UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Desarrollo de un CORE interfaz entre Picoblaze y AXI4-Lite

GRADO EN INGENIERÍA DE
TECNOLOGÍAS DE TELECOMUNICACIÓN

CARLOS PÉREZ MUÑOZ
MÁLAGA, 2018

E.T.S. DE INGENIERÍA DE TELECOMUNICACIÓN, UNIVERSIDAD DE MÁLAGA

Desarrollo de un CORE interfaz entre Picoblaze y AXI4-Lite

Autor: Carlos Pérez Muñoz
Tutor: Martín González García
Departamento: Tecnología Electrónica
Titulación: Grado en Ingeniería de Tecnologías de Telecomunicación
Palabras clave: FPGA, VHDL, SoC, Picoblaze, AXI4-Lite, FSMD, ASMD

Resumen

Este trabajo fin de grado consiste en el desarrollo y testeo de un módulo VHDL que implementa una interfaz entre el microcontrolador Picoblaze y el bus AXI4-Lite para ser implementados como parte de un System on Chip (SoC) dentro de una FPGA de Xilinx.

Con este módulo, es posible hacer uso de periféricos que usan el estándar AXI4-Lite como interfaz para registros desde Picoblaze. Así, combinando un microcontrolador empotrado con los periféricos necesarios, se puede diseñar un sistema completo dentro de un solo chip para una funcionalidad concreta. Para este trabajo, se ha escogido un sistema capaz de generar patrones de imagen y transmitirlos a un monitor vía HDMI. La interfaz diseñada permitirá controlar dicho patrón desde un PC con ayuda de Picoblaze y un transceptor UART. También será posible consultar la información de los registros de los periféricos integrados en dicho sistema.

Se ha desarrollado una interfaz para un bus AXI4-Lite con un ancho de bus tanto de datos como de direcciones de 32 bits para lectura y escritura. Se ha puesto especial énfasis en la rapidez y facilidad de uso desde Picoblaze, consiguiendo que, por ejemplo, una operación de escritura solo necesite cinco instrucciones.

E.T.S. DE INGENIERÍA DE TELECOMUNICACIÓN, UNIVERSIDAD DE MÁLAGA

Development of an interface CORE between Picoblaze and AXI4-Lite

Author: Carlos Pérez Muñoz

Supervisor: Martín González García

Department: Electronic Technology

Degree: Grado en Ingeniería de Tecnologías de Telecomunicación

Keywords: FPGA, VHDL, SoC, Picoblaze, AXI4-Lite, FSMD, ASMD

Abstract

This thesis consists of the development and testing of a VHDL module which implements an interface between the Picoblaze microcontroller and the AXI4-Lite bus to be implemented as part of a SoC inside a Xilinx FPGA.

With this module, it is possible to make use of peripherals which use the AXI4-Lite standard as the interface to access its registers from Picoblaze. So, combining an embedded microcontroller with the necessary peripherals, a complete system inside a chip with a specific functionality can be created. For this thesis, a system able to generate video patterns and transmit them to a screen by HDMI has been chosen. The designed interface will allow the control of the pattern from a PC with help of Picoblaze and an UART transceiver. Also, it will be possible to consult the information inside the registers of those peripherals.

An interface for an AXI4-Lite bus with 32-bit wide data and address, read and write buses has been developed. The design is specially focused on speed and easiness of use from Picoblaze. For example, a write operation only requires five instructions to be executed.

Agradecimientos

A mi familia, especialmente a mis padres, a mis amigos y compañeros de carrera y a mi tutor de este TFG por todo el apoyo recibido.

A mis padres, por haberme ayudado

y apoyado todos estos años...

Contenido

Lista de Acrónimos

AP SoC	All Programmable System on Chip
ASMD	Algorithmic State Machine with Datapath
AXI	Advanced eXtensible Interface
CPU	Central Processing Unit
DCM	Digital Clock Management
E/S	Entrada/Salida
HDMI	High-Definition Multimedia Interface
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FSMD	Finite State Machine with Datapath
IP	Intellectual Property
LC	Logic Cell
LUT	Look-Up Table
SoC	System on Chip
UART	Universal Asynchronous Receiver-Transmitter
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

Lista de figuras

Lista de tablas

Capítulo 1. Introducción

1.1. Contexto y problemas a resolver

Desde hace varias décadas, se viene buscando la integración de la mayor cantidad de elementos de un sistema electrónico dentro de un único chip, es decir, acercarse al concepto de SoC. El uso de SoC no solo permite crear sistemas de menor tamaño, sino que permite que éste funcione a mayor velocidad y con un consumo menor gracias a la disminución de las capacidades parásitas respecto a un sistema montado sobre un PCB. Los SoC son prácticamente omnipresentes hoy en día gracias al aumento de su capacidad de integración, de velocidad, de eficiencia energética y, sobre todo, a la reducción de sus costes de fabricación. Sin ellos, tecnologías como los *smartphones*, las *tablets* y la domótica no serían posibles

Los SoC contienen tres elementos principales: una Unidad de Procesamiento Central (CPU), uno o varios elementos de memoria para almacenar datos e instrucciones, y periféricos que dotan a la CPU de nuevas capacidades o mejoran las ya existentes. Los periféricos pueden tener funciones muy variadas, desde sistemas de procesado digital de la señal (DSP) como multiplicadores hardware, hasta sistemas de comunicaciones como USB o Wi-Fi. Sin embargo, todos tienen algo en común, y es que deben estar comunicados de alguna forma con la CPU para que ésta pueda hacer uso de ellos y controlarlos. Sin embargo, hay dos grandes problemáticas en esta cuestión:

- Los recursos con los que cuenta la CPU para conectarse con el exterior son limitados, sin embargo, pueden existir decenas, e incluso cientos, de periféricos a conectar, cada uno con decenas o cientos de registros.
- La comunicación entre CPU y periféricos ha de ser tan rápida como sea posible. Una de las funciones de los periféricos es aliviar la carga de trabajo sobre la CPU en tareas concretas. Pero si la comunicación con

el periférico es lenta, el rendimiento global del sistema puede verse seriamente afectado.

Para dar solución a estos problemas, ARM, compañía dedicada al desarrollo de IP (propiedad intelectual), ideó el estándar de interconexión de periféricos AXI4, orientado a la transmisión a ráfagas. Sin embargo, esta interfaz es bastante compleja, por lo que ARM publicó una variante del estándar con menos capacidades, pero más fácil de implementar: AXI4-Lite.

AXI4-Lite permite la interconexión de uno o varios maestros con uno o varios esclavos y realizar una operación de lectura y otra de escritura simultáneamente. Los detalles de este estándar serán tratados en el capítulo correspondiente.

Por otro lado, otra de las tecnologías clave en este proyecto es la tecnología FPGA (Field Programmable Gate Array), que permite la implementación de cualquier circuito digital en su interior de forma fácil y todas las veces que sean necesarias. Para este proyecto, nos centraremos en las FPGA de Xilinx, puesto que es una compañía con mucha experiencia en este campo y, además, ofrece también una gran variedad de IP para ser utilizadas en nuestros diseños. Gran parte de esas IP son periféricos que usan el estándar AXI4-Lite como interfaz de control, lo que da cuenta de la importancia de este bus y de su utilidad.

Una de las IP de Xilinx en la que más nos centraremos será el microprocesador Picoblaze. Picoblaze es un sencillo microcontrolador de 8 bits que puede ser implementado en cualquier FPGA moderna de Xilinx consumiendo pocos recursos. Este procesador, por su ligereza, es perfecto para tomar el papel de CPU en sistemas simples. Por contrapartida, el sistema de entrada/salida (E/S) es muy simple y no se puede conectar directamente a periféricos que hagan uso de AXI4-Lite como interfaz de configuración. Es aquí donde entra en juego la interfaz que será objeto de desarrollo en esta memoria.

1.2. Objetivos del proyecto

EL objetivo principal es desarrollar un Soft CORE descrito en VHDL a nivel de transferencia entre registros (RTL) y con metodología RT que permita la

interconexión de un microcontrolador Picoblaze con un bus AXI4-Lite para ser usado como parte de un sistema completo. Se persigue un diseño rápido en términos de ciclos de reloj, fácil de usar (a nivel hardware y software) y completamente integrable junto con otros periféricos. Además, ha de ser sintetizable y portable a cualquier FPGA.

También se ha propuesto diseñar un sistema completo (esto es, un microcontrolador Picoblaze, su memoria de programa y varios periféricos para proporcionar una funcionalidad completa) que haga uso de esta interfaz y de un programa en ensamblador para Picoblaze capaz de aprovecharlo. Este sistema sirve tanto para testear el CORE como para comprender su uso a modo de ejemplo y aportar código VHDL y rutinas en ensamblador para su uso en futuros proyectos.

Por último, está la redacción de esta memoria que, junto con el sistema y programa arriba mencionados, permitirá su fácil reutilización.

1.3. Estructura de la memoria

El primer paso es especificar el esquema circuital y el funcionamiento lógico de los dos elementos que se quieren conectar: Picoblaze y el bus AXI4-Lite. Su comprensión es fundamental para el diseño del CORE, el cuál será tratado en profundidad inmediatamente después. También se hará una pequeña introducción a la tecnología FPGA, la serie 7 de Xilinx y la placa Zybo, que será usada para poner a prueba nuestro sistema. También se dará una pequeña explicación de la metodología de diseño RT.

El capítulo siguiente se usará para explicar el desarrollo del CORE, mostrando el diseño que se persigue y su implementación en VHDL. A continuación, se dedicará un capítulo a ofrecer consejos y guías para la integración y utilización de nuestro diseño dentro de un sistema completo, tanto a nivel hardware como software. Luego se procederá a su testeo valiéndose de una simulación RTL en ModelSim y su integración en un sistema generador de video controlado por UART. Por último, se expondrán las conclusiones sacadas de este proyecto,

describiendo los objetivos alcanzados y proponiendo posibles mejoras futuras para el CORE.

Capítulo 2. Consideraciones previas

En este capítulo se describirán el esquema circuital y el comportamiento lógico de los dos sistemas que se pretenden conectar: Picoblaze, poniendo énfasis en su sistema de E/S, y el bus AXI4-Lite.

2.1. Picoblaze

Picoblaze es un CORE propiedad de Xilinx para ser usado en sus FPGA. Se trata de un sencillo microcontrolador de 8 bits optimizado para consumir la menor cantidad de recursos posible. Debido a su simplicidad y al entorno de desarrollo que incorpora, solo se puede programar en ensamblador. Sus características son las siguientes [1]:

- 32 registros de 8 bits, agrupados en dos bancos de 16 registros cada uno.
- Memoria de programa de hasta 4Kb.
- Cada instrucción consume dos ciclos de reloj.
- Flags de resultado cero (Z) y acarreo (C).
- Hasta 256 entradas y 256 salidas

Este último punto es el que nos interesa para este proyecto. El sistema E/S de Picoblaze contiene las siguientes señales:

- *Port_id*: bus de 8 bits de salida para identificar el puerto donde se desea leer o escribir.
- *In_port*: bus de 8 bits de entrada para leer datos.
- *Out_port*: bus de 8 bits de salida que contiene el dato de salida
- *Read_strobe*: señal de strobe que se activa durante un ciclo para indicar cuándo se está ejecutando una lectura.
- *Write_strobe*: señal de strobe con duración de un ciclo de reloj para indicar que el dato es válido. Se debería usar como señal de habilitación del registro donde se está escribiendo.
- *K_write_strobe*: similar al anterior desde el punto de vista hardware, pero solo utilizada por la instrucción OUTPUTK y solo los 4 bits menos

significativos (LSB) de *port_id* son válidos. Los demás deberían ser ignorados.

A continuación, en las figuras 2.1 y 2.2, se muestran un ejemplo de escritura y otro de lectura desde Picoblaze:

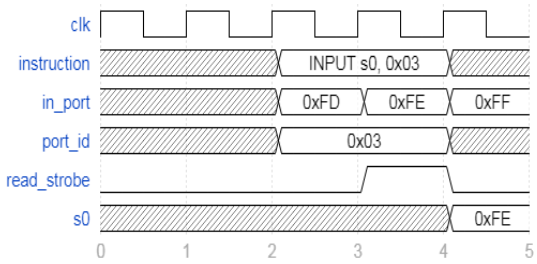


Figura 2.1. Lectura desde Picoblaze [Fuente: propia]

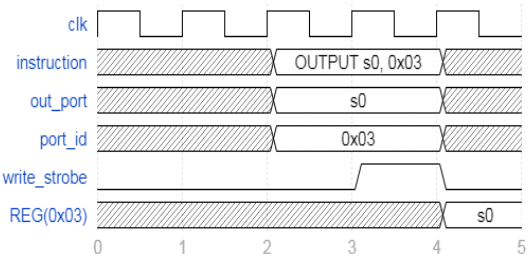


Figura 2.2. Escritura desde Picoblaze [Fuente: propia]

En la figura 2.1 se ve el proceso de lectura. El dato presente en *in_port* es capturado en el registro especificado en la instrucción en el flanco de subida de reloj siguiente a la activación de la señal *read_strobe*. Dicha señal puede ser usada, por ejemplo, como señal de *pop* si se lee desde una FIFO. La señal *port_id* debería usarse como señal de selección en un multiplexor si se quiere leer desde varias fuentes.

En cuanto al proceso de escritura, se aprecia en la figura 2.2 que el dato correcto aparece en *out_port* al mismo tiempo que se activa *write_strobe*. En el flanco de subida siguiente, se debería capturar ese dato en el registro apropiado. Para ello, se debería obtener la señal de habilitación de dicho registro a partir de *port_id* y *write_strobe*. En cuanto a la señal *k_write_strobe,* su funcionamiento es idéntico al de *write_strobe*, pero solo puede usar los 4 bits menos significativos de *port_id*, por lo que se deberían ignorar los otros 4.

