UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA EMBEBIDO BASADO EN MICROBLAZE PARA LA ADQUISICIÓN DE DATOS EN TIEMPO REAL MEDIANTE UN ACELERÓMETRO ADXL345"

TRABAJO FIN DE GRADO

Junio -2024

AUTOR: Marco Pérez Rodríguez

DIRECTOR: Roberto Gutiérrez Mazón

RESUMEN

El presente proyecto se centra en el desarrollo de un sistema embebido capaz de adquirir datos en tiempo real mediante un sistema basado en MicroBlaze. El objetivo principal es comunicarse con el acelerómetro ADXL345, integrado en el kit de desarrollo EVAL-ADXL345Z-DB de Analog Devices mediante I^2C para realizar mediciones de aceleración en tres ejes y transmitirlas al usuario en tiempo real. Además, se busca guardar los datos en un sistema de almacenamiento, en este caso una tarjeta MicroSD, aunque este aspecto no se logró ejecutar con éxito.

Se consideraron funcionalidades avanzadas del ADXL345, como la calibración de offset a nivel de registro y la gestión de interrupciones para detectar eventos específicos.

En resumen, este trabajo se enfoca en implementar un sistema eficiente para la adquisición y procesamiento de datos de aceleración tanto estática como dinámica en tiempo real, utilizando tecnología FPGA y un microprocesador embebido. Este sistema tiene posibles aplicaciones en monitoreo y análisis de movimiento.

Palabras clave: ADXL345, acelerómetro, FPGA, MicroBlaze, embebido, I^2C, SPI, Artix-7.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

GLOSARIO DE TÉRMINOS

AXI (Advanced eXtensible Interface) : protocolo de comunicación para buses que interconexiona componentes en sistemas integrados.

Bare-metal: programación directa sobre el hardware sin un sistema operativo intermedio.

Bitstream: archivo binario que contiene la configuración específica del diseño FPGA para programarla basándose en el hardware.

Constraints: restricciones que se aplican en el diseño de *hardware*.

DDR2 SDRAM (Double Data Rate 2 Synchronous Dynamic Random-Access Memory): tipo de memoria RAM síncrona que transfiere datos dos veces por ciclo de reloj.

Diafonía (Crosstalk): interferencia electromagnética entre circuitos adyacentes en una PCB, que puede causar distorsión de la señal.

FAT (File Allocation Table): Sistema de archivos desarrollado por Microsoft, utilizado para organizar y gestionar datos en dispositivos de almacenamiento como discos duros, memorias USB y tarjetas de memoria.

FPGA (Field-Programmable Gate Array): dispositivo integrado que puede ser configurado para realizar diversas funciones lógicas.

Flash (memoria): tipo de memoria no volátil que puede ser borrada y reprogramada eléctricamente.

GPIO (General-Purpose Input/Output): pines de propósito general en un microcontrolador que pueden ser configurados como entradas o salidas.

HyperRAM: tipo de memoria RAM diseñada para ofrecer alta velocidad y baja latencia en aplicaciones integradas.

I^2C (Inter-Integrated Circuit): protocolo de comunicación en serie que permite

la conexión de múltiples dispositivos en un solo bus.

IP (Intellectual Property): bloques lógicos reutilizables que pueden ser integrados en diseños digitales.

JTAG (Joint Test Action Group): estándar para la verificación y prueba de circuitos integrados mediante puertos de depuración.

Jumper: dispositivo usado para cerrar o abrir circuitos temporalmente en una PCB, configurando parámetros de funcionamiento sin necesidad de soldaduras.

MicroSD: tarjeta de memoria flash utilizada principalmente en dispositivos móviles y sistemas embebidos para almacenamiento adicional.

PCB (Printed Circuit Board): placa de circuito impreso que proporciona un soporte mecánico y eléctrico para el montaje de componentes electrónicos.

Pmod (Peripheral Module): estándar de interfaz para conectar periféricos a una FPGA o SoC. Desarrollado por Digilent.

PWM (Pulse Width Modulation): técnica de modulación de ancho de pulso utilizada para controlar la potencia suministrada a dispositivos electrónicos.

Puerto COM: interfaz de comunicación serie utilizada en ordenadores para conectarse con dispositivos periféricos.

Sistema embebido: sistema basado en un microprocesador o un microcontrolador diseñado para realizar una o algunas pocas funciones dedicadas.

SoC (System on Chip): circuito que integra todos o casi todos los componentes de un ordenador u otro sistema electrónico en un solo chip.

Soft processor: procesador implementado mediante lógica programable en una FPGA.

SPI (Serial Peripheral Interface): protocolo de comunicación serie síncrono utilizado para la transferencia de datos entre microcontroladores y periféricos.

SRAM (Static Random-Access Memory): tipo de memoria volátil que utiliza circuitos biestables para almacenar datos.

UART (Universal Asynchronous Receiver-Transmitter): protocolo de comunicación serie asíncrono.

Verilog: lenguaje de descripción de hardware usado para modelar sistemas electrónicos.

Wishbone: tipo de bus de interconexión entre diferentes módulos dentro de un sistema integrado.

1. Introducción

La adquisición de datos en tiempo real es crucial en numerosos campos debido a su capacidad para proporcionar información actualizada y precisa sobre distintos fenómenos. Permite tomar decisiones rápidas en ámbitos que requieren monitoreo continuo y respuesta inmediata como la automoción, el análisis médico o las aplicaciones industriales. Esta capacidad de procesamiento inmediato es fundamental para optimizar operaciones y mejorar la seguridad.

1.1. Motivación

En el laboratorio, contábamos con la placa de evaluación EVAL-ADXL345Z-DB [1] de Analog Devices (Figura 1.1) usada para medir la aceleración en tres ejes con el acelerómetro ADXL345. El propósito para el que fue diseñada esta placa es poder probar el *software* que se diseñe para usar el ADXL345 mientras se está trabajando y no se tiene finalizada la parte *hardware* de nuestro proyecto. Esto es útil para reducir el tiempo de desarrollo en aplicaciones en las que se use este acelerómetro.

La placa está preconfigurada como un registrador de datos que se puede utilizar para, en general, familiarizarse con la comunicación del acelerómetro. Puede ser alimentada por dos baterías AAA y reprogramada, integrándose perfectamente en aplicaciones portátiles.

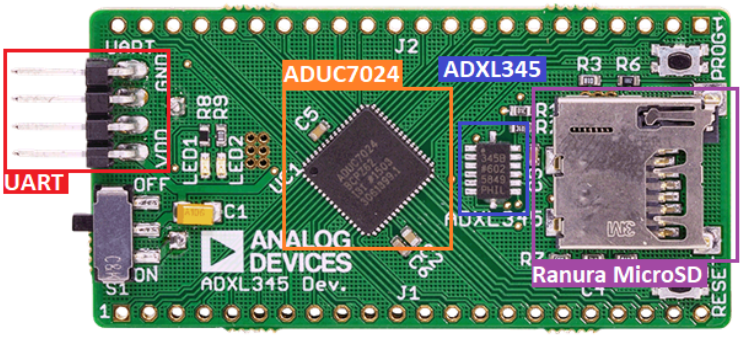

Figura 1.1: Placa EVAL-ADXL345Z-DB de Analog Devices

Si bien en el código que se proporciona, la información de aceleración de los tres ejes

se almacena en una tarjeta MicroSD, sería más interesante disponer de los valores en tiempo real. Además, si se quisieran hacer ampliaciones de *hardware*, estarían limitadas a las capacidades del microcontrolador incluido en el kit.

1.2. Objetivos

Es por lo presentado antes que, si bien la funcionalidad puede ser útil y suficiente en algunos casos, la opción de poder acceder a los valores en tiempo real sería más beneficiosa para el desarrollo de distintas aplicaciones y proyectos, a la par que podrían requerirse nuevas funcionalidades no pensadas previamente para ser ejecutadas con la EVAL-ADXL345Z-DB.

