() == true );

		//Proceso que requerirá el AMBA como método de comunicación:
```

```
		SC_CTHREAD( source, HCLK.pos() );
			watching( HRESETn.delayed() == true );
	}
	void EsclavoAMBA_AHB();
	void source();
};


void Fuente::source()
{
	int i, j, k, p;
	sc_uint<8> matriz[4][8][8];

	//Inicializa matrices
	k=0;
	for(p=0; p<4; p++)
		for(i=0; i<8; i++)
			for(j=0; j<8; j++)
				matriz[p][i][j]=k++;
	no_datos=false;
	wait_until(HRESETn.delayed() == false);

	i=0; j=0; p=0;

	//Envía matrices segun se le van pidiendo
	while(true)
	{
		if(inicio_operacion || j<8)
		{
			if(j<8)
			{
				datos_out.write(matriz[p][i][j]);
				j++;
			}
		}

		if(fin_operacion)
		{
			switch(cod_operacion)
			{
				case 0:
					cout << "Source --> Codigo de operación 0: sin valor" << endl;
				break;

				case 1:
					cout << "Source --> Se ha completado la recepcion de datos" << endl;
				break;

				case 2:
					cout << "Source --> Datos enviados correctamente " << i << endl;
				break;

				case 3:
				cout << "Source --> Operacion abortada" << endl;
			}
			fin_operacion=false;
		}

		if(j == 8)
		{
			j=0;
			i++;
			if(i==8)
			{
				i=0;
				p++;
				if(p==4)  break;
			}
		}
		wait();
	}
	while(true) wait();
}
```

8.3.5. Matriz

```
#include "systemc.h"
#include "constantes.h"

SC_MODULE(MPEG)
{
	//Variables locales
	int estado;		//Indica el estado actual de la máquina de estados
	int num_datos_incr;	//Número de datos a enviar en una ráfaga INCR

	//Variables para la sincronización entre procesos del módulo
	int cod_operacion;		//Indica la existencia de errores en la operación
	bool empieza;			//Indica el inicio/fin de una operación
	bool escritura;			//Tipo de operación: 1 escritura, 0 lectura
	bool buffer_lleno;		//Indica que la cola datos_in se ha llenado
	bool datos_requeridos;		//Pide datos a los procesos del programador
	sc_bv< 3 > tipo;		//Indica tipo de ráfaga --> señal HBURST
	sc_bv< 2 > ancho;		//Indica tamaño de datos --> señal HSIZE
	sc_bv< 32 > direccion_base;	//Proporciona la dirección base del esclavo con el que operar
	sc_fifo< sc_bv< BUS_LENGTH > > datos_in;	//Cola de datos recibidos a través del bus
	sc_fifo< sc_bv< BUS_LENGTH > > datos_out;	//Cola de datos a enviar a través del bus

	/**** Entrada ****/
	//Señales básicas
	sc_in_clk		HCLK;		//Reloj del sistema
	sc_in< bool >		HRESETn;	//Reset del sistema

	//Señales del árbitro
	sc_in< bool >		HGRANT;		//Si 1 bus concedido, si 0 no
	sc_in< sc_bv<4> >	HMASTER;	//Indica qué master controla el bus

	//Señales del esclavo
	sc_in< bool >		HREADY;		//Indica la disponivilidad del esclavo para operar
	sc_in< sc_bv<BUS_LENGTH> >HRDATA;	//Bus de datos de esclavo a master
	sc_in< sc_bv<2> >	HRESP;		//Indica estado del esclavo
	/**** Fin Entrada ****/

	/**** Salida ****/
	//Señales hacia el árbitro
	sc_out< bool >		HBUSREQ;	//Petición de bus
	sc_out< bool >		HLOCK;		//Si 1 indica petición bloqueante