2.2. AXI4-Lite

El estándar de interconexión AXI4-Lite fue desarrollado por ARM como una versión más sencilla del estándar AXI4. Sus características principales son las siguientes [2]:

- Funcionamiento totalmente síncrono
- *Reset* activo a nivel bajo
- Cinco canales agrupados en dos para lectura y tres para escritura:
    - Canal de direcciones de lectura
    - Canal de datos de lectura
    - Canal de direcciones de escritura
    - Canal de datos de escritura
    - Canal de respuesta de escritura
- Funcionamiento basado en *handshake*
- Distinción entre interfaces maestras y esclavas
- Lectura y escritura simultáneas a dos direcciones distintas

En esta sección, primero se explicará el sistema de *handshake* mencionado anteriormente, debido a que todos los canales hacen uso de él, para después listar todas las señales y describir las operaciones de lectura y escritura.

Para explicar el proceso de *handshake,* se define el concepto de emisor (quien muestra el dato a transmitir en el bus) y receptor (quien debe capturar dicho dato presente en el bus). En todos los canales hay un mínimo de tres señales: uno o varios buses de datos, gestionados por el emisor, una señal *xVALID*, también activada por el emisor, y una señal *xREADY,* controlada por el receptor. Las reglas del *handshake* son las siguientes:

- La señal *xVALID* deberá activarse obligatoriamente a nivel alto cuando haya un dato válido presente en el bus.
- La señal *xREADY* debería activarse a nivel alto tan pronto como el receptor esté listo para recibir. Puede ser antes o después de la activación de *xVALID.* La norma no define ninguna causa de activación obligatoria, lo deja a merced del desarrollador.
- El *handshake* se produce cuando ambas señales están activadas simultáneamente. En el flanco de subida de reloj siguiente, el dato del bus deberá ser capturado por el receptor y las señales *xREADY* y *xVALID* deberán ser desactivadas, es decir, puestas a 0.

- Ninguna de las dos señales podrá activarse de nuevo hasta el siguiente flanco de subida de reloj como mínimo.

En las figuras 2.3 y 2.4 se muestran este proceso de forma visual para facilitar su comprensión:

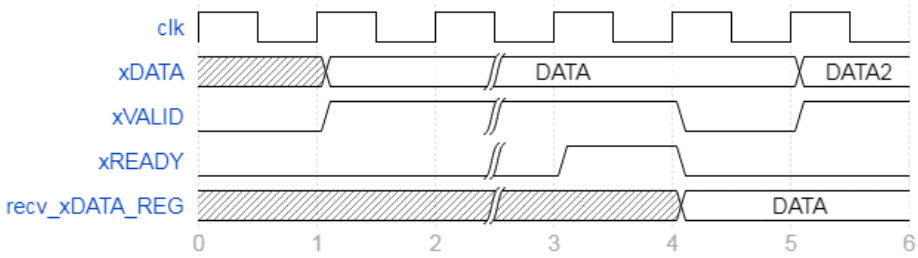


Figura 2.3. *Handshake* iniciado por emisor [Fuente: propia]

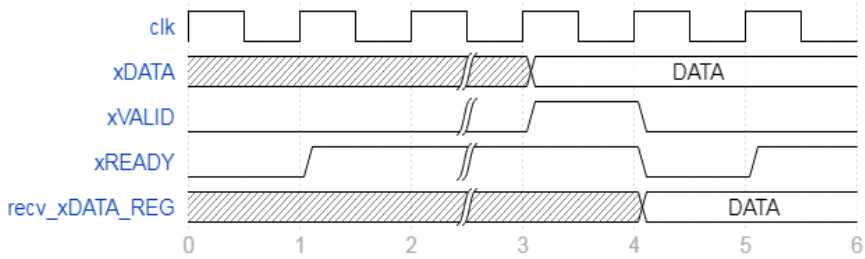


Figura 2.4. *Handshake* iniciado por receptor [Fuente: propia]

En la figura 2.3, el dato está presente en el bus antes de que el receptor esté listo, y en la figura 2.4, el receptor está preparado antes que el emisor.

A continuación, se enumerarán y explicarán todas las señales que componen este bus. Aquí hay que explicar los conceptos de maestro y esclavo. Un dispositivo maestro siempre inicia las peticiones de lectura y escritura. Uno esclavo siempre ha de esperar que un maestro le haga una solicitud y ha de atenderla. En la tabla 2.1 se explican todas las señales, agrupadas por canales. En cada canal se define quién es el emisor de acuerdo a la definición explicada anteriormente para el *handshake.*

Canal	Señal	Descripción de la señal
Canal de direcciones de escritura (emisor: maestro)	AWADDR	Bus de direcciones de escritura. Ancho variable. Para este proyecto, se ha escogido un ancho de 32 bits
	AWPROT	Bus de 3 bits destinado a proporcionar distintos niveles de acceso de escritura
	AWVALID	Señal *xVALID* de este bus
	AWREADY	Señal *xREADY* de este bus
Canal de datos de escritura (emisor: maestro)	WDATA	Bus de datos de escritura. En el estándar AXI4-Lite puede tener un ancho de 32 o 64 bits. Para este proyecto se ha escogido un ancho de 32 bits
	WSTRB	Un bit de strobe por cada byte del bus de datos. En AXI4-Lite siempre se usa el bus completo, así que todos los bits tienen siempre el mismo valor
	WVALID	Señal *xVALID* de este bus
	WREADY	Señal *xREADY* de este bus
Canal de respuesta de escritura (emisor: esclavo)	BRESP	Señal de 2 bits. En AXI4-Lite, cualquier valor distinto de “00” se considera un error en escritura
	BREADY	Señal *xREADY* de este bus
	BVALID	Señal *xVALID* de este bus
Canal de direcciones de lectura (emisor: maestro)	ARADDR	Bus de direcciones de lectura. Ancho variable. Para este proyecto, se ha escogido un ancho de 32 bits
	ARPROT	Bus de 3 bits destinado a proporcionar distintos niveles de acceso de lectura
	ARVALID	Señal *xVALID* de este bus
	ARREADY	Señal *xREADY* de este bus
Canal de datos de lectura (emisor: esclavo)	RDATA	Bus de datos de lectura. En el estándar AXI4-Lite puede tener un ancho de 32 o 64 bits. Para este proyecto se ha escogido un ancho de 32 bits
	WSTRB	Un bit de strobe por cada byte del bus de datos. En AXI4-Lite siempre se usa el bus completo, así que todos los bits tienen siempre el mismo valor
	RRESP	Señal de 2 bits. En AXI4-Lite, cualquier valor distinto de “00” se considera un error de lectura
	RVALID	Señal *xVALID* de este bus
	RREADY	Señal *xREADY* de este bus

Tabla 2.1. Listado de señales del bus AXI4-Lite

Aparte de las señales descritas en la tabla 2.1, hay otras dos señales comunes al maestro y al esclavo y generadas por un elemento externo: aCLK, el reloj del bus, y RSTn, la señal de *reset* activa a nivel bajo. De acuerdo a la norma, todas las señales han de ser síncronas con aCLK y, cuando se activa RSTn, todas las señales *xREADY* y *xVALID* han de configurarse a nivel bajo y no puede volver al nivel alto hasta un ciclo de reloj después de desactivarse RSTn.

A continuación, se describirán los procesos de escritura y lectura dentro de este bus. Para realizar una operación de escritura, el maestro ha de presentar en el canal de direcciones de escritura la dirección y el nivel de acceso, y el dato a escribir en el canal de datos de escritura. Una vez hecho, hay que esperar los *handshakes.* Aunque no viene explícitamente en el estándar, es lógico pensar que la dirección y el nivel de acceso serán capturados antes que el dato, ya que es necesario que el periférico sepa que el dato va dirigido a él antes de capturar el dato. Una vez enviados los datos de dichos canales, el maestro esperará a la respuesta en el canal de respuesta de escritura. Una vez hecho el *handshake* en ese canal, de acuerdo a la especificación de AXI4-Lite, cualquier dato distinto de “00” es considerado un error en la escritura.

El proceso de lectura es similar, pero prescindiendo del dato a escribir. El maestro presenta la dirección y el nivel de acceso y espera al *handshake* por parte del esclavo. Una vez hecho, el esclavo ha de entregar por el canal de datos de lectura tanto el dato, si la lectura ha ido bien, como la respuesta, que informa de si ha habido algún error. Una vez hecho el *handshake* de dicho canal, si la respuesta es distinta de “00”, se considera que ha habido un error y que el dato no es válido. En caso contrario, el maestro captura el dato.

2.3. Tecnología FPGA, serie 7 y placa Zybo

La tecnología FPGA permite la implementación de cualquier circuito digital en un chip de forma rápida y sencilla. Además, dependiendo de la tecnología, se puede volver a implementar un número prácticamente infinito de veces. Esto constituye una gran ventaja a la hora de desarrollar circuitos integrados y CORES, ya que no es necesario pedir a una compañía de semiconductores que nos fabriquen un prototipo, con el coste económico y temporal que supone.

Nos centraremos en las FPGA de la serie 7 de Xilinx, puesto que nuestro diseño y sistemas serán sintetizados e implementados en la placa Zybo de Digilent, que hace uso de un All Programmable System on Chip (AP SoC) que incluye una FPGA Artix-7. La base de esta tecnología es un elemento que se

llama Logic Cell (LC) [3]. En las FPGA de la serie 7, dicha LC está compuesta por:

- Una Look Up Table (LUT) de 6 entradas, que permite definir cualquier función lógica de 6 entradas y una salida o una función de 5 entradas y dos salidas. Algunas, además, pueden funcionar como memoria RAM distribuida y registro de desplazamiento serie.
- Un Flip-Flop (FF) que puede conectarse a la salida de la LUT para registrar dicha salida de acuerdo a una señal de reloj. Además, dispone de terminal de habilitación y *reset* configurable en modo síncrono o asíncrono.
- Lógica de multiplexación y acarreo que permite la fácil combinación de varias LUT y FF para dar lugar a circuitos más complejos.

Junto con las LC, también hay elementos dedicados a una función concreta, como los gestores digitales de reloj (Digital Clock Management o DCM), multiplicadores hardware o bloques de E/S (IOB). Nos interesa para este proyecto principalmente el DCM, ya que perseguimos que el diseño sea totalmente síncrono. El DCM permite tratar la señal de reloj que se suministra a la FPGA por uno de sus pines. Entre sus funcionalidades están la corrección del ciclo de trabajo a la entrada, la síntesis digital de frecuencia (DFS) y la mitigación del *skew* dentro de la FPGA. En el sistema que desarrollaremos después necesitaremos, por ejemplo, dos señales de reloj para poder transmitir por HDMI.

Queda por hablar de la placa Zybo, que será utilizada posteriormente. Esta placa incluye un AP SoC Zynq-7000 de Xilinx, que consiste en un SoC ARM ya fabricado, lo que Xilinx denomina Sistema Programable (PS), y una FPGA Artix-7 que se puede conectar al PS para extender sus funcionalidades con periféricos a medida [4]. Esta FPGA es denominada Lógica Programable (PL). Fuera de la FPGA se encuentra conectada a ella un oscilador que proporciona una señal de reloj de 125MHz. En cuanto a los periféricos de interés, nos encontramos 4 botones, 4 switches, 4 LED, varios puertos PMOD y un conector HDMI capaz de configurarse como emisor o receptor. En la figura 2.5 se muestra el aspecto físico de esta placa:

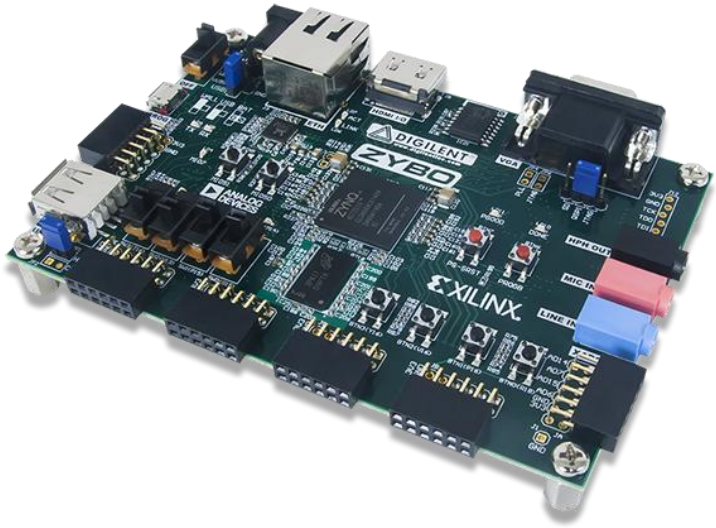


Figura 2.5. Placa *Zybo* [Fuente: página web de Digilent]

2.4. Metodología RT

La metodología RT permite diseñar a nivel RTL máquinas de estados finitos con *datapath* (FSMD) que pueden implementar cualquier algoritmo secuencial, de un modo similar al que ejecutan los microprocesadores.

La base se encuentra en los diagramas ASMD (Algorithmic State Machine with Datapath) [5]. Estos diagramas se asemejan a una combinación de un diagrama de flujo y un diagrama de estados. La unidad principal del diagrama ASMD es el bloque ASMD, que se identifica con un estado. Dentro de ese bloque nos podemos encontrar:

- Caja de estado (obligatorio): muestra el nombre del estado y las salidas Moore de ese estado, si las hay.
- Cajas de decisión: dentro se introduce un operador booleano que determina el flujo a seguir si el resultado es verdadero o falso.
- Caja de salida condicional: si el flujo seguido pasa por ella, las salidas listadas serán configuradas a los valores especificados.

Además, existen varias reglas a seguir:

- Las acciones dentro de un bloque ASMD se han de realizar en un único ciclo de reloj.
- Los caminos solo pueden dividirse en cajas de decisión.
- Las salidas de un bloque siempre irán a la caja de estado del mismo bloque u otro.

También hay que mencionar que, en este proyecto, se usará la estrategia de salida anticipada, esto es, si el siguiente estado conlleva alguna asignación nada más cambiar a él, dicha asignación se definirá en el anterior estado para que se produzca al mismo tiempo que la transición entre estados.