Como objetivos principales, se establecen:

- Desarrollar un sistema embebido y en consecuencia su aplicación *software* con la capacidad de comunicarse con el acelerómetro ADXL345 e imprimir las medidas de aceleración por pantalla en tiempo real.
- Añadir funcionalidades nuevas que permita el ADXL345, como su calibración de *offset* a nivel de registro o el manejo de sus interrupciones para detectar eventos diversos.

Como objetivo secundario, también sería interesante poder mantener la funcionalidad de almacenar los valores de aceleración en la tarjeta MicroSD.

1.3. Alternativas

Si bien desde antes de comenzar el desarrollo se sabía que se iba a usar un módulo FPGA con un sistema microprocesado embebido, la forma de realizar la comunicación con la EVAL-ADXL345Z-DB no estaba clara. En este momento surgieron dos opciones distintas:

La primera de ellas, conectar la FPGA con la placa de evaluación mediante una

UART para poder programar su microcontrolador. Con esta opción, se podría reutilizar el código inicial, manteniendo la funcionalidad de comunicación con el ADLX345 a la vez que se mantiene la funcionalidad con la MicroSD. Para devolver las medidas de aceleración a la FPGA e imprimirlas por pantalla, se podría usar tanto la UART que programa el microcontrolador como el bus SPI que guarda los valores en la MicroSD. El problema principal de esta opción es que, para programar el microcontrolador del acelerómetro, primero habría que programar la FPGA, complicando innecesariamente el sistema. Además, haría al sistema más lento ya que los datos deben ser mandados entre procesadores antes de ser imprimidos por pantalla.

La segunda opción, la finalmente adoptada, es la de conectar al ADXL345 directamente al sistema embebido de la FPGA mediante I^2C, haciendo uso de los pines externos de ambas placas. De la misma manera y mediante protocolo de comunicación serie SPI, se podrían comunicar la tarjeta MicroSD y la FPGA. Esta opción reduce la complejidad del sistema a la par que mantiene las funcionalidades y mejora el rendimiento.

2. Material

Para la consecución de los objetivos, serán necesarios distintos dispositivos y programas.

En la parte de *hardware*:

- Como se ha avanzado en el apartado anterior, usaremos la placa EVAL-ADXL345-DB de Analog Devices como acelerómetro.
- El módulo FPGA de Trenz Electronic TE0725-100-2I9 y el adaptador TE0790 para programarlo por JTAG.
- El analizador lógico Analog Discovery de Digilent para capturar las señales digitales y así comprobar y depurar el funcionamiento del hardware.

En la parte de *software*:

- Vivado para diseñar y desarrollar el sistema embebido.
- Vitis Classic para desarrollar el software de nuestro sistema.
- MobaXterm como terminal para la comunicación serie entre el sistema embebido y el PC.
- WaveForms, necesario para utilizar el analizador lógico.
- ARMWSDv1.8 para borrar la memoria del microcontrolador de la placa de evaluación.

2.1. Placa EVAL-ADXL345Z-DB

Como se explicó anteriormente, la placa está ideada para poder avanzar en el desarrollo de *firmware* y *hardware* simultáneamente, facilitando la creación de prototipos.

Las comunicaciones y el procesamiento se realizan mediante un microcontrolador ADuC7024, totalmente reprogramable mediante el cabezal UART de 4 pines y los botones de reinicio y descarga presentes en la placa.

Cuenta con dos conectores a los laterales que hacen externos todos los pines del microcontrolador. Esto facilita el diseño de expansiones en el *hardware*. Puede ser alimentada por dos baterías AAA o a 3.3V desde una fuente externa.

El ADuC7024 está conectado mediante I^2C al acelerómetro ADXL345. También se conecta mediante SPI a una ranura para tarjeta MicroSD, donde se pueden registrar los datos de aceleración.

2.1.1. El ADXL345

El ADXL345 [2] es un chip de bajo consumo desarrollado por Analog Devices para medir la aceleración tanto estática como dinámica. La corriente de alimentación varía, a 2.5V entre 23 µA y 140 µA y aumenta con la velocidad de adquisición de datos.

El sistema de medición tiene resolución de hasta 13 bits y rango seleccionable de ±2 g, ±4 g, ±8 g o ±16 g. La resolución es de 3.9 mg/LSB y puede detectar cambios en la inclinación menores a 1°.

El sensor está formado por una estructura micromecanizada de polisilicio construida sobre una oblea de silicio. Ésta está suspendida sobre la oblea por resortes también de polisilicio, que ofrecen resistencia a las fuerzas de aceleración. El desplazamiento de la estructura se detecta a través de condensadores diferenciales, produciendo una salida cuya amplitud es proporcional a la aceleración.

Los registros del acelerómetro son accesibles mediante los protocolos serie SPI (de 3 o 4 hilos) o I^2C para leer y escribir datos. El dispositivo actúa como esclavo. Mediante la lectura, podemos obtener cualquier información de los registros accesibles. Mediante la escritura, podemos configurar opciones como la resolución de los datos de salida, ancho de banda, rango, configuraciones de ahorro de energía, distintos

umbrales de detección de eventos, cambiar el modo de funcionamiento de la memoria FIFO, activar el modo reposo, modo ahorro de energía, auto test, etc. La tabla 2.1 muestra todos los registros accesibles.

(hex.)	**Nombre**	**Tipo**	**Descripción**
0x00	DEVID	R	ID del dispositivo
0x01 a 0x0C	Reserved	-	Reservado, no acceder
0x1D	THRESH_TAP	R/W	Umbral de toque
0x1E	OFSX	R/W	*Offset* del eje X
0x1F	OFSY	R/W	*Offset* del eje Y
0x20	OFSZ	R/W	*Offset* del eje Z
0x21	DUR	R/W	Duración de toque
0x22	Latent	R/W	Latencia de toque
0x23	Window	R/W	Ventana de toque
0x24	THRESH_ACT	R/W	Umbral de actividad
0x25	THRESH_INACT	R/W	Umbral de inactividad
0x26	TIME_INACT	R/W	Tiempo de inactividad
0x27	ACT_INACT_CTL	R/W	Control de detección de actividad e inactividad
0x28	THRESH_FF	R/W	Umbral de caída libre
0x29	TIME_FF	R/W	Tiempo de caída libre
0x2A	TAP_AXES	R/W	Control en ejes para toque doble o simple
0x2B	ACT_TAP_STATUS	R	Fuente del toque único o doble
0x2C	BW_RATE	R/W	Control de tasa de muestreo y modo de potencia
0x2D	POWER_CTL	R/W	Control de ahorro de energía
0x2E	INT_ENABLE	R/W	Control de habilitación de interrupciones
0x2F	INT_MAP	R/W	Control de mapeo de interrupciones
0x30	INT_SOURCE	R	Fuente de las interrupciones
0x31	DATA_FORMAT	R/W	Control de formato de datos
0x32	DATAX0	R	Datos del eje X - 0
0x33	DATAX1	R	Datos del eje X - 1
0x34	DATAY0	R	Datos del eje Y - 0
0x35	DATAY1	R	Datos del eje Y - 1
0x36	DATAZ0	R	Datos del eje Z - 0
0x37	DATAZ1	R	Datos del eje Z - 1
0x38	FIFO_CTL	R/W	Control de la memoria FIFO
0x39	FIFO_STATUS	R	Estado de la memoria FIFO

Tabla 2.1: Mapa de registros del ADXL345

Además, incluye funciones especiales de detección. Detecta actividad e inactividad comparando la aceleración en cualquier eje con umbrales previamente definidos. También puede detectar toques simples y dobles en cualquier dirección y caídas libres. Estas funciones se pueden asignar individualmente a cualquiera de los dos pines de interrupción del dispositivo. En nuestro caso no los podremos usar ya que están conectados al microcontrolador y no son accesibles mediante los pines externos de la placa. Igualmente, se puede acceder a la información de la generación

de interrupciones mediante registro.