	//Señales hacia el esclavo
	sc_out< bool >		HWRITE;		//Si 1 --> escritura, si 0 --> lectura
	sc_out< sc_bv<BUS_LENGTH> > HADDR;	//Bus de direcciones
	sc_out< sc_bv<2> >	HTRANS;		//Tipo de transferencia
	sc_out< sc_bv<2> >	HSIZE;		//Tamaño de los datos que circulan por el bus
	sc_out< sc_bv<3> >	HBURST;		//Tipo de ráfaga
	sc_out< sc_bv<3> >	HPROT;		//Seguridad en los datos (No se utiliza)
	sc_out< sc_bv<BUS_LENGTH> > HWDATA;	//Bus de datos de master a esclavo
	/**** Fin Salida ****/

	SC_CTOR(MPEG)
	{
		//Proceso de comunicación por AMBA AHB
		SC_CTHREAD( MasterAMBA_AHB, HCLK.pos() );
			watching( HRESETn.delayed() == true );

		//Proceso que requerirá el AMBA como método de comunicación:
		SC_CTHREAD( Matrix, HCLK.pos() );
			watching( HRESETn.delayed() == true );
	}

	void MasterAMBA_AHB();
	void Matrix ();
};


void Matriz::Matrix ()
{
	int i, j, p;
	sc_uint<8> matriz[4][8][8];
```

```
wait_until(HRESETn.delayed()==false);

for(p=0; p<4; p++)
{
	for(i=0; i<8; i++)
	{
		j=0;
		tipo=INCR8;
		ancho=BYTE;
		direccion_base=SOURCE_ADRESS;
		escritura=false;
		wait();
		empieza=true;
		while(empieza)
		{
			if(datos_in.num_available() > 0 && j<8)
			{
				matriz[p][j][i]=datos_in.read();
				j++;
			}
			wait();
		}

		switch(cod_operacion)
		{
			case 0:
				while(datos_in.num_available() > 0 && j<8)
				{
					matriz[p][j][i]=datos_in.read();
					j++;
					wait();
				}
				cout<<"MPEG --> Operacion Finalizada "<< i << endl;
			break;

			case 1:
				cout<< "MPEG --> Tipo de ráfaga erronea" << endl;
			break;

			case 2:
				cout<< "MPEG --> Ancho de palabra erroneo" << endl;
			break;

			case 3:
				cout<< "MPEG --> Operacion ABORTADA" << endl;
		}
		//wait();
	}
}

cout << endl << endl <<"MPEG --> RECEPCION DE DATOS COMPLETADA" << endl << endl;

for(p=0; p<4; p++)
{
	for(i=0; i<8; i++)
	{
		j=0;
		tipo=INCR8;
		ancho=BYTE;
		direccion_base=SINK_ADRESS;
		escritura=true;
		wait();
		empieza=true;
		while(empieza)
		{
			if(datos_out.num_free() > 0 && j<8)
			{
				datos_out.write(matriz[p][i][j]);
				j++;
			}
			wait();
		}

		switch(cod_operacion)
		{
			case 0:
```

```
                                        cout<< "MPEG --> Operacion Finalizada "<< i << endl;
                                        wait();
                                break;

                                case 1:
                                        cout<< "MPEG --> Tipo de ráfaga erronea" << endl;
                                break;

                                case 2:
                                        cout<< "MPEG --> Ancho de palabra erroneo" << endl;
                                break;

                                case 3:
                                        cout<< "MPEG --> Operacion ABORTADA" << endl;
                        }
                        wait();
                }
        }

        while(true)
                wait();
}
```

8.3.6. Destino

```
SC_MODULE(Destino)
{
        //Variables locales
        int estado;             //Indica el estado actual de la máquina de estados

        //Variables para la sincronización entre procesos del módulo
        int cod_operacion;
        bool buffer_lleno;      //Indica que la cola datos_in esta llena
        bool no_datos;          //Programa al esclavo para que aborte todas las operaciones de lectura
        bool fin_operacion;     //Notifica el fin de una operación
        bool inicio_operacion;          //Notifica el comienzo de una operación
        bool datos_requeridos;          //Petición de datos para una operación de lectura
        sc_fifo< sc_bv<BUS_LENGTH> > datos_in;          //Cola de datos recibidos a través del bus
        sc_fifo< sc_bv<BUS_LENGTH> > datos_out;         //Cola de datos a enviar a través del bus