Basándose en un diagrama ASMD y siguiendo estas reglas, la traducción a VHDL es inmediata. Por ejemplo, si tenemos el siguiente diagrama, el de la figura 2.6:

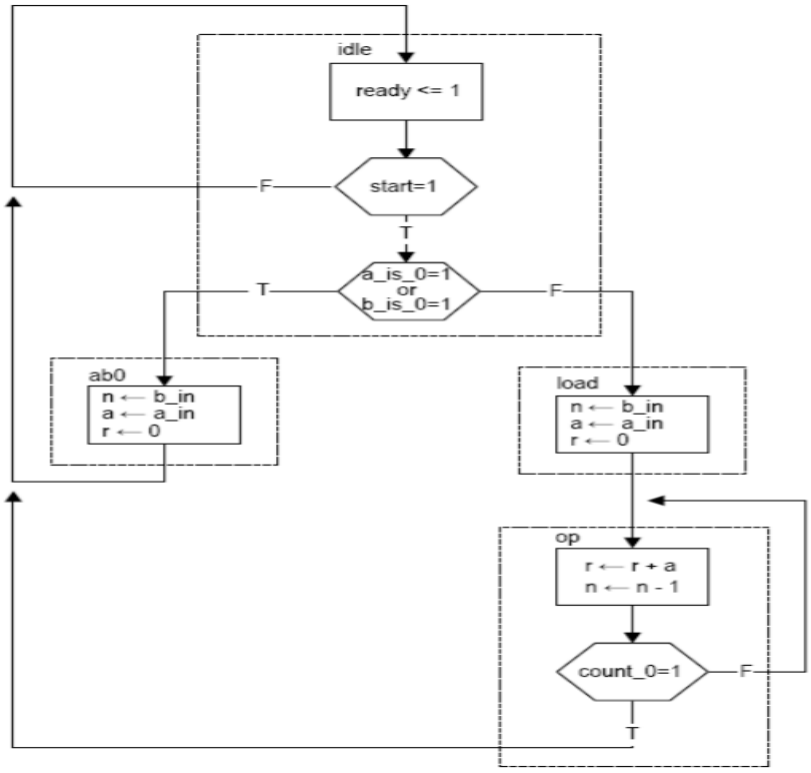


Figura 2.6. Ejemplo de diagrama ASMD [Fuente [5]]

El código VHDL correspondiente sería así (figura 2.7):

```
14    ready <= '1' when estado = idle else '0';
15    process(clk) begin
16     if rising_edge(clk) then
17      case estado is
18       when idle =>
19        if start = '1' then
20          if a_is_0 = '1' or b_is_0 = '1' then
21           estado <= ab0;
22           n <= b_in;
23           a <= a_in;
24           r <= 0;
25          else
26           estado <= load;
27           n <= b_in;
28           a <= a_in;
29           r <= 0;
30          end if;
31         end if;
32        when abo =>
33         estado <= idle;
34        when load =>
35         estado <= op;
36         r <= r + a;
37         n <= n + 1;
38        when op =>
39         if count_0 = '1' then
40          estado <= idle;
41         else
42          r <= r + a;
43          n <= n + 1;
44         end if;
45      end case;
46     end if;
47    end process;
```

Figura 2.7. Código VHDL del diagrama ASMD de ejemplo [Fuente: propia]

Capítulo 3. Desarrollo del módulo interfaz

En este capítulo se procederá a describir la especificación y el funcionamiento del CORE objetivo de este proyecto. El CORE en cuestión se ha desarrollado como una descripción a nivel RTL en VHDL en el fichero *PB_AXIL_32.vhd.*

3.1. Entidad del módulo

Empezamos hablando de las conexiones del módulo, lo que en VHDL se conoce como entidad. Para empezar, el CORE dispone de un genérico *port_pos,* que se trata de un vector de 8 bits en el que se puede indicar la dirección que Picoblaze usará para acceder al bus AXI4-Lite. Por defecto, es el 0x00. El apartado de señales se ha dividido en dos: la parte de Picoblaze y la parte de AXI4-Lite. En la parte de Picoblaze, se han declarado señales del mismo tipo, con el mismo nombre y modo opuesto a las señales del sistema de E/S de Picoblaze descritas en el apartado 2.1, con la idea de conectarlas directamente. Además, se ha añadido una señal de salida de un bit, *capturing,* cuya utilidad se explicará en el siguiente capítulo. Por último, la parte de AXI4-Lite tiene todas las señales listadas en la tabla 2.1, cada una con un modo tal que el interfaz actúe como maestro. En la figura 3.1 se muestra la entidad descrita en VHDL:

```
entity PB_AXIL_32 is
  generic(
    port_pos      : std_logic_vector:=x"00"                          --Puerto de referencia de la interfaz
);
  port(
    --Entradas comunes-------------------------------------------------------------------------------------------------------
    CLK           : in std_logic;                                    --Entrada de reloj
    RSTn          : in std_logic;                                    --Reset activo a nivel bajo
    --Interfaz PicoBlaze------------------------------------------------------------------------------------------------------
    in_port       : out std_logic_vector(7 downto 0):=x"00";         --Bus de datos de entrada de PicoBlaze
    out_port      : in std_logic_vector(7 downto 0);                 --Bus de datos de salida de Picoblaze
    port_id       : in std_logic_vector(7 downto 0);                 --Bus de direcciones de E/S de Picoblaze
    read_strobe   : in std_logic;                                    --Habilitación de lectura de puerto
    write_strobe  : in std_logic;                                    --Habilitación de escritura de variable en puerto
    k_write_strobe: in std_logic;                                    --Habilitación de escritura de constante en puerto
    capturing     : out std_logic:='0';                              --No se debería acceder a otros periféricos mientras esté activo.

    --Interfaz AXI Lite-------------------------------------------------------------------------------------------------------
    ----Bus de direcciones de escritura-----------------------------------------------------------------------------------------
    AWVALID       : out std_logic:='0';                              --Dato de bus válido
    AWREADY       : in std_logic;                                    --Esclavo listo para recibir
    AWADDR        : out std_logic_vector(31 downto 0):=(others => '0'); --Bus de direcciones para escritura
    AWPROT        : out std_logic_vector(2 downto 0):="000";         --Nivel de acceso en la escritura
    ----Bus de datos de escritura----------------------------------------------------------------------------------------------
    WVALID        : out std_logic:='0';
    WREADY        : in std_logic;
    WDATA         : out std_logic_vector(31 downto 0):=(others =>'0');  --Bus de datos de escritura
    WSTRB         : out std_logic_vector(3 downto 0);                --Bytes válidos del bus de datos
    ----Bus de respuesta de escritura------------------------------------------------------------------------------------------
    BVALID        : in std_logic;
    BREADY        : out std_logic:='0';
    BRESP         : in std_logic_vector(1 downto 0);                 --Respuesta del esclavo con el estado final de la escritura
    ----Bus de direcciones de lectura------------------------------------------------------------------------------------------
    ARVALID       : out std_logic:='0';
    ARREADY       : in std_logic:='0';
    ARADDR        : out std_logic_vector(31 downto 0):=(others => '0'); --Bus de direcciones para lectura
    ARPROT        : out std_logic_vector(2 downto 0):="000";         --Nivel de acceso en la lectura
    ----Bus de datos de lectura------------------------------------------------------------------------------------------------
    RVALID        : in std_logic;
    RREADY        : out std_logic:='0';
    RDATA         : in std_logic_vector(31 downto 0):=(others =>'0');
    RRESP         : in std_logic_vector(1 downto 0)
  );
end PB_AXIL_32;
```

Figura 3.1. Entidad del CORE

3.2. Parte declarativa y asignaciones concurrentes

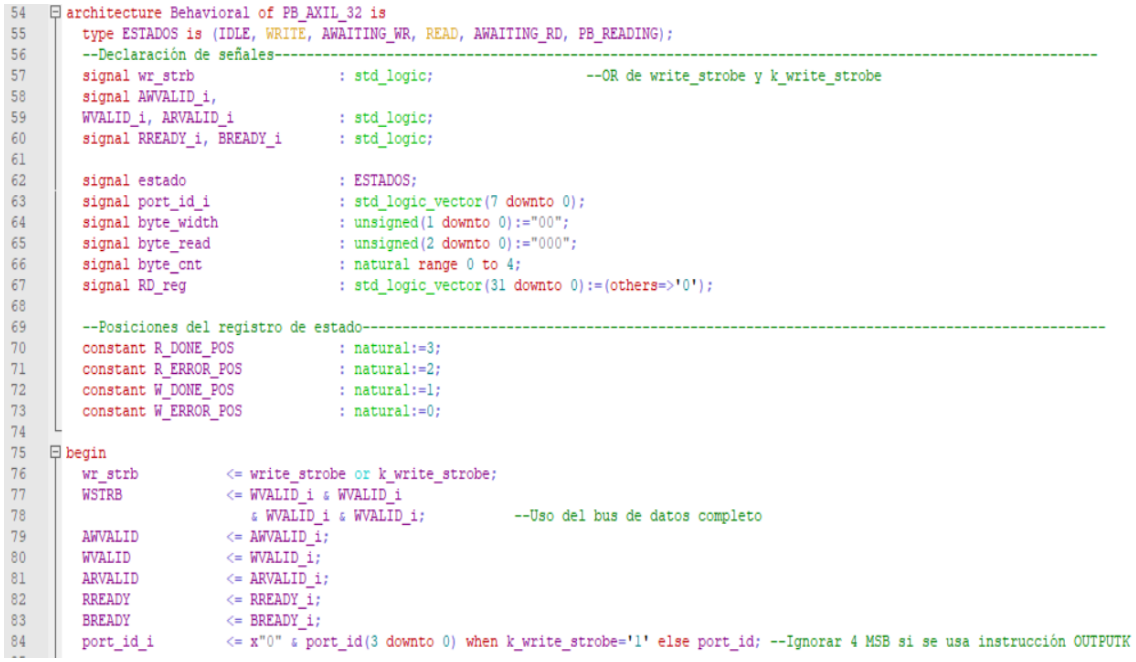
A continuación se describirá la parte declarativa de la arquitectura, esto es, las señales internas del módulo. Lo primero es definir un tipo *ESTADOS* ya que, como se verá después, este CORE consiste en una FSMD. El tipo *ESTADOS* contiene los posibles estados *IDLE, WRITE, AWAITING_WR, READING, AWAITING_RD* y *PB_READING.* Junto con este tipo *ESTADOS* se declara una señal *estado* de dicho tipo que modelará el registro de estado. El resto de señales son las siguientes:

- *wr_strb:* señal de un bit consistente en la OR de write_strobe y k_write_strobe.

- *AWVALID_i, WVALID_i, BREADY_i, ARVALID_i* y *BREADY_i:* señales internas para poder leer las salidas a las que se refieren. Dichas salidas tendrán por valor el de estas señales.
- *port_id_i:* señal interna que configura a 0 los 4 bits más significativos de *port_id* cuando *k_write_strobe* está a 1, es decir, cuando se usa la instrucción OUTPUTK de Picoblaze. Permite, junto con wr_strb, usar OUTPUT y OUTPUTK por igual.
- *byte_width:* registro tipo *unsigned* de 2 bits que almacena el número de bytes de datos y direcciones a leer desde Picoblaze.
- *byte_read:* similar a *byte_width,* pero de longitud 3 para indicar cuántas veces se espera que lea Picoblaze.
- *byte_cnt:* señal de tipo natural desde 0 a 4 para ser usada como contador junto con *byte_width* y *byte_read.*
- *RD_reg:* registro de 32 bits para almacenar el dato de lectura de AXI4-Lite tras el *handshake*,

Junto con estas señales, también hay declaradas unas constantes de tipo natural que sirven para indicar posiciones del registro de estado del CORE. Actualmente se usan cuatro de las ocho posiciones posibles. Las constantes declaradas son, desde el valor 3 al 0: *R_DONE_POS, R_ERROR_POS, W_DONE_POS* y *W_ERROR_POS*.

Ahora se explicará con detalle la implementación del módulo VHDL. Esta parte del fichero empieza con las asignaciones concurrentes que se acaban de describir en el apartado de señales internas, además de asignar a la señal *WSTRB* el valor de la señal *WVALID*. Todo lo descrito en este apartado se muestra en la figura 3.2:



```
54  architecture Behavioral of PB_AXIL_32 is
55      type ESTADOS is (IDLE, WRITE, AWAITING_WR, READ, AWAITING_RD, PB_READING);
56      --Declaración de señales-----------------------------------------------------------------------------------------
57      signal wr_strb                  : std_logic;                --OR de write_strobe y k_write_strobe
58      signal AWVALID_i,
59      WVALID_i, ARVALID_i             : std_logic;
60      signal RREADY_i, BREADY_i       : std_logic;
61
62      signal estado                   : ESTADOS;
63      signal port_id_i                : std_logic_vector(7 downto 0);
64      signal byte_width               : unsigned(1 downto 0):="00";
65      signal byte_read                : unsigned(2 downto 0):="000";
66      signal byte_cnt                 : natural range 0 to 4;
67      signal RD_reg                   : std_logic_vector(31 downto 0):=(others=>'0');
68
69      --Posiciones del registro de estado-------------------------------------------------------------------------------
70      constant R_DONE_POS             : natural:=3;
71      constant R_ERROR_POS            : natural:=2;
72      constant W_DONE_POS             : natural:=1;
73      constant W_ERROR_POS            : natural:=0;
74
75  begin
76      wr_strb          <= write_strobe or k_write_strobe;
77      WSTRB            <= WVALID_i & WVALID_i
78                          & WVALID_i & WVALID_i;          --Uso del bus de datos completo
79      AWVALID          <= AWVALID_i;
80      WVALID           <= WVALID_i;
81      ARVALID          <= ARVALID_i;
82      RREADY           <= RREADY_i;
83      BREADY           <= BREADY_i;
84      port_id_i        <= x"0" & port_id(3 downto 0) when k_write_strobe='1' else port_id; --Ignorar 4 MSB si se usa instrucción OUTPUTK
```

Figura 3.2. Parte declarativa y asignaciones concurrentes del CORE

3.3. Proceso registrado y FSMD

A continuación, está la parte más importante de este fichero: el proceso registrado que modela la FSMD. Como se trata de un diseño síncrono, el proceso solo incluye la señal de reloj CLK en su lista de sensibilidad y todas las acciones se llevan a cabo dentro de un bloque *rising_edge(CLK),* esto es, en el flanco de subida de la señal de reloj. En primer lugar, vienen las acciones del *reset* a nivel bajo *RSTn*:

- Configura todas las señales de *handshake* controladas por la interfaz a 0, de acuerdo con la especificación.
- Reinicia a 0 la señal *capturing.*
- Pone a 0 todos los bits de *in_port,* que actúa como registro de estado.
- Reinicia la máquina de estados a *IDLE.*

Se trata de un *reset* a nivel bajo debido a que así lo requiere el estándar AXI4-Lite, tal y como se describió en el capítulo anterior. Si no está el *reset* activado, entra en acción la máquina de estados con una sentencia *case* que modela el comportamiento del CORE para cada posible estado. Su comportamiento se muestra en el diagrama ASMD que aparece en las figuras 3.3, 3.4, 3.5 y 3.6.