2.1.2. Comunicación serie I²C

I²C (Inter-Integrated Circuit) [3] es un tipo de comunicación serie síncrona usada para conectar dispositivos a corta distancia. La peculiaridad principal es su simplicidad, al poder controlar la red solamente con dos pines *inout*. Las velocidades de transmisión más comunes soportadas son la estándar de 100kbit/s y la rápida de 400kbit/s, aunque existan otras. Las líneas deben de estar conectadas mediante una resistencia pull-up de usualmente 10kΩ a 3,3 o 5V.

Otra característica del tipo de transmisión es la posibilidad de conectar más de un maestro o esclavo en el bus de datos, ya que cada dispositivo tiene una dirección asignada por la que es identificado. La estándar es de 7 bits, aunque existen direcciones de 10 bits para una mayor cantidad de dispositivos conectados a la línea.

Las dos líneas son:

- SCL (Serial Clock Line): transmite la señal de reloj que sincroniza la transferencia de datos entre el maestro y el esclavo. La señal es generada por el maestro.
- SDA (Serial Data Line): transporta los datos bidireccionales entre el maestro y el esclavo.

Una comunicación estándar con direcciones de 7 bits, seguiría la siguiente secuencia:

1. Ambas líneas comenzarían estando en reposo, es decir, en valor lógico alto. El controlador que desea comunicar datos genera la condición de inicio poniendo SDA = '0' mientras mantiene SCL = '1'.
2. A continuación, SCL pasa también a nivel lógico bajo. A partir de este momento y hasta el final de la transmisión, SCL se encargará de marcar la velocidad de transmisión de datos.

3. El maestro envía mediante SDA la dirección de 7 bits del esclavo con el que se quiere comunicar, seguida de un bit que indica si la operación será de lectura '1' o de escritura '0'.

4. Si el dispositivo con la dirección objetivo se encuentra en el bus, responderá con el bit de reconocimiento ACK poniendo SDA a '0' durante el siguiente pulso de reloj. Posteriormente, el esclavo transmitirá un byte y el maestro responderá con ACK. Se seguirán enviando bytes respondidos por ACK hasta que el maestro realice la condición de parada. Si por el contrario el dispositivo no se encuentra en la línea, SDA se mantendrá en '1' el siguiente pulso de reloj, y el maestro procederá a realizar la condición de parada.

5. La condición de parada se da cuando SDA pasa a '1' mientras SCL = '1'. A partir de ese momento, la línea se encontraría en reposo y otro dispositivo podría reclamarla para comunicar datos.

La figura 2.3 muestra un ejemplo.

En nuestra aplicación, las características relevantes para el funcionamiento y configuración de los dispositivos son:

- El ADXL345 actúa como esclavo.
- Velocidad de transmisión de 100kbit/s.
- Voltaje de la línea 3.3V.
- Direcciones de 7 bits, siendo la del ADXL345 0x53.

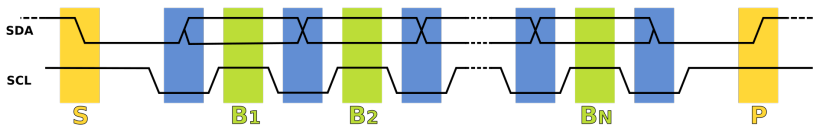

Figura 2.1: Secuencia de transferencia de datos en I^2C.

Para la comunicación con el ADXL345, además de la dirección del dispositivo, es necesario a continuación enviar una dirección del registro específica. Si la operación

es de escritura, después, se enviarán los bytes que se desean escribir. Si el número es mayor a uno, los siguientes bytes serán escritos en los registros sucesivos.

Para la lectura, será necesario primero realizar una operación de escritura mandando la dirección de registro a leer y posteriormente cortar la transmisión. A continuación, se vuelve a enviar la dirección del dispositivo, pero en modo lectura, leyendo tantos bytes como se desee. Si se lee más de un byte, se transmitirán de los registros sucesivos. Una descripción más detallada del protocolo en el ADXL345 se ilustra en la figura 2.2.

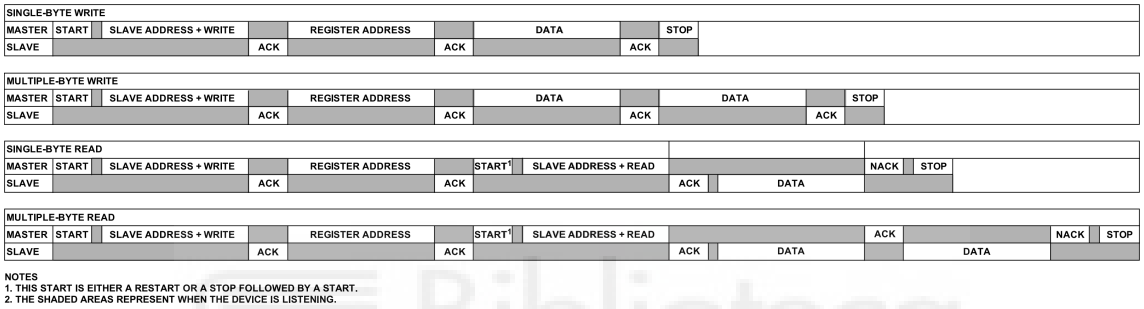

Figura 2.2: Ejemplo de comunicación I^2C en el ADXL345.

2.1.3. Ranura para tarjeta MicroSD y comunicación SPI

La ranura para MicroSD es una opción bastante útil para poder guardar los valores de aceleración que se van leyendo del acelerómetro. En concreto, en la EVAL-ADXL345Z-DB, el tipo de conexión es la necesaria para comunicar la MicroSD mediante protocolo SPI, una de las opciones de comunicación para tarjetas SD. En resumen, la tarjeta MicroSD trabaja como esclavo mientras que el controlador, en este caso el ADuC7024, trabaja como maestro.

Hay una serie de comandos que se deben mandar con el fin de establecer comunicación adecuadamente, como se puede consultar en el manual '*Physical Layer Simplified Specification*' de la SD Association [4]. No es el objeto del proyecto entender las distintas secuencias de comandos, pero sí se explicará el protocolo SPI.

El protocolo SPI (*Serial Peripheral Interface*) [5] es un protocolo de comunicación serie asíncrono usado para comunicación entre dispositivos a corta distancia, sobre

todo entre microcontroladores en sistemas embebidos y periféricos como sensores o memorias. Las características principales son:

- Comunicación '*Full-Duplex*', es decir, se pueden transmitir y recibe datos a la vez.
- Comunicación maestro-esclavo, pudiendo haber más de un esclavo en el bus, pero solo un maestro.
- Transmisión de datos síncrona.
- Alta velocidad de transmisión, del orden de MHz.

Existen varias configuraciones para la transmisión de datos, pero la que usa la MicroSD, la básica, está formada por cuatro señales lógicas:

- CS (*Chip Select*): línea usada por el maestro para seleccionar el esclavo con el que establecer la comunicación. Es activa a nivel bajo y hay una por cada esclavo en el bus.
- SCK (*Serial Clock*): señal de reloj generada por el maestro para sincronizar la transmisión.
- MOSI (*Master Out Slave In*): transmite datos del maestro al esclavo.
- MISO (*Master In Slave Out*): transmite datos del esclavo al maestro.

La comunicación SPI sigue la siguiente secuencia:

1. El maestro elige el esclavo bajando su línea CS a '0' lógico.
2. Se transmiten los datos simultáneamente bit por bit, empezando por el más significativo, normalmente en grupos de uno o más bytes. El maestro lo hace por la línea MOSI y el esclavo por MISO.
3. Una vez terminada la transmisión el maestro deselecciona al esclavo devolviendo CS a '1'.