        /**** Entrada ****/
        //Señales básicas
        sc_in_clk               HCLK;           // Reloj del sistema
        sc_in< bool >           HRESETn;        // Reset del sistema

        //Señales del decodificador
        sc_in< bool >           HSEL;           // Activacion del Esclavo

        //Señales del master
        sc_in< bool >           HWRITE;                 //Si 1 --> escritura, si 0 --> lectura
        sc_in< sc_bv<BUS_LENGTH> > HADDR;               //Bus de direcciones
        sc_in< sc_bv<BUS_LENGTH> > HWDATA;              //Bus de datos de master a esclavo
        sc_in< sc_bv<2> >       HTRANS;         //Tipo de transferencia
        sc_in< sc_bv<3> >       HBURST;         //Tipo de ráfaga
        /**** Fin Entrada ****/

        /**** Salida ****/
        sc_out< bool >          HREADY;         //Indica la disponivilidad del esclavo para opera
        sc_out< sc_bv<BUS_LENGTH> > HRDATA;     //Bus de datos de esclavo a master
        sc_out< sc_bv<2> >      HRESP;          //Indica estado del esclavo
        /**** Fin Salida ****/

        SC_CTOR(Destino)  {
                //Proceso de comunicación por AMBA AHB
                SC_CTHREAD( EsclavoAMBA_AHB, HCLK.pos() );
                        watching( HRESETn.delayed() == true );

                //Proceso que requerirá el AMBA como método de comunicación:
                SC_CTHREAD( sink, HCLK.pos() );
                        watching( HRESETn.delayed() == true );
        }
        void EsclavoAMBA_AHB();
        void sink();
};
```

```
void Destino::sink()
{
	int i, j, p;
	sc_uint<8> matriz[4][8][8];

	no_datos=false;
	wait_until(HRESETn.delayed() == false);

	i=0; j=0; p=0;

	while(true)
	{
		if(inicio_operacion)
		{
			if(j<8 && datos_in.num_available() > 0)
			{
				matriz[p][i][j] = datos_in.read();
				j++;
			}
		}

		if(j == 8)
		{
			j=0;
			i++;
			if(i==8)
			{
				i=0;
				p++;

			}
		}

		if(fin_operacion)
		{
			switch(cod_operacion)
			{
				case 0:
					cout << "Sink --> Codigo de operacion 0: sin valor" << endl;
				break;

				case 1:
					while(j<8 && datos_in.num_available() > 0)
					{
						matriz[p][i][j] = datos_in.read();
						j++;
						wait();
					}

					if(j == 8)
					{
						j=0;
						i++;
						if(i==8)
						{
							i=0;
							p++;

						}
					}

					cout << "Sink --> Se ha completado la recepcion de datos" << i << endl;
				break;

				case 2:
					cout << "Sink --> Datos enviados correctamente " << i << endl;
				break;

				case 3:
					cout << "Sink --> Operación abortada" << endl;
			}
			fin_operacion=false;
			if(p==4)
			{
				wait();
				break;
```

```
                        }
                }
                wait();
        }

        cout <<"SINK --> Los datos recibidos son:" << endl << endl;
        ofstream f("matriz.txt",ios::out);
        for(p=0; p<4; p++)
        {
                for(i=0; i<8; i++)
                {
                        for(j=0; j<8; j++)
                                f << matriz[p][i][j] << " ";
                        f << endl;
                }
                f << endl << endl;
        }

        f.close();
        cout << "Fin de la operación" << endl;
        sc_stop();
}
```

Memoria elaborada por ……..…….
que la firma para dar fe de ello.