Se ha dividido el diagrama entre todas esas figuras debido a su tamaño y complejidad.

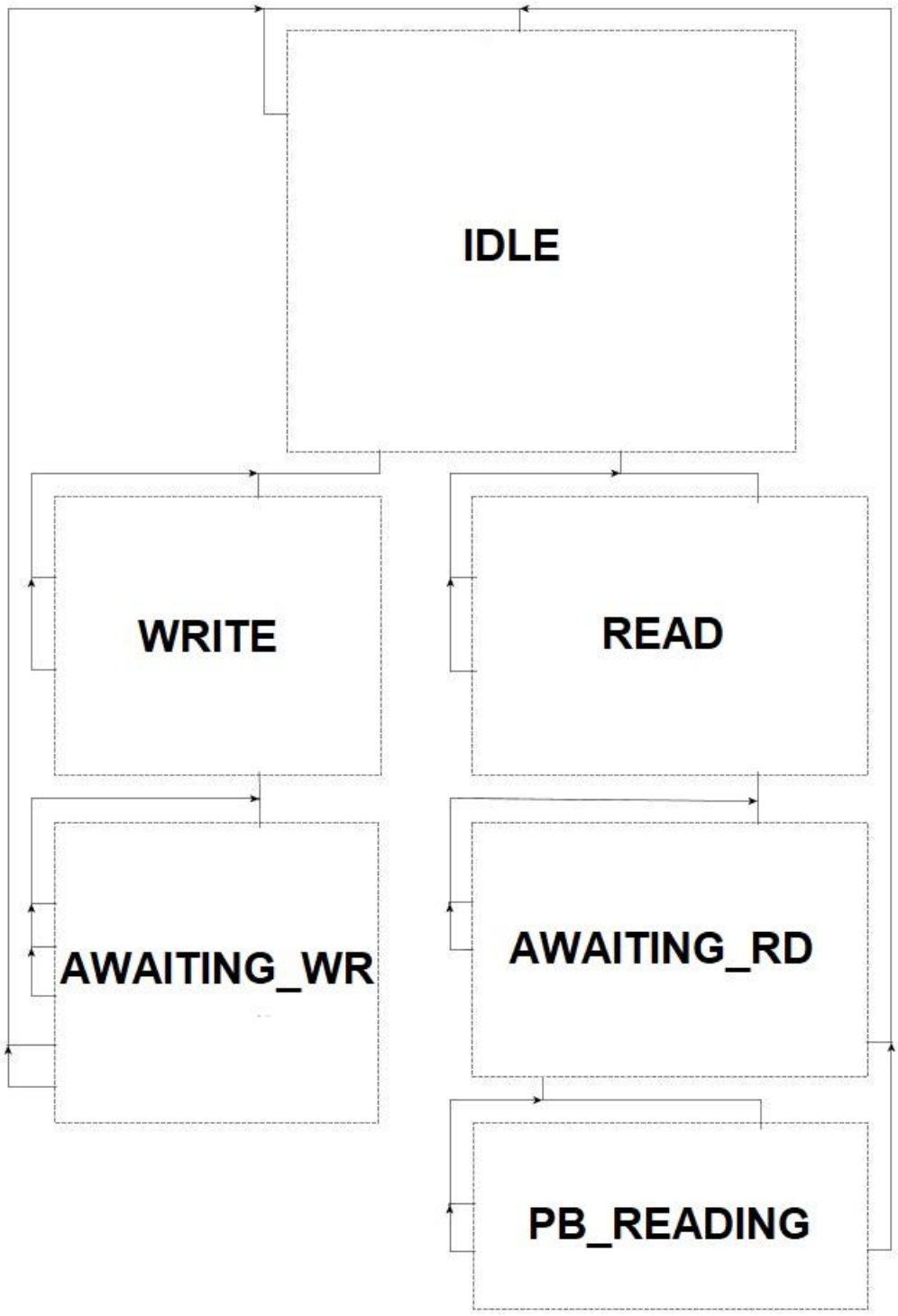


Figura 3.3. Relación entre los estados de la ASMD

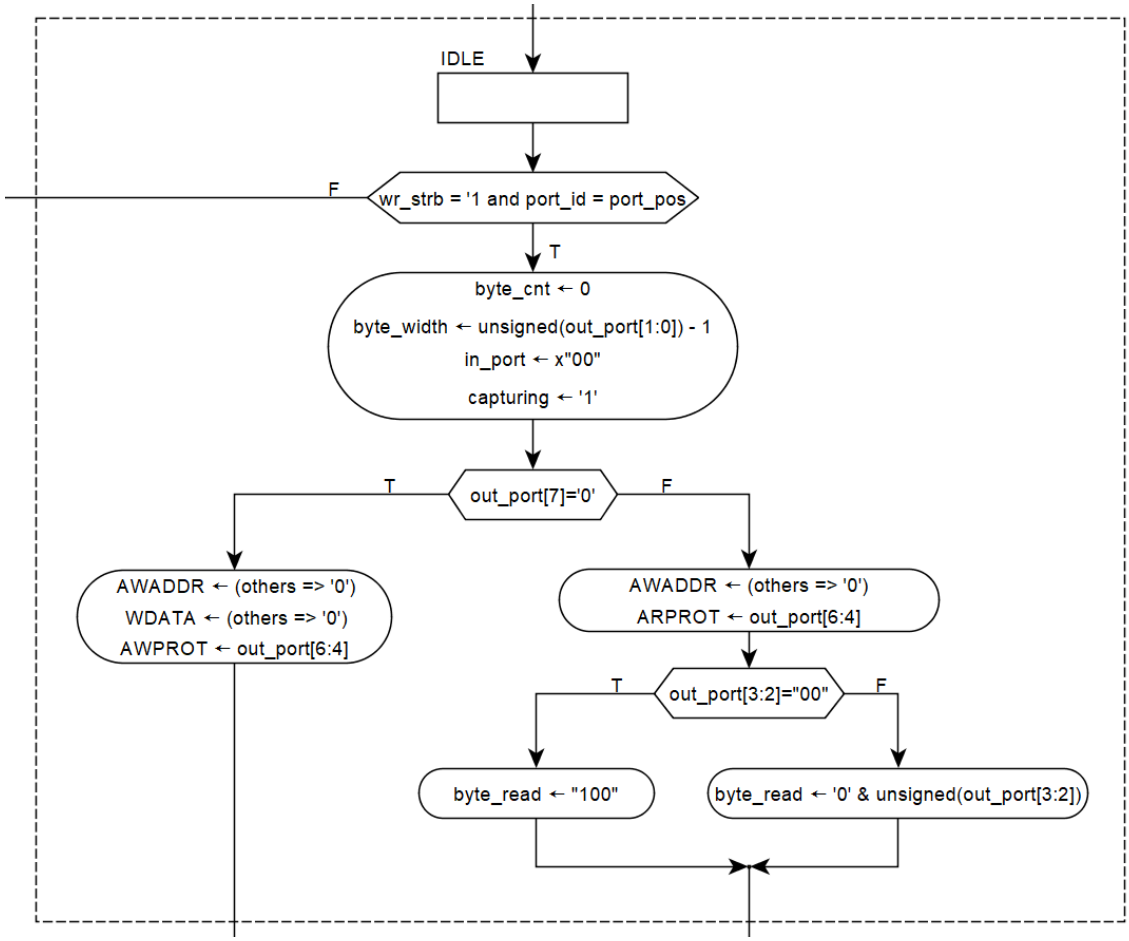


Figura 3.4. Estado *IDLE* de la ASMD

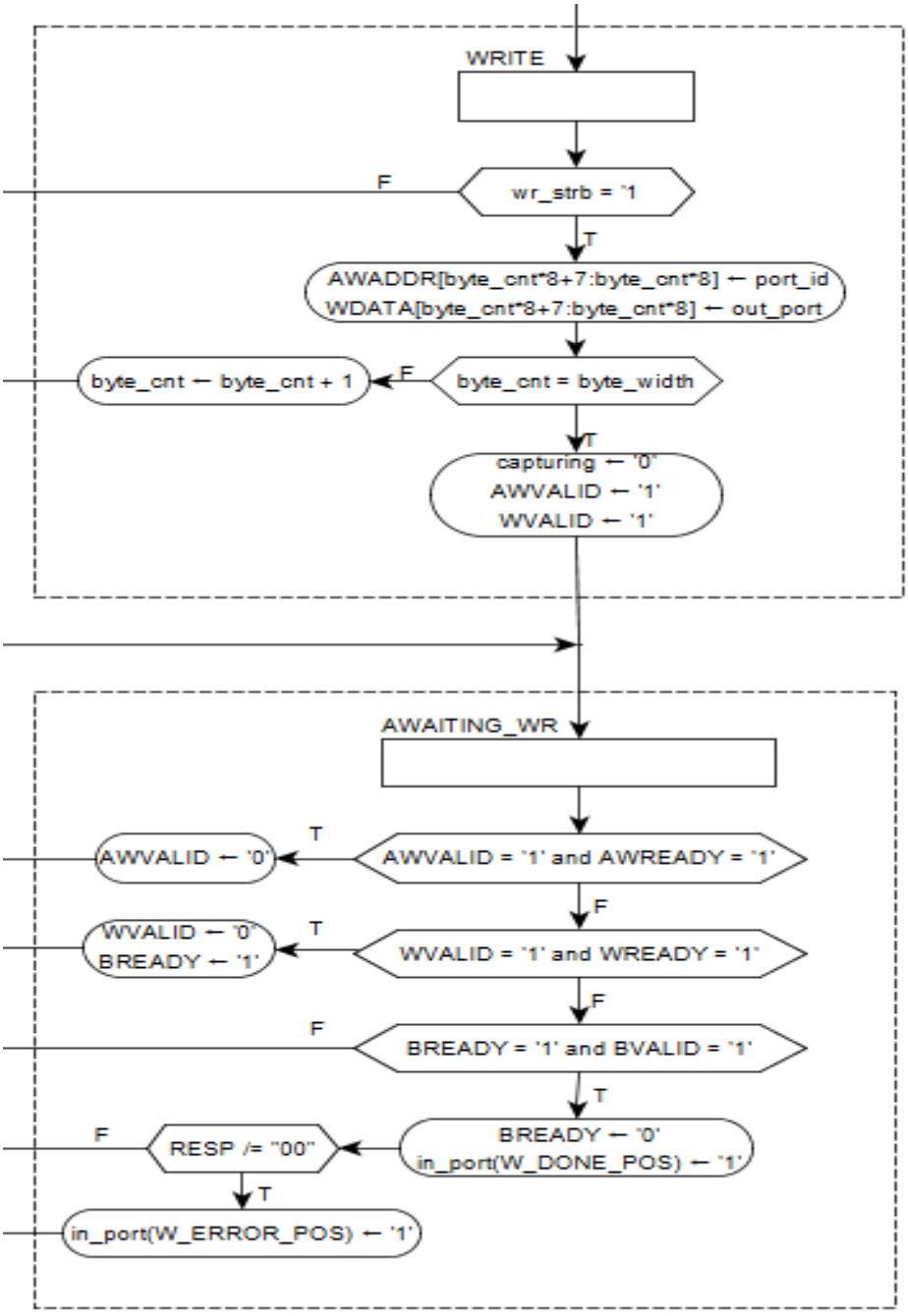


Figura 3.5. Estados *WRITE* y *AWAITING_WR* en el diagrama ASMD

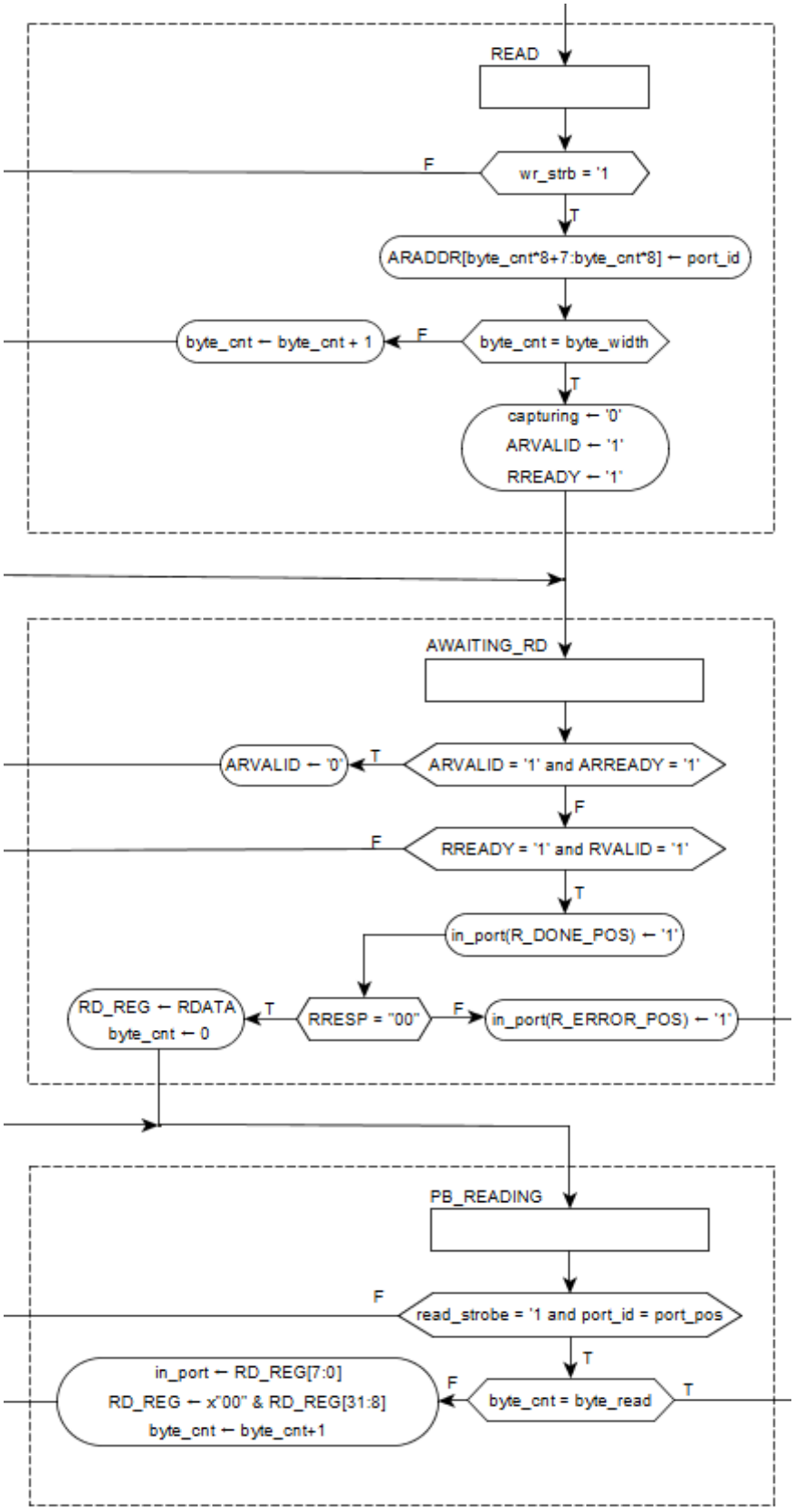


Figura 3.6. Estados *READ, AWAITING_RD* y *PB_READING* en el diagrama ASMD

A continuación, se explicará su funcionamiento al completo. Hay que aclarar que, cuando se menciona un puerto de salida controlado por una señal interna, en el código se hace uso de la señal interna. Lo mismo se aplica a *port_id,* que se hace uso de *port_id_i.*