La figura 2.3 muestra un ejemplo de comunicación SPI entre el ADuC7024 y la tarjeta MicroSD.

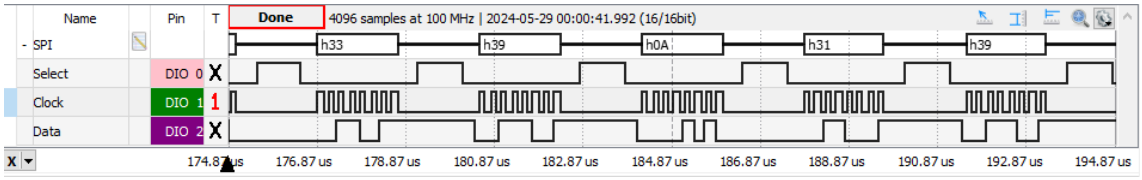

Figura 2.3: Transmisión de datos en modo SPI por la linea MOSI entre el ADuC7024 y la tarjeta MicroSD.

2.1.4. ADuC7024 y programación mediante UART

El ADuC7024 es un microcontrolador desarrollado por Analog Devices [6] usado en aplicaciones de control industrial, automatización e instrumentación. Dispone de convertidores analógico-digital y digital-analógico, interfaces de comunicación I2C, SPI y UART, puertos GPIO, generadores PWM, entre otros. Además, cuenta con una memoria Flash/EE de 62kB y 8kB de SRAM.

Como se ha adelantado anteriormente, no se usará el dispositivo.

La forma de desprogramar el microcontrolador desde el ordenador es muy sencilla, basta con descargar el software ARMWSDv1.8 de la web de Analog Devices y seguir las instrucciones que aparecen en el manual [7]. Será necesario hacerlo, ya que, por defecto, está instalado el *software* de ejemplo. Esto evita que el microcontrolador pueda reclamar la línea de I2C y ralentizar nuestro programa. Pero, sobre todo, para evitar que existan colisiones a la hora de usar el bus SPI para almacenar datos en la MicroSD, ya que el protocolo solo admite un maestro.

Como indica el manual para programar el ADuc7024, si iniciamos el software podremos usar la opción '*Mass erase*' para borrar la memoria (Figura 2.4). La secuencia sería la siguiente:

1. Insertar 2 baterías AAA en la placa.
2. Conectar a un cable de UART a RS-232.

3. Conectar al ordenador directamente o mediante un cable de RS-232 a USB.

4. Seleccionar en el software el puerto COM asignado a la placa y cambiar la velocidad de transmisión a 115200 Bd con 8 bits de datos, 1 de parada y ninguno de paridad.

5. Seleccionar como componente el ADuC7024.

6. Pulsar el botón de ejecutar.

7. Presionar los botones '*Program*' y '*Reset*' presentes en el hardware.

8. Ya podemos cerrar el programa y apagar la placa.

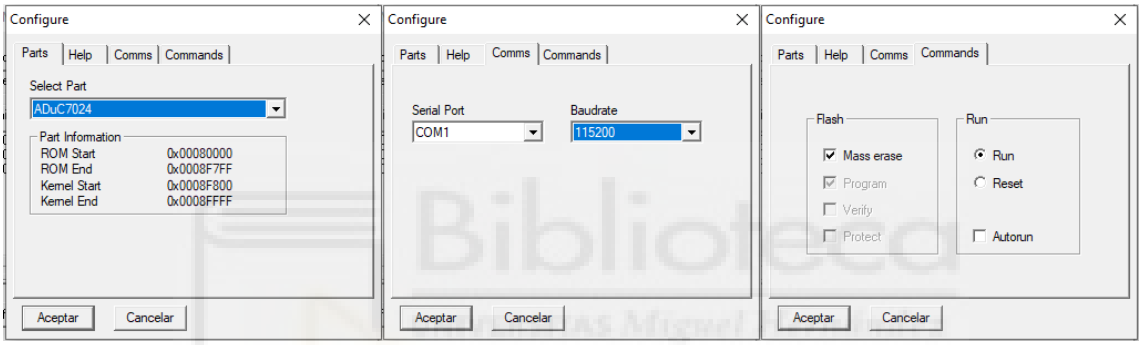

Figura 2.4: Configuración en el *software* ARMWSDv1.8 para borrar la memoria del ADuC7024.

2.2. FPGA y adaptador JTAG

Como plataforma de desarrollo, se usará la FPGA TE0725 y el adaptador específico TE0790 para poder programarla mediante JTAG.

2.2.1. TE0725

El TE0725-03-100-2I9 de Trenz Electronic [8] es un módulo FPGA con un tamaño reducido que integra un dispositivo Artix-7 A100T de Xilinx (Figura 2.5). Las características principales serían:

- Reloj de Sistema de 100MHz.

- EEPROM de 128-Kbits.
- Memoria Flash de 32MBytes.
- 2 cabezales de 50 pines con distancia de separación de 2,54mm.
- Conector JTAG/UART.
- Memoria externa HyperRAM de 8Mbytes de Cypress.

Figura 2.5: Planta de la placa TE0725-03-100-2I9.

Esto lo hace una perfecta elección para nuestro proyecto, ya que requeriremos una memoria externa como la HyperRAM para mejorar la eficiencia del sistema liberando memoria interna del procesador embebido que vamos a configurar. Además, dispondremos de una capacidad de almacenamiento y un ancho de banda aumentados. Al ser una placa pequeña, la hace muy manejable para aplicaciones portátiles.

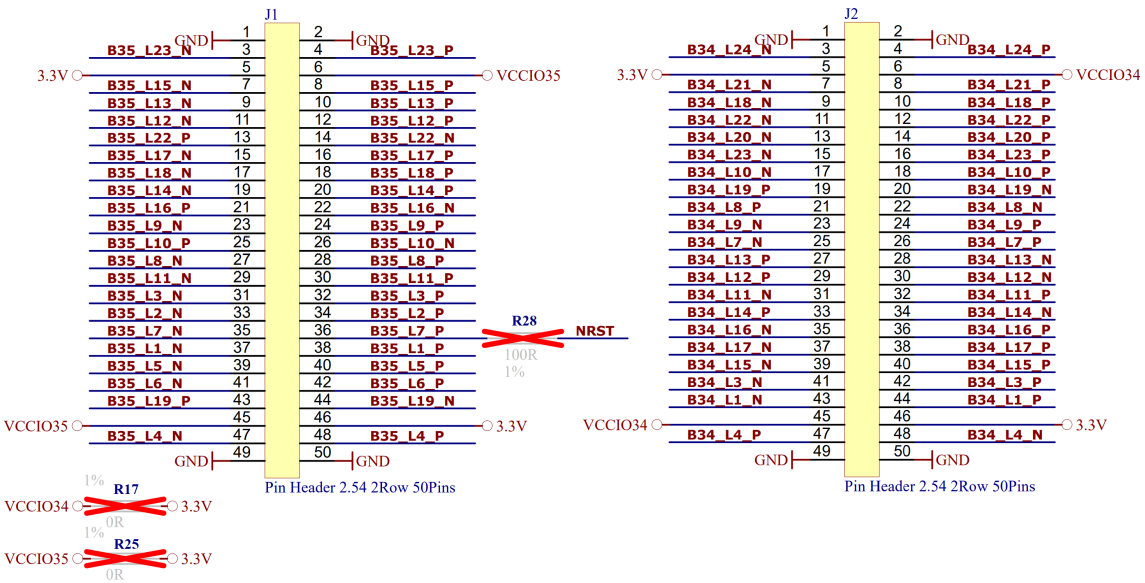

Figura 2.6: Pines de la placa TE0725-03-100-2I9

Uno de los problemas que alargó mucho el tiempo de desarrollo fue una peculiaridad que solo afecta a la versión 2I9 de todas los módulos TE0725. Como se puede apreciar en la figura 2.6 extraída del esquemático [9], las resistencias R17 y R25 de 0 Ω que conectan la alimentación a las filas de pines del módulo FPGA no se encuentran soldadas. La finalidad de esto es poder alimentar los pines mediante una fuente externa, ya que la FPGA tiene una limitación de 100mA. Esto causaba que la salida de los pines configurados fuese siempre 0V. Para no tener que soldar las resistencias se decidió unir la alimentación de los pines de la fila J1 VCCIO35 y de la fila J2 VCCIO34 con un cable.