La máquina empieza en el estado *IDLE* y espera a una escritura desde Picoblaze (con la señal *wr_strb)* a su dirección, la especificada en *port_pos.* Cuando se realiza dicha escritura, la máquina se pone en marcha: reinicia *byte_cnt* a 0, asigna a *byte_width* los dos últimos bits de *out_port* y les resta uno, reinicia el registro *in_port* a 0 y configura a 1 la señal *capturing*. A continuación, se procede a evaluar el bit 7 de *out_port,* el más significativo. Si está a 1, se va a realizar una operación de lectura, y en el caso contrario, será de escritura. Si es de escritura, se reinician a 0 *AWADDR* y *WDATA,* se asigna a *AWPROT* los bits 4 a 6 de *out_port* y se configura el estado siguiente a *WRITE.* En el caso de una lectura, se reinicia *ARADDR,* se asignan los 3 bits a *ARPROT* y el estado siguiente es *READ*. Además, a *byte_read* se le asigna el valor 0 seguido de los bits 3 y 2 de *out_port*, salvo que estos sean “00”, en cuyo caso el valor asignado es “100”. El *reset* y el estado *IDLE* son descritos tal y como aparecen en la figura 3.7:

```
86  process(CLK) begin
87    if rising_edge(CLK) then
88    if RSTn='0' then
89      AWVALID_i <= '0';
90      WVALID_i  <= '0';
91      ARVALID_i <= '0';
92      RREADY_i  <= '0';
93      BREADY_i  <= '0';
94      capturing <= '0';
95      in_port   <= x"00";
96      estado    <= IDLE;
97    else
98      case estado is
99        when IDLE =>                                          --Estado inicial
100       if wr_strb='1' and port_id_i=port_pos then            --Escritura a registro indica inicio de operación
101         byte_cnt <= 0;
102         byte_width <= unsigned(out_port(1 downto 0)) - 1;   --Cantitad de bytes esperada para bus de direcciones (resto a 0)
103         in_port <= x"00";
104         capturing <= '1';
105         if out_port(7)='0' then                             --Si operación de escritura
106           AWADDR <= (others=>'0');
107           WDATA <= (others=>'0');
108           AWPROT <= out_port(6 downto 4);
109           estado <= WRITE;
110         else                                                --Si operación de lectura
111           ARADDR <= (others=>'0');
112           ARPROT <= out_port(6 downto 4);
113           if out_port(3 downto 2)="00" then                 --Bytes que Picoblaze espera leer incluyendo registro de estado
114             byte_read<="100";
115           else
116             byte_read<="0" & unsigned(out_port(3 downto 2));
117           end if;
118           estado <= READ;
119         end if;
120       end if;
```

Figura 3.7. *Reset* síncrono y estado *IDLE* en VHDL

Si se ha configurado para una operación de escritura, el CORE espera a cualquier escritura de Picoblaze, sin importar la dirección. Cuando llega la primera escritura, asigna el valor de *port_id* al byte menos significativo de *AWADDR* y el valor de *out_port* al byte menos significativo de *WDATA*. Entonces, procede a comprobar si se han escrito los bytes necesarios comparando *byte_cnt* con *byte_width.* Aquí hay que explicar que el número esperado de lecturas es el valor de *byte_cnt* más uno: si vale “00”, se esperará una escritura, si vale “01”, dos escrituras, y así sucesivamente. Si no se ha llegado al final de cuenta, se incrementa en uno *byte_cnt,* y en la siguiente escritura, los bytes de datos y dirección serán asignados al byte menos significativo por encima del escrito anteriormente, es decir, el sistema funciona en modo *Big Endian*. Si, por el contrario, se ha llegado al final de cuenta, se configuran la señales *AWVALID* y *WVALID* a 1 para indicar al bus AXI4-Lite que el dato ya es válido, se pone a 0 la señal *capturing* y se define el estado siguiente como *AWAITING_WR.*

En el estado *AWAITING_WR*, la interfaz espera en primer lugar al *handshake* del canal de direcciones de escritura. Una vez que se ha producido, cambia *AWVALID* a 0 y espera al *handshake* de los datos. Una vez hecho, *WVALID* conmuta a 0 y *BREADY* se configura a 1 para poder recibir la respuesta de escritura. Cuando se produce el *handshake* de este canal, se cambia a 1 el bit de estado en *in_port* definido por *W_DONE_POS* para indicar a Picoblaze que la escritura ya ha terminado. Además, si el valor de *BRESP* es distinto de “00”, significa que ha habido un error, y se activa el bit señalado por *W_ERROR_POS* para que Picoblaze sepa de dicho error. Por último, se vuelve al estado IDLE a la espera de otra operación. Se muestra a continuación la descripción VHDL de estos dos estados en la figura 3.8:

```
121         when WRITE =>                                          --Recopilando dato y dirección a escribir
122           if wr_strb='1' then
123             AWADDR(byte_cnt*8+7 downto byte_cnt*8) <= port_id_i;--Captura byte a byte de menos a más significativo
124             WDATA(byte_cnt*8+7 downto byte_cnt*8) <= out_port;
125             if byte_cnt=byte_width then
126               estado <= AWAITING_WR;
127               capturing <= '0';
128               AWVALID_i <= '1';                                --Dato y dirección ya disponibles en el bus
129               WVALID_i <= '1';
130             else
131               byte_cnt <= byte_cnt + 1;
132             end if;
133           end if;
134         when AWAITING_WR =>                                      --Esperando a que se efectúe la escritura
135           if AWVALID_i = '1' and AWREADY = '1' then AWVALID_i <= '0';--Handshake de bus de direcciones
136           elsif WVALID_i = '1' and WREADY = '1'  then             --Handshake de bus de datos
137             WVALID_i  <= '0';
138             BREADY_i <= '1';                                     --Listo para recibir respuesta
139           elsif BREADY_i = '1' and BVALID = '1' then             --Handshake de respuesta
140             BREADY_i <= '0';
141             in_port(W_DONE_POS) <= '1';
142             estado <= IDLE;
143             if BRESP/="00" then in_port(W_ERROR_POS) <= '1'; end if; --Código distinto de "00" implica error en la escritura
144           end if;
```

Figura 3.8. Estados *WRITE* y *AWAITING_WR* en VHDL

Ahora se explica el caso de la operación de lectura que, en muchos aspectos, es similar al de escritura. En el estado *READ,* como en el *WRITE,* se esperan escrituras desde picoblaze sin importar la dirección. La señal *AWADDR* es sobrescrita en modo *Big Endian* tantos bytes como indique *byte_width* más uno. Si se ha llegado al final del ciclo, se desactiva *capturing*, se activan *ARVALID* y *RREADY* y se pasa al estado *AWAITING_RD.*

En el estado *AWAITING_RD,* se espera primero al *handshake* del canal de direcciones de lectura y, una vez hecho, se espera al de datos. Cuando se produce dicho *handshake,* se captura el dato de *RDATA* en *RD_reg* y se activa a 1 el bit de *in_port* señalado por *R_DONE_POS* para indicar la finalización de la lectura. Si ha habido algún error (*RRESP* distinto de “00”), se activa el bit señalado por *R_ERROR_POS* y se vuelve al estado *IDLE.* Si, por el contrario, no se ha producido ningún error, se reinicia otra vez el contador *byte_cnt*, porque volverá a usarse, y se salta al estado *PB_READING.* Los estados *READING* y *AWAITING_RD* son descritos tal y como aparecen en la figura 3.9:

```
145         when READ =>                                          --Recopilando dirección para leer
146           if wr_strb='1' then
147             ARADDR(byte_cnt*8+7 downto byte_cnt*8) <= port_id_i;   --Recibe dirección byte a byte, de menos a más significativo
148             if byte_cnt=byte_width then
149               estado <= AWAITING_RD;
150               capturing <= '0';
151               ARVALID_i <= '1';
152               RREADY_i <= '1';
153             else
154               byte_cnt <= byte_cnt + 1;
155             end if;
156           end if;
157         when AWAITING_RD =>                                   --Esperando recibir dato del registro
158          if ARVALID_i = '1' and ARREADY = '1' then ARVALID_i <= '0';--Handshake de bus de direcciones
159          elsif RREADY_i = '1' and RVALID = '1' then           --Handshake de bus de datos
160           RREADY_i <= '0';
161           RD_reg <= RDATA;
162           in_port(R_DONE_POS) <= '1';
163           if RRESP = "00" then                                --Si no ha habido error, Picoblaze puede leer
164              estado <= PB_READING;
165              byte_cnt <= 0;
166           else                                                --Si hay error, volver a estado inicial
167              in_port(R_ERROR_POS) <= '1';
168              estado <= IDLE;
169           end if;
170          end if;
```

Figura 3.9. Estados *READ* y *AWAITING_RD* en VHDL

El estado *PB_READING* es similar al de *READ* o *WRITE,* pero con Picoblaze haciendo lecturas en vez de escrituras. Cuando se llega a este estado, el CORE espera una lectura del procesador a su dirección, a la de *port_pos*. Cuando se realiza dicha lectura, se incrementa el contador *byte_cnt,* se carga en *in_port* el byte menos significativo del registro *RD_reg* y se desplaza éste ocho bits a la derecha. Así, se espera que Picoblaze lea *byte_read* bytes de los cuatro recibidos en *RDATA* siguiendo un esquema *Big Endian*, leyendo los bytes de menos a más significativos. Cuando se llega a fin de cuenta, la FSM vuelve al estado *IDLE*. La descripción VHDL de este estado se muestra en la figura 3.10:

```
171         when PB_READING =>                                --Enviando dato leido a Picoblaze
172           if read_strobe = '1' and port_id_i=port_pos then
173             if byte_cnt=byte_read then
174               estado <= IDLE;
175             else
176               in_port <= RD_reg(7 downto 0);                --Picoblaze lee byte a byte, empezando por el menos significativo
177               RD_REG <= x"00" & RD_REG(31 downto 8);
178               byte_cnt <= byte_cnt + 1;
179             end if;
180           end if;
181       end case;
182     end if;
183     end if;
184   end process;
185 end;
```

Figura 3.10. Estado *PB_READING* en VHDL

Capítulo 4. Guía de utilización

En este capítulo se ofrecerán guías y consejos para integrar y utilizar este módulo dentro de un sistema mayor. Este capítulo se desglosará en una parte hardware, donde se indica cómo conectar la interfaz con el resto del sistema, y la parte software, donde se explica cómo se deberían programar las rutinas de Picoblaze para hacer un uso eficiente del bus AXI4-Lite a través de nuestro CORE.

4.1. Parte hardware

Esta interfaz se ha diseñado para que sea fácil de conectar. El caso más simple es cuando ésta es el único periférico de Picoblaze. En este caso, es tan sencillo como conectar los puertos a aquellos de Picoblaze y AXI4-Lite con el mismo nombre, dejando el puerto *capturing* en circuito abierto tal y como se muestra en la figura 4.1:

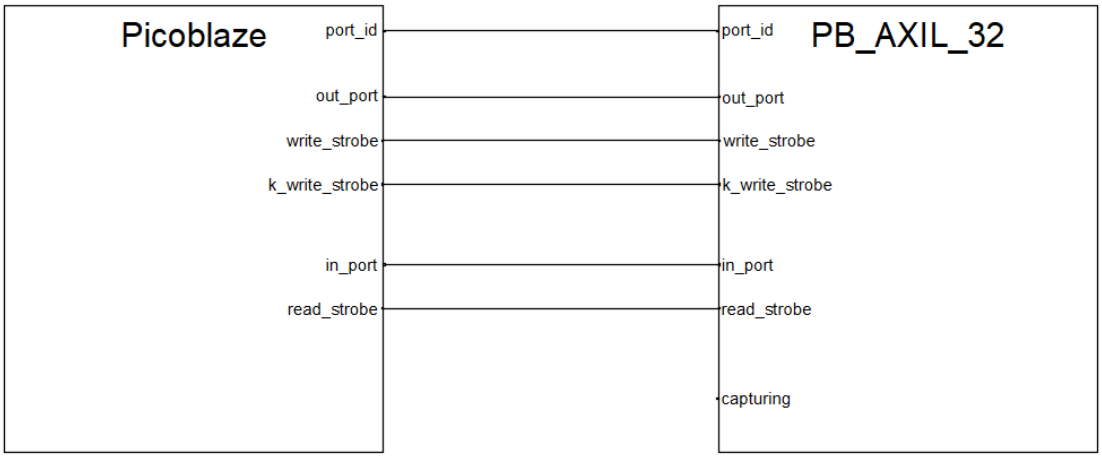


Figura 4.1. Configuración PB + interfaz

Si, por el contrario, va a convivir con otros periféricos de Picoblaze, han de tenerse en cuenta las siguientes dos consideraciones:

- Para los demás puertos de escritura, considerándose estos como registros habilitados por *write_strobe* o *k_write_strobe*, se sugiere utilizar la señal capturing como señal de habilitación a nivel bajo. Esto se puede hacer simplemente con la AND de la señal de habilitación convencional y la negada de *capturing*. La razón es para evitar escrituras accidentales mientras se cargan los datos en la interfaz.
- Respecto a los puertos de lectura, se puede multiplexar la señal *in_port* del CORE junto con los demás puertos de entrada. Sin embargo, es importante que no haya ninguna etapa registrada en el multiplexor, debido a que se pueden generar errores en la operación de lectura si el valor de *in_port* cambia en medio de una instrucción *INPUT*. También han de coincidir la dirección de *port_pos* con la dirección de la interfaz en el multiplexor.