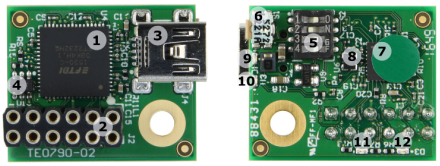

Figura 2.7: Vistas frontal y trasera del adaptador TE0790.

Aún con estos cambios, sin embargo, la placa no puede ser utilizada por sí sola de la forma en la que queremos trabajar. Necesitaremos otro componente, el TE0790, un adaptador que se puede conectar en todas las FPGA TE0725 para convertir el conector JTAG/UART en un conector con el que poder programar la placa desde el ordenador.

2.2.2. TE0790

El TE0790 de Trenz Electronic [10] es un adaptador universal de USB2.0 a JTAG, UART y GPIO (Figura 2.7). La memoria EEPROM de configuración de la placa está programada de fábrica con un *firmware* que garantiza su compatibilidad con las herramientas de Xilinx.

El adaptador puede ser alimentado vía USB o a 3.3V mediante sus pines. La configuración se elige mediante los cuatro interruptores de S2. Como se puede observar en el diagrama de bloques (Figura 2.8) del componente, para poder utilizar el modo JTAG y alimentar el dispositivo desde el ordenador, se requiere cerrar S2-1, S2-3,

S2-4 y dejar abierto S2-2.

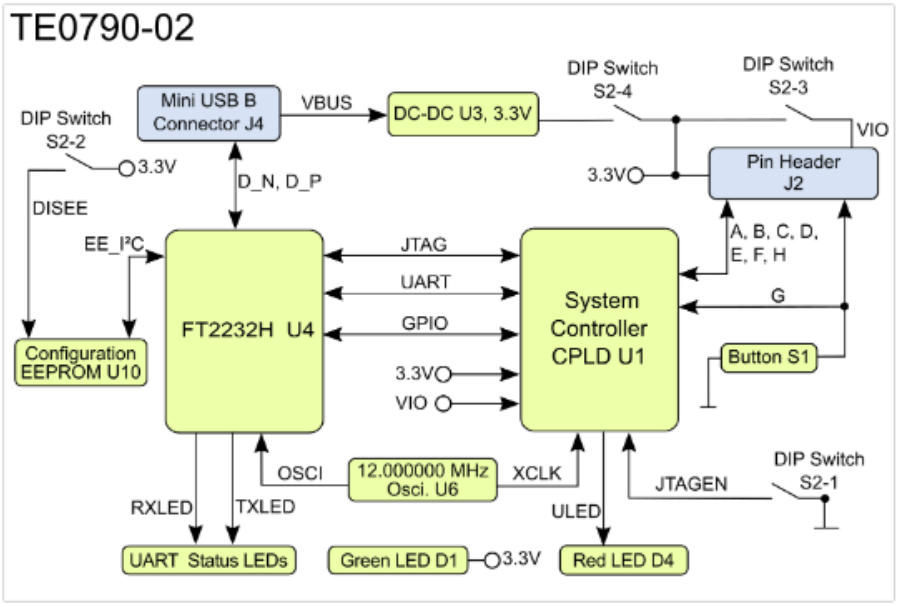

Figura 2.8: Diagrama de bloques del adaptador TE0790.

3. Sistema MicroBlaze

El MicroBlaze [11] es un *soft processor* de 32/64 bits diseñado por Xilinx pensado para ser integrado fácilmente en sus FPGA y SoC. Es personalizable y simplifica el proceso de desarrollo de un sistema embebido microprocesado, ya que pueden ser configurados muchos aspectos como la interfaz, el tamaño de los cachés de datos e instrucciones o el número de periféricos, entre otros.

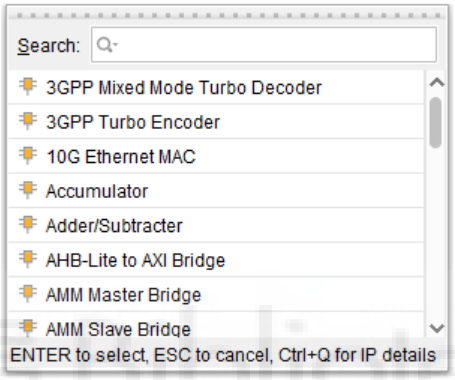

Figura 3.1: Catálogo de IPs de Vivado.

El hecho de que tenga un buen rendimiento y se pueda integrar bien con otros componentes de Xilinx lo hace una buena elección. Su uso pone a nuestra disposición un catálogo de IPs "gratuitas" (Figura 3.1) con el que se agiliza el tiempo de tiempo de desarrollo y se reduce la complejidad de la tarea.

Los campos en los que se usa el MicroBlaze son muy variados, y van desde industria, medicina, automoción, consumo o comunicaciones.

Puede ser configurado de tres formas:

- Microcontrolador: para ejecutar código *bare-metal* , es decir, aplicaciones que no requieren sistema operativo.
- Procesador en tiempo real: asegura un procesamiento determinístico en sistemas operativos de tiempo real.

- Procesador de aplicaciones: pensado para ejecutar aplicaciones complejas que incluyan sistemas operativos embebidos como Linux.

3.1. Vivado y el IP integrator

Vivado es un *software* desarrollado por Xilinx que abarca todos los aspectos del desarrollo de *hardware* en FPGAs, incluyendo diseño, simulación, síntesis, implementación y programación [12].

Para realizar el sistema embebido MicroBlaze, se sigue el tutorial del GitHub de Xilinx diseñado para un módulo FPGA Spartan-7 [13]. Aunque nuestro dispositivo sea un Artix-7, al ser de la misma familia el proceso es muy similar. Sin embargo, no lo seguimos al pie de la letra, ya que nuestra placa cuenta con una memoria HyperRAM en lugar de una DDR3, pero es útil para comprender los pasos necesarios a la hora de crear un proyecto basado en MicroBlaze.

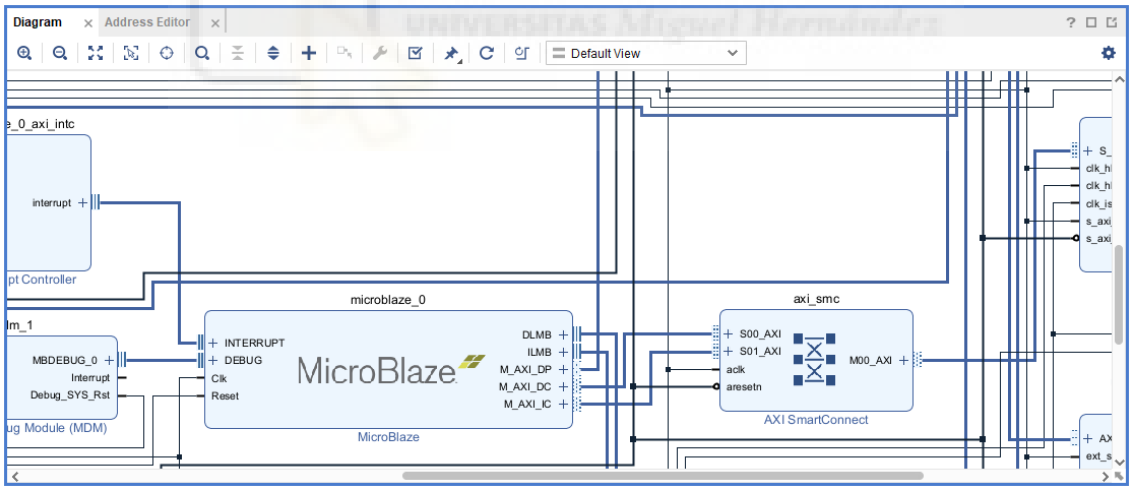

Figura 3.2: Ejemplo del area de trabajo del IP Integrator.

La herramienta principal usada es el IP integrator (Figura 3.2), disponible dentro del *software* de Vivado. Nos permite diseñar sistemas mediante una interfaz gráfica conectando bloques. Cada bloque representa una IP (Propiedad Intelectual), es decir, un componente pre-diseñado y verificado con una función específica. Nuestro diseño de sistema se construye mediante la interconexión de varios bloques IP.

El diagrama de bloques está basado en los dos tutoriales proporcionados por Trenz Electronic para el módulo TE0725 [14]. Primero se implementa la *'Test Board'* y a continuación la memoria externa 'HyperRAM'.

3.2. Diagrama de Bloques

El primer paso es crear el proyecto y seleccionar la placa que vamos a usar en nuestro diseño (Figura 3.3). El archivo de placa se puede descargar del GitHub de Xilinx [15] y describe las características físicas y eléctricas.

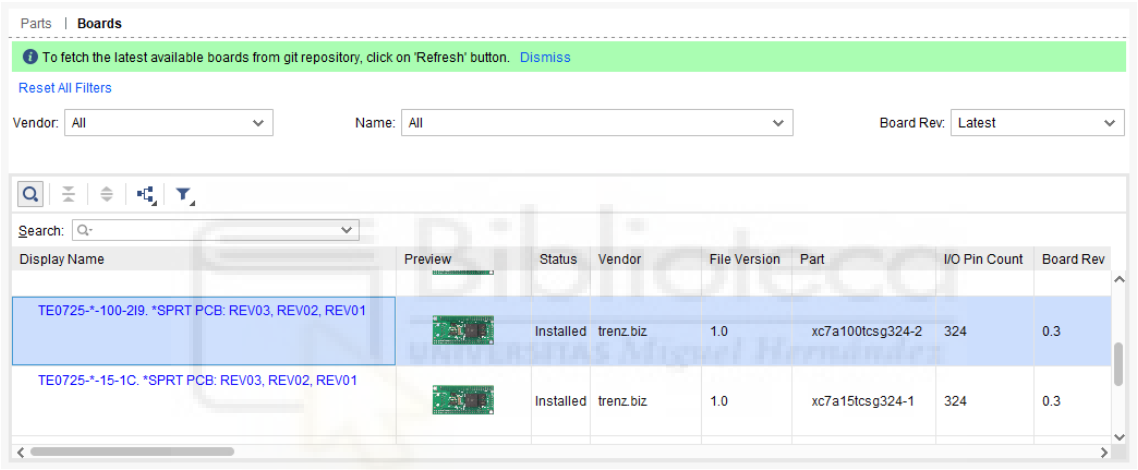

Figura 3.3: Menú de selección de placa FPGA.

El proceso para crear el diagrama de bloques de nuestro sistema embebido básico es bastante sencillo, simplemente creamos un diagrama de bloques en el menú '*IP INTEGRATOR*' y copiamos los periféricos a añadir del tutorial '*Test Board*'. Además, aunque no siempre funciona como puede desear el usuario, Vivado ofrece la opción de configurar los bloques basándose en el archivo de placa que estamos usando y realizar las conexiones entre los bloques de las diferentes IPs (Figura 3.4).

El diagrama básico está formado por el MicroBlaze en modo bare-metal y algunos controladores de periféricos de la placa. Se añaden:

- Un '*Cloking Wizard*', una IP que permite generar señales de reloj con distinta frecuencia y fase a las generadas por el cristal de cuarzo externo. Por ahora se

mantendrá la frecuencia de referencia de 100MHz en su salida.

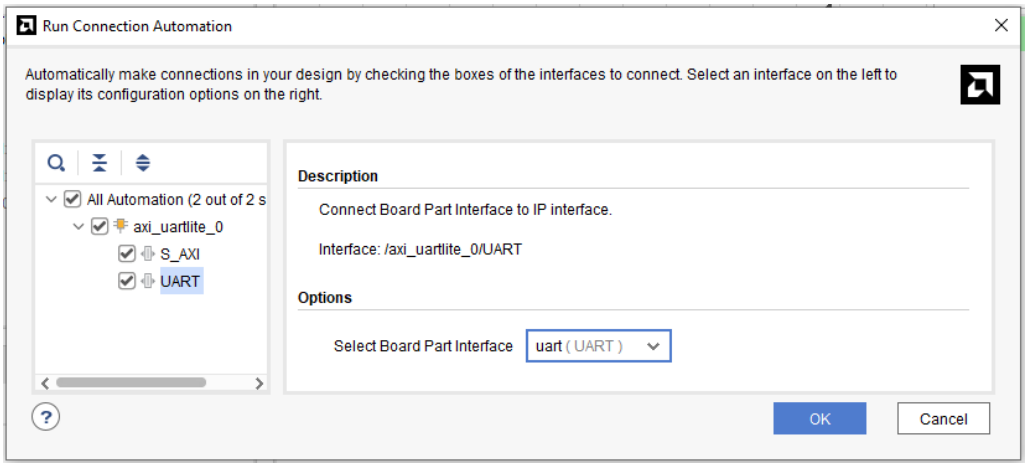

Figura 3.4: Ejemplo de conexión automatizada.

- El bloque '*Processor System Reset*', que se encarga de gestionar las señales de reinicio en el sistema MicroBlaze. Sincroniza las señales de reinicio con el reloj del sistema para evitar estados de metaestabilidad y coordina el orden de reinicio de diferentes componentes. Además filtra pulsos cortos para evitar señales de reinicio no deseadas.

- El '*Microblaze Debug Module*', que ayuda a la depuración del *software* ejecutado en el MicroBlaze. Es imprescindible a la hora de desarrollar la aplicación, ya que permite usar el JTAG para detener la ejecución del código en momentos específicos y monitorear el estado del procesador.

- Tanto '*AXI SmartConnect*' como '*AXI Interconnect*'. Son componentes que facilitan la conexión de múltiples maestros y esclavos que usen el protocolo AXI (*Advanced eXtensible Interface*) [16], desarrollado por ARM. Es el protocolo de comunicación por excelencia adoptado por Xilinx y muchos de sus socios.

- '*AXI Interrupt Controller*' para gestionar las interrupciones de los periféricos del sistema. Agrupa todas las señales de interrupción y les asigna un identificador específico. Usa la interfaz AXI para comunicarse con el MicroBlaze.

- La IP '*AXI Uartlite*'. Es un periférico para la transmisión de datos mediante UART. Estará conectado directamente al conector JTAG/UART TE0790 y se usará para imprimir datos por pantalla. Se configura a 9600 Bd, con 8 bits de

datos, sin paridad y con un bit de parada (Figura 3.5).