En las figuras 4.2 y 4.3 se ilustran cómo deberían ser las conexiones:

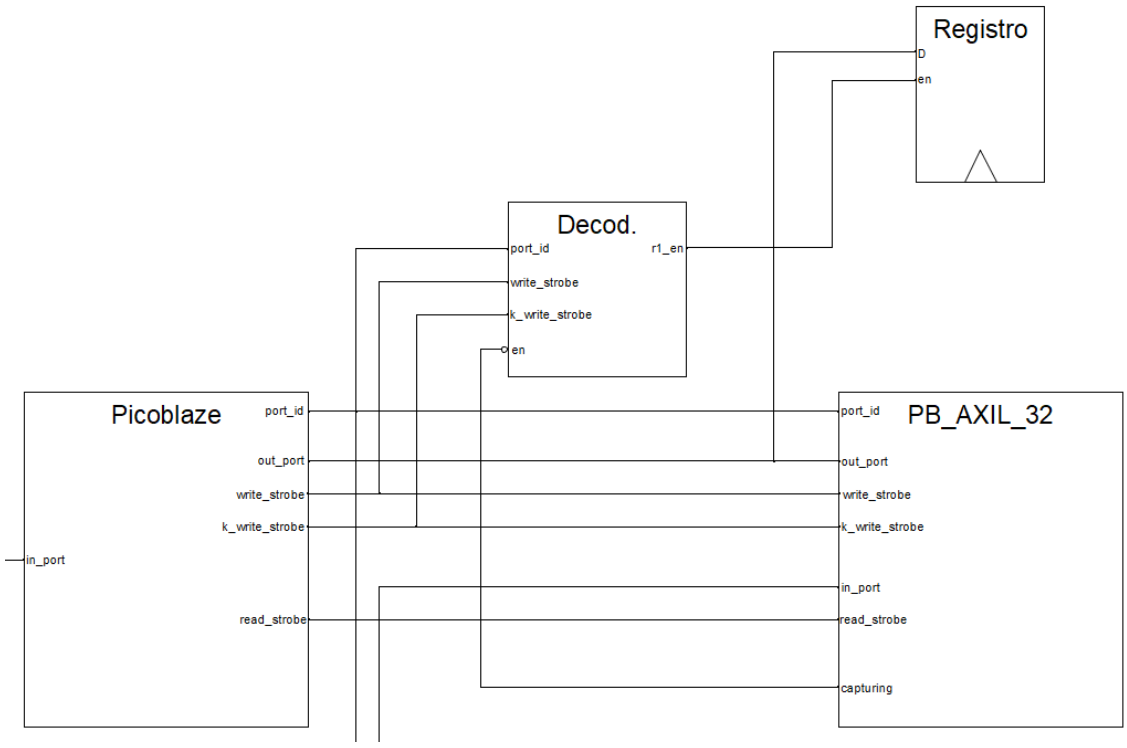


Figura 4.2. Configuración PB + interfaz + puerto de escritura

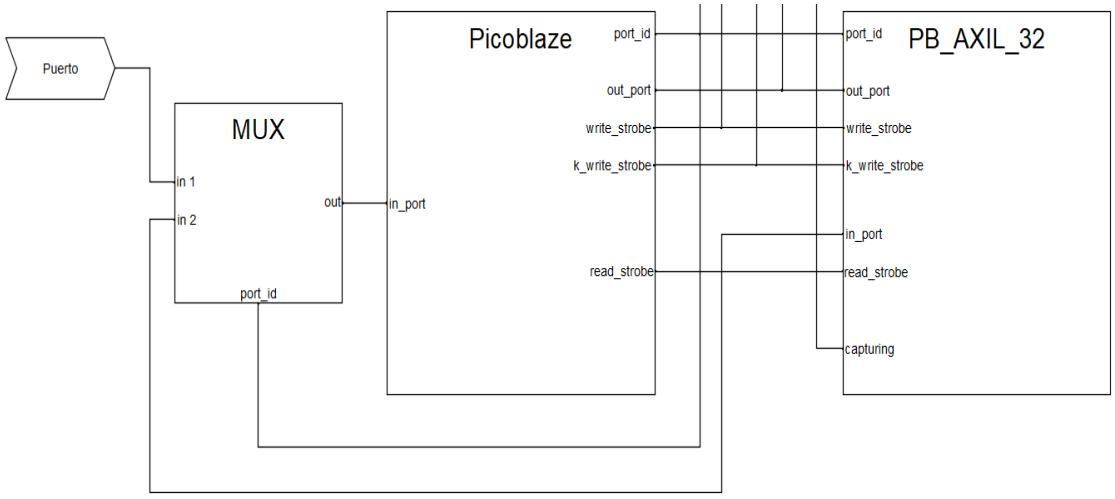


Figura 4.3. Configuración PB + interfaz + puerto de lectura

4.2. Parte software

Tal como se indicó en la sección 2.1, solo se puede programar Picoblaze en ensamblador. Para poder explicar el uso del CORE a nivel software, esto es, en el código ensamblador de Picoblaze, primero hay que listar sus bancos de registros y la función de cada bit dentro de ellos. La interfaz presenta a Picoblaze dos registros, uno de solo lectura y otro de solo escritura, ambos accesibles mediante la misma dirección, la especificada en el genérico *port_pos.*

El puerto de lectura, que es accesible en cualquier estado salvo en *PB_READING* tras la primera lectura, presenta la siguiente estructura:

Reservado	R_DONE	R_ERROR	W_DONE	W_ERROR
(bits 4 a 7)	(bit 3)	(bit 2)	(bit 1)	(bit 0)
(0)	(0)	(0)	(0)	(0)

Tabla 4.1. Registro de lectura del CORE

Los bits R_DONE y W_DONE indican cuando se ha completado una solicitud de lectura y escritura respectivamente, mientras que los bits R_ERROR y W_ERROR indican si ha habido un error en dicha operación de lectura o escritura. Este registro se inicializa a 0 al iniciarse, al activarse la señal de reset y al empezar una nueva operación de lectura o escritura.

El puerto de escritura, en el que solo se puede escribir en el estado *IDLE,* cuando no se está procesando ninguna solicitud, tiene el siguiente aspecto:

READ	AxPROT	BYTE_READ	BYTE_WIDTH
(bit 7)	(bits 6 a 4)	(bits 3 y 2)	(bits 1 y 0)

Tabla 4.2. Registro de escritura del CORE

Este puerto funciona tal que, cuando se hace una escritura a él, se inicia un proceso de lectura o escritura en el bus con la configuración escrita en dicho registro. El bit READ indica si la próxima operación va a ser de lectura (1) o escritura (0). Los 3 bits de AxPROT serán los que se carguen en la señal *ARPROT* o *AWPROT* dependiendo del tipo de operación. Estos bits configuran el nivel de acceso en el bus AXI4-Lite. Para más información, se ha de consultar la especificación. Los dos bits BYTE_READ solo son utilizados en la operación de lectura, e indican cuántas lecturas va a realizar Picoblaze cuando esté disponible el dato leído. Si se escribe “00”, se considera que se van a leer los 4 bytes del bus. Los dos bits BYTE_WIDTH indican cuántas escrituras se van a hacer tras escribir en este registro, siendo “00” cuatro escrituras.

El procedimiento para escribir a una dirección desde Picoblaze es el siguiente:

1) Escribir al puerto designado en *port_pos.* El bit 7 tendrá que ser ‘0’, los bits 4 a 6 serán los que se quiere escribir en *AWPROT* (dependiendo del periférico, se puede obviar) y en los bits 1 y 0 se escribirá el número de escrituras que se quiere hacer a la interfaz, siendo “00” cuatro escrituras.
2) A continuación, se harán tantas escrituras como se hayan especificado en los bits 1 y 0 de la siguiente forma: se empieza escribiendo el byte menos significativo del dato al byte menos significativo de la dirección, luego el siguiente menos significativo del dato al siguiente menos significativo de la dirección, y así sucesivamente. Es importante no escribir a ningún otro periférico hasta haber acabado este paso, puesto que dicho puerto no capturará el dato, y en su lugar lo capturará la interfaz.
3) Ahora, hay dos opciones: esperar a que el interfaz haya terminado la lectura, o continuar con la ejecución del programa de Picoblaze. La interfaz

no estará disponible para realizar otra operación hasta haber terminado la actual. Si se quiere saber si ha terminado, se ha de leer su puerto de lectura y testear el bit 1 (*W_DONE*). Cuando se activa a 1, la operación ha terminado y la interfaz vuelve a estar disponible. También conviene, cuando se activa *W_DONE,* testear el bit 0 (*W_ERROR*) para comprobar si se ha llevado a cabo la operación correctamente.

Vamos a mostrar un ejemplo: supongamos que queremos escribir el dato 0x00010203 a la dirección 0x00000405 con los bits de *AWPROT* configurados a “101”. Como el dato es más grande que la dirección y solo tiene 3 bytes útiles (el más significativo es todo ceros), la operación puede realizarse con solo 3 escrituras. Suponiendo que la dirección del CORE es la predeterminada (0x00), la operación podría hacerse como sigue:

```
; se carga el dato a transmitir en el par de registros [s2, s1
; s0]
LOAD s0, 03
LOAD s1, 02
LOAD s2, 01
OUTPUTK 01010011’b, 00; Se avisa a la interfaz de que se va a
; realizar una operación de escritura con AWPROT=”101” y se va a
; escribir 3 veces (“11”)
; Se escribe el dato a la dirección deseada, byte a byte de
; menos a más significativo
OUTPUT s0, 05
OUTPUT s1, 04
OUTPUT s2, 00
```

Ahora se puede seguir con otras tareas o esperar en un bucle a que el CORE haya terminado:

```
Wait_loop: INPUT 00, s0; Lee el registro de estado de la interfaz
           TEST s0, 00000010’b; Comprueba si ha terminado la escritura
           JUMP Z, Wait_loop; Sale del bucle si ha terminado
           TEST s0, 00000001’b; Si Z=0, error
```

La última instrucción es para saber si ha habido algún error. Si estas instrucciones se usan dentro de una subrutina, es recomendable hacer ese

TEST para activar el flag Z y que la rutina llamante sepa si la operación se ha llevado a cabo con éxito o no.

La operación de lectura es similar, aunque algo más complicada. Los primeros pasos son casi idénticos a la escritura:

1) Escribir a *port_pos*. Las principales diferencias son que hay que escribir un 1 en el bit 7, y en los bits 2 y 3 hay que escribir el número de lecturas a realizar. Si no se conoce la longitud esperada del dato, se recomienda escribir “00” para luego hacer cuatro lecturas y leer los cuatro bytes.
2) Realizar las escrituras igual que antes. La única diferencia es que solo se captura la dirección, por lo que el dato escrito es irrelevante.
3) Leer del registro de lectura para comprobar si se ha leído el dato o no. En las lecturas, el bit *R_DONE* que indica que se ha terminado la lectura es el bit 3, mientras que el bit que indica error (*R_ERROR*) es el 2. A diferencia de la escritura, es obligatorio comprobar si ha habido error en la lectura. Si lo ha habido, la operación ya ha terminado. Si no lo hay, se salta al paso siguiente.
4) Realizar tantas lecturas como se especificó en los bits 2 y 3 al registro de lectura. Lo que se está leyendo es el dato leído en AXI4-Lite, byte a byte, de menos a más significativo. El CORE volverá a estar disponible cuando se hayan realizado dichas lecturas.

Vamos con otro ejemplo para la lectura. Supongamos que queremos leer del registro 0x0000008E con los bits de *ARPROT* a “001” y sabemos que el dato tiene una longitud de 16 bits o 2 bytes. Una secuencia de instrucciones válida sería la siguiente:

```
            OUTPUTK 10011001’b, 00; Configurar la interfaz para lectura
            ;(1 escritura, 2 lecturas)
            OUTPUT  s0,  8E;  Se  escribe  a  la  dirección  deseada,  no
            ; importa el dato
            ; Mismo bucle de espera que para el caso de escritura
Wait_loop: INPUT s0, 00
            TEST s0, 00001000’b
            JUMP Z, Wait_loop
            TEST s0, 00000100’b
            JUMP NZ, end_read
            ; Se lee dos bytes del dato, de menos a más significativo
            INPUT s0, 00
            INPUT s1, 00
end_read:
```

Al terminar esta secuencia de instrucciones, si la lectura ha salido bien, se dispondrá del dato en el conjunto de registros [s1, s0]. Además, el flag *Z* permite saber a una rutina llamante si el dato en esos dos registros es válido.

Capítulo 5. Plan de pruebas y resultados

En este capítulo se describirán las pruebas hechas al CORE para comprobar su correcto funcionamiento. Se han hecho dos tipos de pruebas fundamentalmente: una simulación en ModelSim y la creación de un sistema generador de imagen controlado por UART donde se ha hecho uso de este CORE.

5.1. Simulación en ModelSim

Se trata de una simulación RTL del módulo interfaz. Todas las señales externas son generadas desde el fichero *PB_AXIL_32.do*, es decir, que son simuladas.