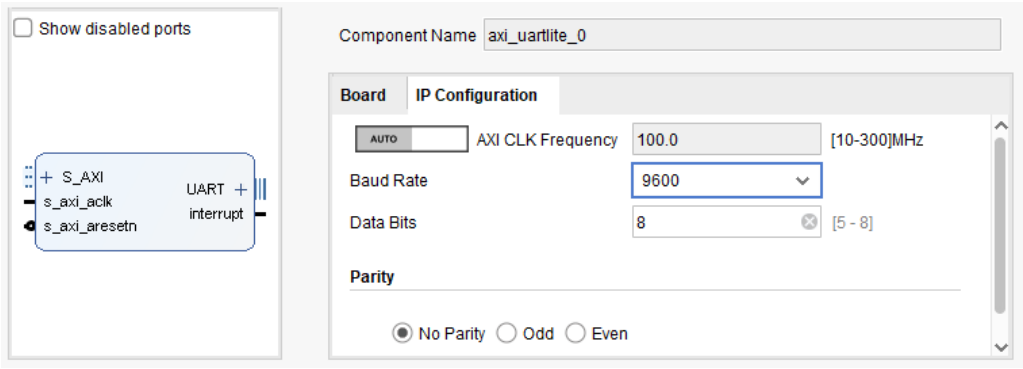

Figura 3.5: Menú de configuración de la IP 'AXI Uartlite' del conector JTAG/UART.

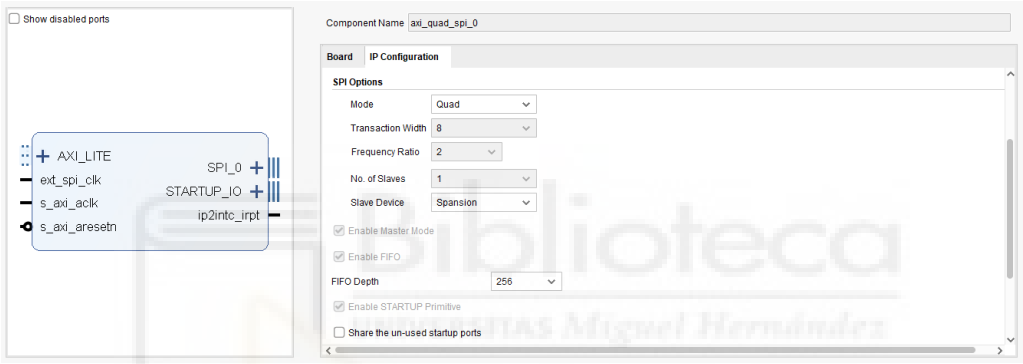

Figura 3.6: Menú de configuración de la IP 'AXI Quad SPI' de la memoria Flash.

- El bloque '*AXI Quad SPI*'. Permite la comunicación en protocolo SPI de uno, dos o cuatro hilos. Se conecta a la memoria Flash SPI (Figura 3.6).

- La IP '*AXI Timer*' sirve como temporizador. Se selecciona el modo doble contador para poder crear más adelante un contador de 64 bits uniendo los dos de 32 bits en modo cascada (Figura 3.7).

- El bloque '*AXI IIC*' para comunicación serie I^2C. Es el periférico encargado de comunicarse con la memoria EEPROM externa, aunque no se usará.

Después se pasa al tutorial 'HyperRAM'. El diagrama de bloques es el mismo. Simplemente se añade y conecta el controlador para la memoria HyperRAM externa de 8MBytes.

La HyperRAM [17] es un tipo de memoria basada en la comunicación HyperBus, una tecnología que combina propiedades de la comunicación serie y la comunicación en paralelo. Las principales ventajas de la HyperRAM son su bajo consumo de energía y la simplicidad del diseño, ya que utiliza pocos pines.

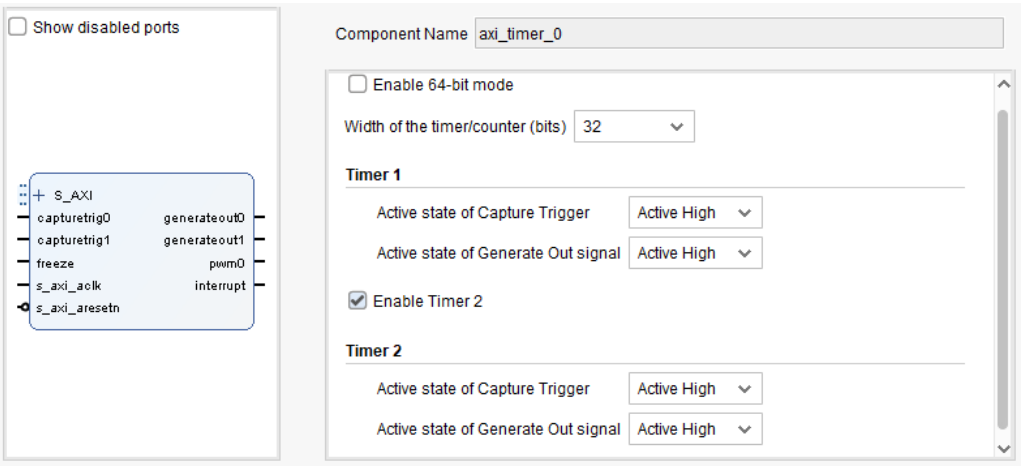

Figura 3.7: Menú de configuración de la IP 'AXI Timer'.

En el tutorial, añaden un controlador de HyperBus de prueba con un tiempo de funcionamiento máximo de 10 minutos. Aunque se intentó conseguir la IP de prueba contactando con el servicio de atención al cliente de Synaptic Laboratories Ltd, se encontró un controlador de código abierto en Github [18] que funcionó perfectamente en el diseño (Figura 3.8).

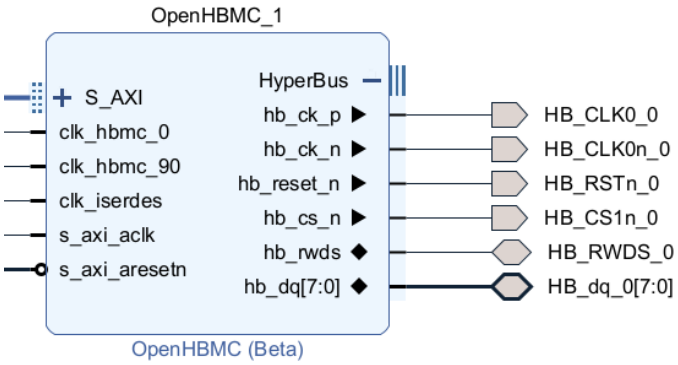

Figura 3.8: Controlador de HyperBus de código abierto OpenHBMC.

Este bloque requiere generar y conectar dos nuevas señales de reloj desde el '*Clo-*

king Wizard'. Una de 100MHz desfasada respecto a la señal original 90° y otra de 300MHz sin desfase (Figura 3.9).

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)	
		Requested	Actual	Requested	Actual	Requested	Actual
☑ clk_out1	clk_out1	100.000	100.00000	0.000	0.000	50.000	50.0
☑ clk_out2	clk_out2	100.000	100.00000	90.000	90.000	50.000	50.0
☑ clk_out3	clk_out3	300	300.00000	0.000	0.000	50.000	50.0

Figura 3.9: Relojes de salida de la IP 'Cloking Wizard'.

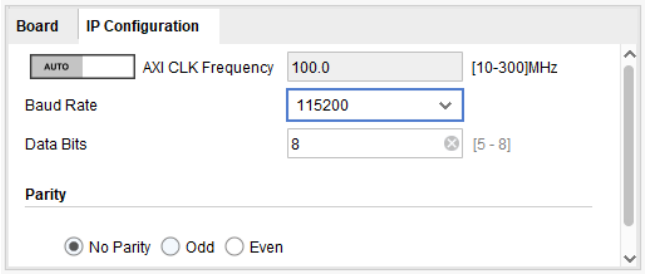

Figura 3.10: Menú de configuración de la IP 'AXI Uartlite' para el ADuC7024.