El primer paso de la simulación ha sido realizar un reset del módulo (1). Luego, se ha simulado la realización de una escritura del dato 0x0B30E007 a la dirección 0x0AF32202 con *AWPROT* con valor “101” (2) y simulando al terminar la operación un error con *BRESP* igual a “01” (3). El resultado aparece en la figura 5.1:

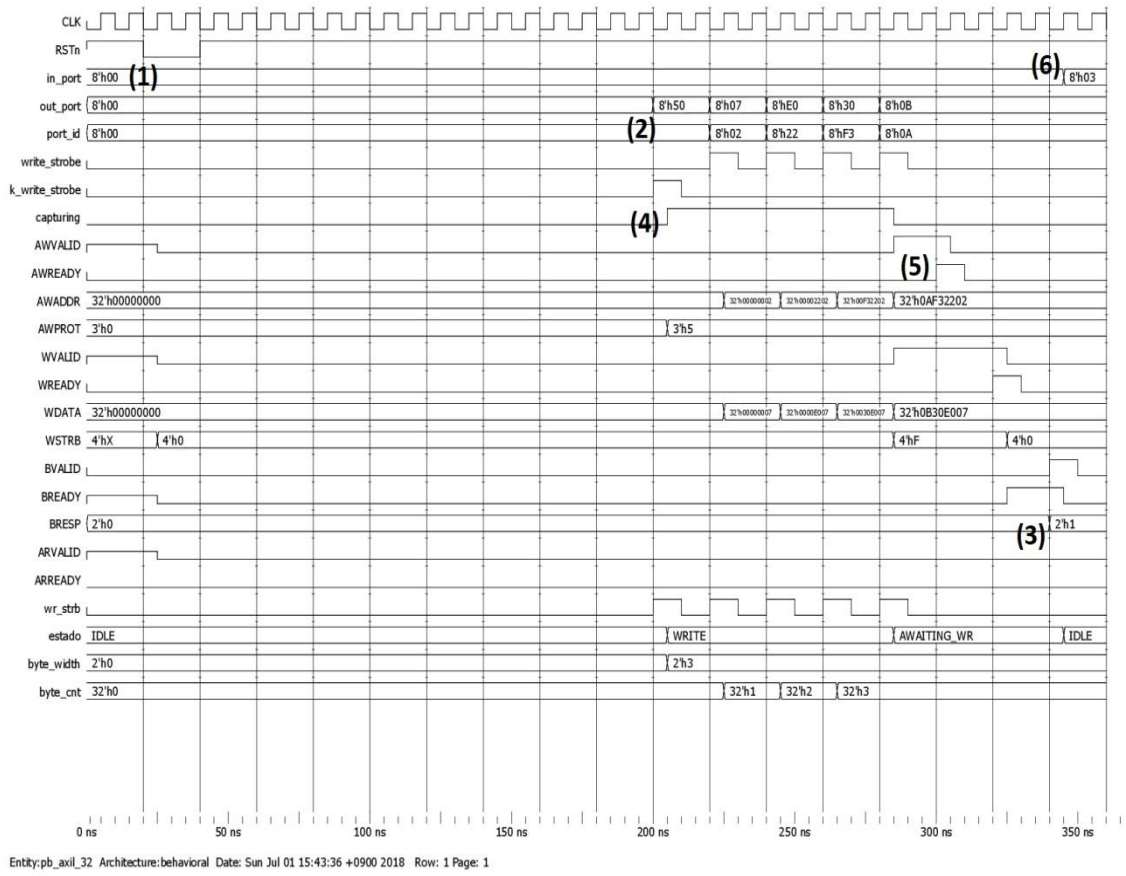


Figura 5.1. Simulación de *reset* y escritura de 4 bytes con error

En esta simulación se observa claramente la función de *capturing*: señalizar cuándo se está capturando el dato para evitar escrituras erróneas (4). También se observa el funcionamiento del *handshake* y que la interfaz responde como debe (5). Por último, se muestra la salida de *in_port* al haber terminado con un error: 0x03 (6).

En la figura 5.2, se simula una operación de lectura interrumpida por un reset (1):

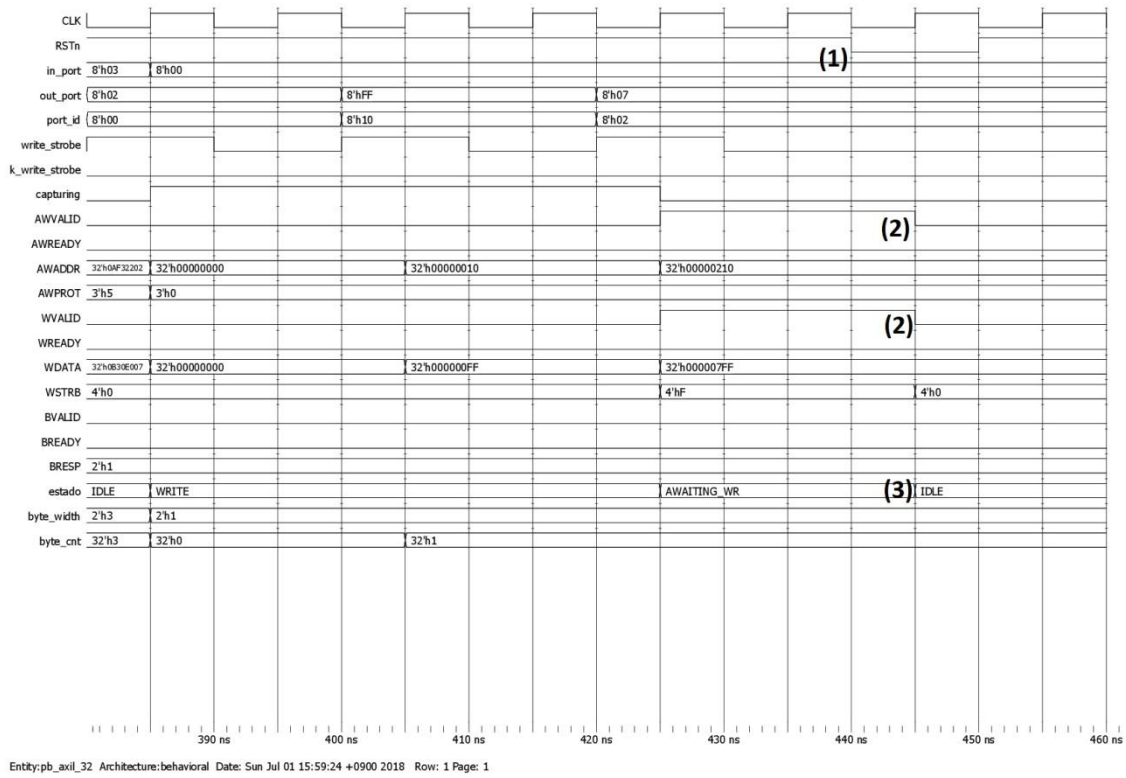


Figura 5.2. Simulación de escritura interrumpida por *reset*

Como se puede observar en dicha figura, la señal de *reset* actúa sobre las señales *VALID* y *READY* controladas por la interfaz, desactivándolas como se especifica en el estándar (2), y reiniciando el estado a *IDLE* (3).

Por último, en la figura 5.3 se realiza una operación de lectura en la dirección 0x640 (1), se obtiene el dato 0x0A0B0C0D en el bus (2) y finalmente éste es leído por Picoblaze (3):

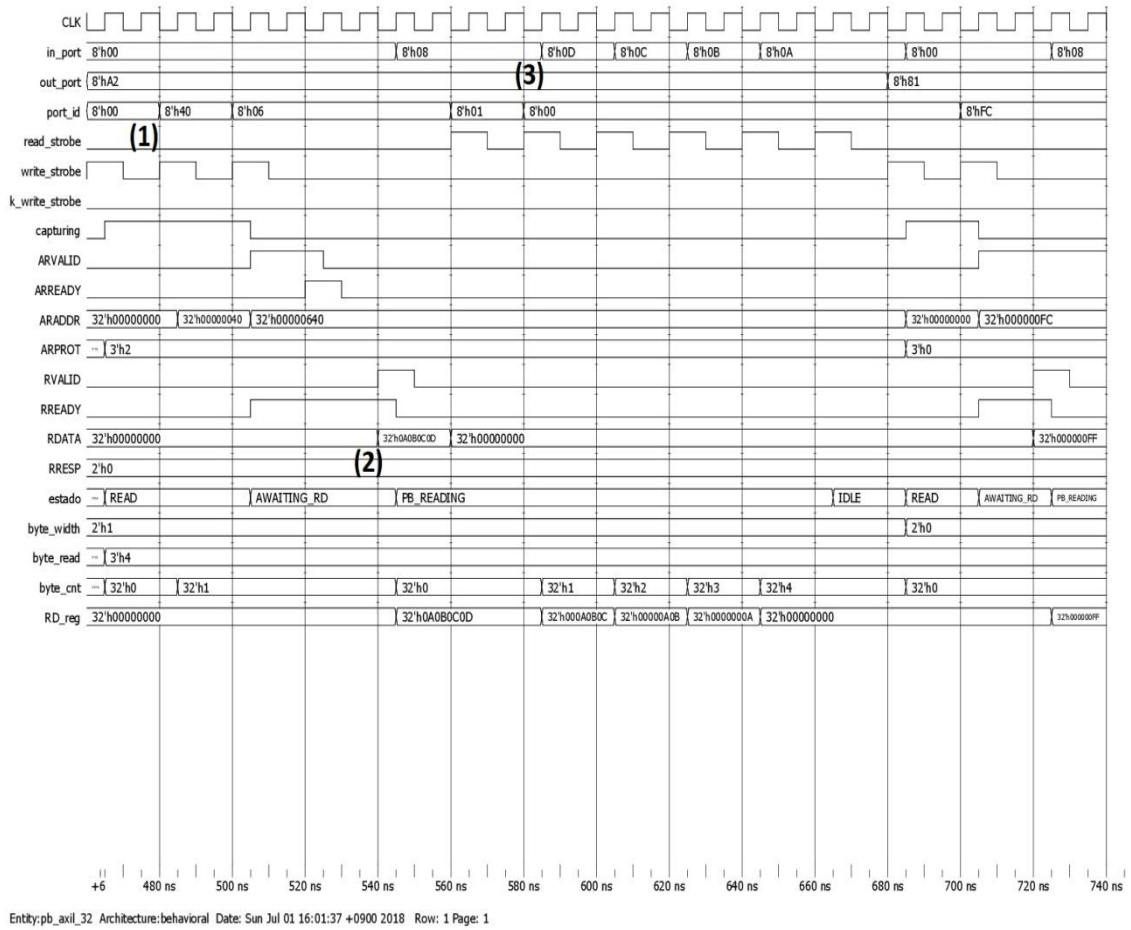


Figura 5.3. Simulación de lectura

5.2. Integración en un sistema completo

Para esta prueba, se ha usado el CORE como parte de un sistema completo. Su función es la de generar un patrón de video y transmitirlo a un monitor por HDMI. Dicho patrón se puede cambiar a través de un terminal de comandos en UART. El fichero *top* de este sistema es TOP_PB-AXIL_HDMI.vhd. El sistema está compuesto por los siguientes bloques:

- Un DCM que proporciona la señal de reloj a la frecuencia de píxel para una transmisión 1080p y la frecuencia de salida para HDMI.
- Un transmisor y un receptor UART conectados a Picoblaze que hacen uso de un bloque generador de baudios. Está incluido en un proyecto de ejemplo de Picoblaze.

- Picoblaze y su memoria de programa. No se hace uso del *reset* de la memoria porque presentaba un comportamiento errático que bloqueaba el sistema incluso haciendo uso del reset externo [6].
- Un bloque diseñado con varias IP de Xilinx (*my_TPG*) para generar una señal de video a 1080p y que puede ser controlado por un bus AXI4-Lite. Dicho bloque hace uso de las siguientes IP:
    - *AXI4 Interconnect* para poder conectar dos esclavos al bus AXI4-Lite de entrada
    - *Test Pattern Generator* (TPG) para generar la señal de video. Se puede controlar mediante AXI4-Lite y usa AXI-Stream como interfaz de salida.
    - *Video Timing Controller* (VTG) que genera las señales de sincronismo a 1080p.
    - *AXIS2Vid* que traduce el flujo de AXI-Stream a una señal RBG haciendo uso de las señales de sincronismo.
- Un codificador y transmisor HDMI que codifica una señal RGB a una preparada para ser transmitida por un puerto HDMI a un televisor o monitor [7].

En la figura 5.4 se muestra un esquema simplificado del sistema:

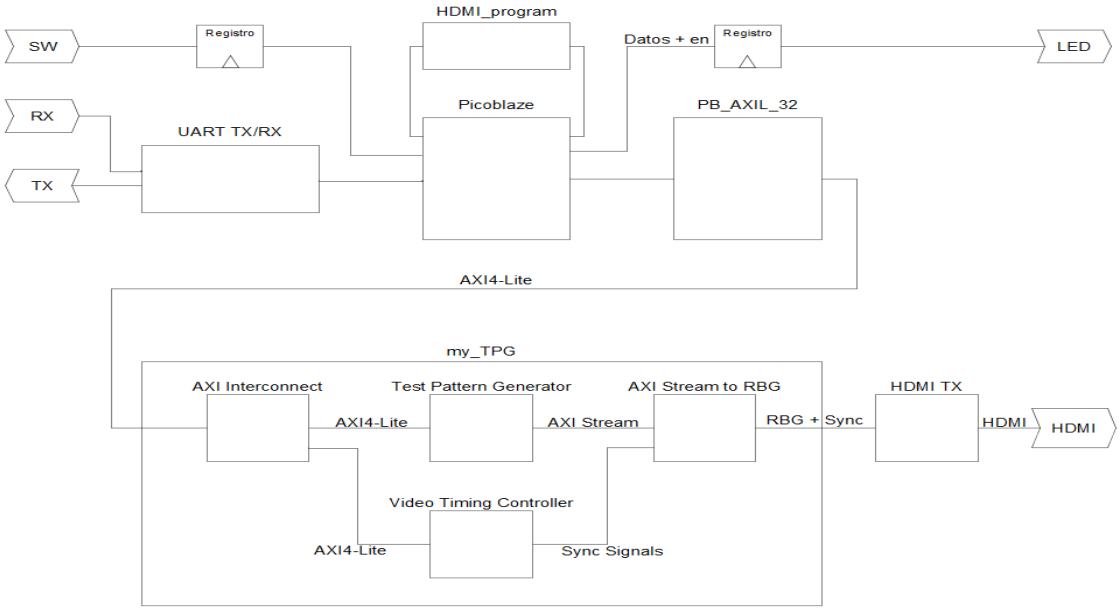


Figura 5.4. Diagrama simplificado del sistema completo

Este sistema se ha sintetizado e integrado en una placa Zybo, por disponer de conector HDMI. Aunque dicha placa cuenta con una parte PS ya incorporada con interfaz AXI4-Lite, se ha utilizado Picoblaze y nuestro CORE debido a que el objetivo final de nuestro sistema es la comprobación y demostración del funcionamiento del CORE.

Debido a que los pines UART integrados en la placa Zybo están conectados a la parte de sistema programable (PS), se ha hecho uso de un conversor USB a UART externo con los pines rx y tx del sistema conectados a los puertos JB02 y JB00 del puerto PMOD B respectivamente. Todos los módulos utilizan la señal MCLK a 148.5MHz excepto el CORE transmisor HDMI, que también usa la señal HDMI_DDR_CLOCK a 742.5MHz. Como señal de *reset*, se ha usado la señal negada de *locked* del DCM, habiendo conectado el botón 0 a su puerto de *reset*.

También es necesario especificar los puertos de E/S usados en Picoblaze. Respecto a los puertos de entrada:

- 0x00: registro de lectura de la interfaz con AXI4-Lite.
- 0x01: carácter recibido por UART.
- 0x02: registro de estado del transmisor y receptor UART
- 0x03: entrada registrada de los *switches*.

Los puertos de salida:

- 0x00: registro de escritura de la interfaz con AXI4-Lite.
- 0x01: dato a transmitir por UART.
- 0x03: registro de los LED de la placa Zybo.

También hay que mencionar que, siguiendo la guía del apartado 4.1, la escritura a la UART y a los LEDs necesita estar habilitada por la señal negada de *capturing* para evitar escrituras erróneas. Por último, respecto al bus AXI4-Lite, los registros del TPG usan un offset de dirección de 0x00, mientras que el VTG usa uno de 0x1000. Esto significa que, para acceder al registro 0x12 del

VTG, por ejemplo, se ha de escribir en la dirección el valor 0x1000 + 0x12 = 0x1012.

Ahora es necesario explicar también el programa de Picoblaze usado en este sistema, el cual está escrito en ensamblador en el fichero HDMI_program.psm. En la figura 5.5 se muestra el diagrama de flujo que define su funcionamiento:

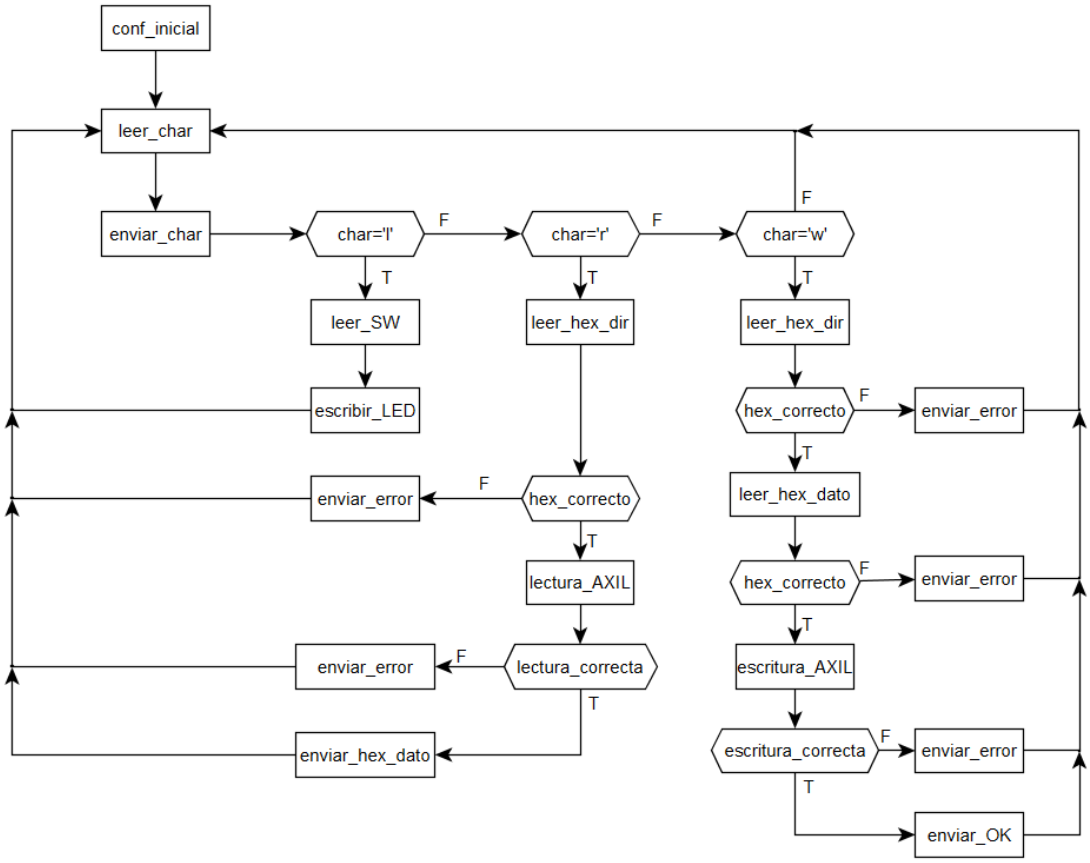


Figura 5.5. Flujograma de *HDMI_program.psm*

El programa principal llama en primer lugar a una rutina de inicialización que configura el sistema para que genere una señal de video senoidal por defecto, además de hacer varias lecturas para comprobar el funcionamiento en la simulación. Para ello, hace uso de unas subrutinas preparadas para usar la interfaz AXI4-Lite desarrollada. Al terminar esa inicialización, el programa espera a recibir comandos por UART. Los comandos aceptados son los siguientes:

- ‘r’: este comando permite leer el valor de cualquier registro AXI4-Lite. Su uso es de la forma ‘r’ + (espacio) + dirección_hex + (espacio). Si se ha realizado bien la lectura, aparece el valor completo en hexadecimal que ha devuelto el bus. De lo contrario, muestra ERR. Por ejemplo, digamos que queremos ver qué colores se están enmascarando en el TPG. De acuerdo a su manual, hay que leer los 3 primeros bits del registro 0x30 [8]. Para ello, hay que escribir “r 30 ”, espacio incluido y sin las comillas. Así, nos devolverá 00000002, por ejemplo, lo que significa, de acuerdo al manual, que se está enmascarando el color verde.
- ‘w’: este comando permite escribir cualquier valor a cualquier registro AXI4_Lite. Su uso es de la forma ‘w’ + (espacio) + dirección_hex + (espacio) + dato_hex + (espacio). Si se ha llevado a cabo la operación con éxito, se muestra OK en la pantalla. De lo contrario, se muestra ERR. Digamos, por ejemplo, que queremos cambiar el patrón mostrado a uno de barras. Para ello, tenemos que escribir el dato 0x9 al registro 0x20. Para ello, se puede escribir “w 20 9 ”, con las mismas normas que antes.
- ‘l’: este comando simplemente lee la disposición de los *switches* y lo escribe en el registro de los LED. Se ha puesto para testear la convivencia de la interfaz con otro periférico. Se usa de la forma ‘l’ + (espacio), es decir, se escribe en consola “l ” simplemente.

Hay que advertir que el diseño de esta consola es muy simple, por lo que apenas tolera fallos al escribir los comandos. Tampoco permite hacer retroceso si ha habido alguna equivocación. La ventaja es que reconoce cuándo se ha introducido un carácter incorrecto en el valor de dirección o dato. Si se ha escrito un valor o comando incorrecto, se puede escribir un mal carácter a propósito para que el sistema lo ignore y espere al siguiente comando en otra línea.

Para desarrollar este programa, se han hecho uso de varias rutinas modificadas que se encontraban en uno de los proyectos de ejemplo de Picoblaze con UART. Las demás se han desarrollado como parte de este proyecto.

Capítulo 6. Conclusiones y trabajo futuro

6.1. Objetivos y logros alcanzados

Se ha desarrollado un CORE capaz de hacer uso de todo el potencial de AXI4-Lite de una forma fácil y rápida desde el punto de vista de Picoblaze. Este CORE destaca por:

- Permitir el uso de periféricos configurables con AXI4-Lite en FPGA carentes de CPU ARM propia, las cuales tienen su propio interfaz incorporado, con el consiguiente ahorro de dinero y recursos internos.
- Disponer de 32 bits para direcciones, que cubre la práctica totalidad de los casos en los que el uso de Picoblaze es factible.
- Ser totalmente síncrono y tener un comportamiento temporal totalmente definido.
- Ser sintetizable y portable a cualquier FPGA.
- Ser compatible con periféricos con interfaz AXI4 esclava. Basta con conectar los buses y señales comunes y dejar los demás a un valor fijo.

Respecto a la utilización de recursos, el módulo puede presumir de una utilización eficiente de los recursos de la FPGA para la funcionalidad que ofrece. De acuerdo al informe de utilización de Vivado, el cual se muestra en la figura 6.1, el CORE utiliza 141 LUT como lógica y 147 FF agrupados en 54 SLICE. Hay que recordar que tan solo 128 FF son utilizados en los buses de direcciones y datos.

Name	Slice LUTs (17600)	Slice Registers (35200)	F7 Muxes (8800)	F8 Muxes (4400)	Slice (4400)	LUT as Logic (17600)	LUT as Memory (6000)	LUT Flip Flop Pairs (17600)	Block RAM Tile (60)	DSPs (80)	Bonded IOB (100)	OLOGIC (100)
TOP_PB_AXIL_HDMI	3381	4629	162	8	1703	3330	51	1513	4	11	22	7
HDMI_CORE (CORE_HDMI)	178	74	9	0	55	178	0	52	0	0	0	7
HDMI_program_i0 (HDMI_program)	1	3	0	0	1	1	0	0	0.5	0	0	0
MMCM (clk_wiz_0)	0	0	0	0	0	0	0	0	0	0	0	0
my_interf (PB_AXIL_32)	141	147	0	0	54	141	0	100	0	0	0	0
my_TPG_i0 (my_TPG)	2883	4264	153	8	1556	2867	16	1252	3.5	11	0	0
Picoblaze (kcpsm6)	127	77	0	0	52	103	24	58	0	0	0	0
uart_baud_gen_x16 (uart_baud_gen)	7	8	0	0	3	7	0	5	0	0	0	0
uart_rx_module (uart_rx6)	26	24	0	0	13	19	7	13	0	0	0	0
uart_tx_module (uart_tx6)	19	24	0	0	10	15	4	16	0	0	0	0

Figura 6.1. Informe de recursos de la FPGA utilizados por el sistema

Además, se ha probado dentro de un sistema real con éxito. La combinación de Picoblaze, UART, programa en ensamblador desarrollado e interfaz también puede utilizarse con cualquier otro bloque controlado por AXI4-Lite. Es útil sobre todo para depuración, ya que se puede controlar desde un terminal qué se está escribiendo, en dónde y cuándo.

Para alcanzar estos logros, ha sido necesario estudiar el uso y funcionamiento de Picoblaze, su sistema de E/S y su juego de instrucciones, así como aprovechar los proyectos de ejemplo de Xilinx. También ha sido necesario estudiar el estándar AXI4-Lite de ARM. Por último, se ha hecho uso de la metodología RT para desarrollar un CORE que funciona en base a un algoritmo secuencial y cuyo comportamiento real post-síntesis corresponde con el simulado.

6.2. Posibles mejoras para el futuro

En cuanto a posibles líneas de trabajo futuras, se podría implementar:

- Lectura y escritura simultáneas, debido a que sus canales son independientes entre ellos.
- FIFO en los buses para almacenar operaciones pendientes y resultados.

- Versión de 64 bits del CORE, es decir, que el ancho de los buses de datos sea de 64 bits. La norma AXI4-Lite contempla el uso de ese ancho de bus.

Hay que señalar que todas estas mejoras traerían consigo un coste de recursos adicional, además de un coste en términos de instrucciones y ciclos en el caso de la versión de 64 bits. Además, los periféricos suelen ser más rápidos que Picoblaze. Sería necesario comparar los costes con la ganancia de rendimiento en cada caso.

Referencias

[1] Xilinx, “PicoBlaze for Spartan-6, Virtex-6, 7-Series, Zynq and UltraScale Devices (KCPSM6)”, septiembre 2014, URL: https://www.xilinx.com/products/intellectual-property/picoblaze.html#design

[2] ARM, “AMBA® AXI™ and ACE™ Protocol Specification”, octubre 2014, URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022d/index.html

[3] Xilinx, “7 Series FPGAs Data Sheet: Overview”, febrero 2018, URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[4] Digilent, “ZYBO™ FPGA Board Reference Manual”, febrero 2017, URL: https://reference.digilentinc.com/_media/reference/programmable-logic/zybo/zybo_rm.pdf

[5] P. P. Chu, *RTL Hardware Design Using VHDL, Coding for Efficiency, Portability and Scalability,* Hovoken, New Jersey: Wiley Interscience, 2006

[6] Xilinx, “Ultra-Compact UART Macros for Spartan-6, Virtex-6 and 7-Series with PicoBlaze (KCPSM6) Reference Designs”, septiembre 2014, URL: https://www.xilinx.com/products/intellectual-property/picoblaze.html#design

[7] J. D. Heredia, "Desarrollo de controlador para interfaz de video de alta definición en FPGA”, Proyecto Fin de Carrera, Universidad de Málaga (España), Octubre, 2017.

[8] Xilinx, “Video Test Pattern Generator v7.0”, URL: https://www.xilinx.com/support/documentation/ip_documentation/v_tpg/v7_0/pg103-v-tpg.pdf