Una vez terminado el tutorial de HypeRAM, pasamos a añadir los periféricos que faltan para nuestra aplicación concreta:

- Un nuevo periférico 'AXI UartLite' para la comunicación con el microcontrolador de la placa de evaluación del ADXL345 (Figura 3.10). Finalmente no se usará.
- Otro 'AXI IIC' para la comunicación con el ADXL345 (Figura 3.11).
- Otro 'AXI Quad SPI' para la comunicación con la tarjeta MicroSD.

Cuando ya hemos incluido todos los bloques que vamos a necesitar, hacemos externos los pines de las diferentes IPs para poder asignarlos a los pines físicos de la placa. Algunos pines como los de reloj de sistema, reinicio, I^2C de la memoria EEPROM, SPI de la memoria Flash o la UART del cable JTAG/UART se pueden asignar automáticamente (Figura 3.12), lo que nos ahorrará la necesidad de asignarlos después manualmente.

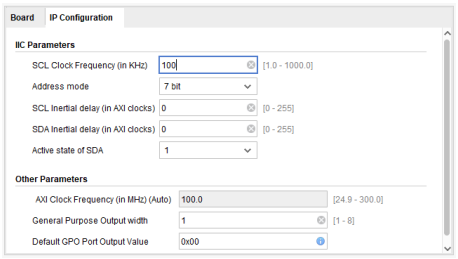

Figura 3.11: Menú de configuración de la IP 'AXI IIC' para la comunicación con el ADXL345.

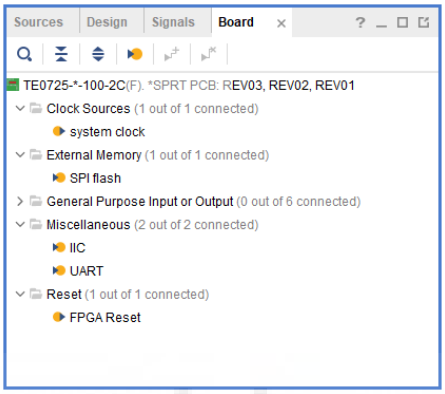

Figura 3.12: Asignación automática de pines en la placa TE0725.

Además, en la pestaña '*Address editor*', se aumenta el caché de datos y el de instrucciones del MicroBlaze a 128KB para mejorar el rendimiento general del sistema y evitar que se acceda tantas veces a la memoria principal (Figura 3.13).

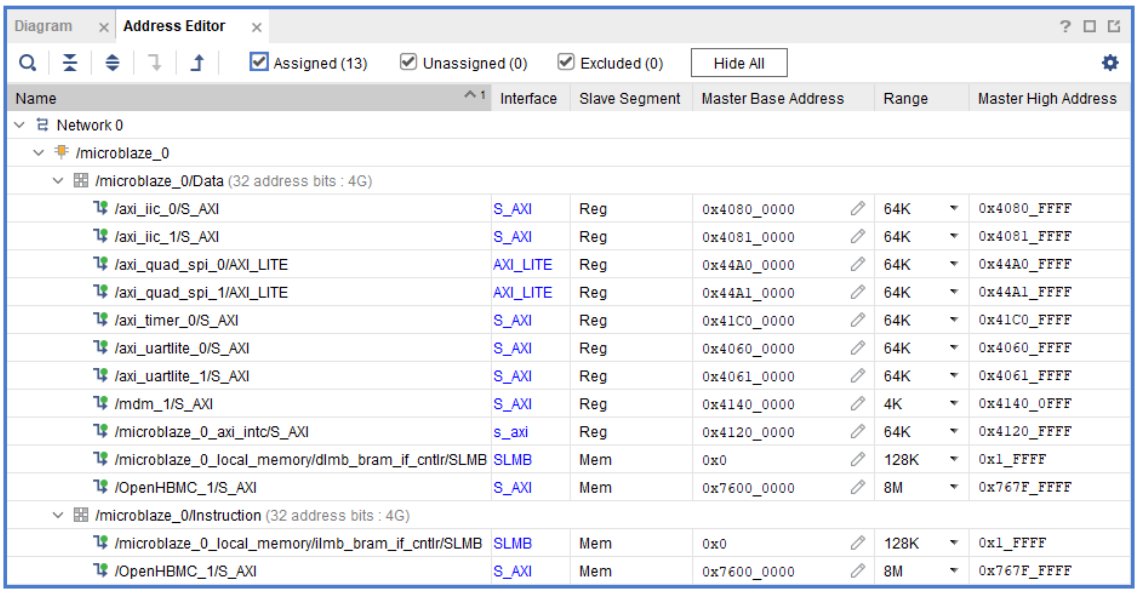

Figura 3.13: Ventana '*Address editor*' del diagrama de bloques.

Se pasa a validar el diseño y verificar que no hay errores. El diagrama de bloques final se muestra en el anexo B.

A continuación se generan los *Output Products* (Figura 3.14). Éstos incluyen las listas de interconexiones entre componentes, archivos de simulación y archivos necesarios para el proceso de síntesis e implementación.

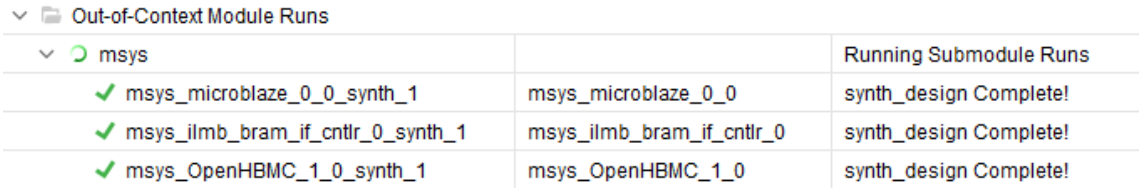

Figura 3.14: Proceso de generación de los *Output Products*.

Después, dejamos a Vivado crear el *HDL Wrapper*. Es el archivo que conectará las entradas y salidas del diseño con los pines físicos descritos en el archivo de *constraints*.

Ahora, se crean los archivos de *constraints*. Un archivo de *constraints* especifica las condiciones y limitaciones del diseño, como la asignación de pines o las restricciones de temporización. El tutorial de la TE0725 especifica dos. Éstos son _i_bitgen _common.xdc, que se encarga de asegurar que el bitstream generado sea compatible con las especificaciones eléctricas, y _i_hyperram.xdc, que se encarga de seleccionar las opciones de configuración de los pines que comunican con la memoria HyperRAM externa.

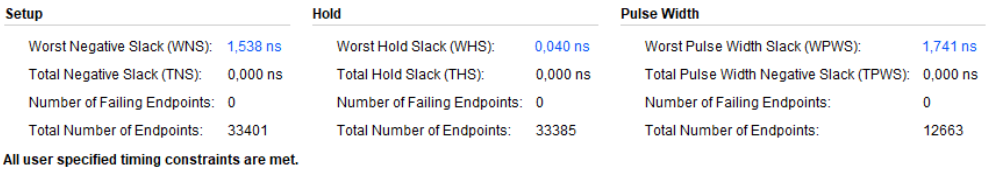

Figura 3.15: Verificación del cumplimiento de las restricciones temporales del sistema MicroBlaze.

Además, creamos un archivo de *constraints* propias llamado 'accel.xdc'. En él se

asignan los pines que queremos usar para la UART que comunica con el microcontrolador de la placa de evaluación, los del periférico SPI a la MicroSD y los de I^2C para comunicación serie con el acelerómetro. Todas los archivos de *constraints* se pueden consultar en el anexo A.

Se genera el *bitstream*, el archivo binario que contiene la configuración específica del diseño FPGA para programarla basándose en el hardware.

Tras verificar que se ha podido generar correctamente y que se cumplen las restricciones temporales (Figura 3.15), podemos ver también el consumo de energía del sistema y el porcentaje de utilización de recursos de la FPGA (Figura 3.16).

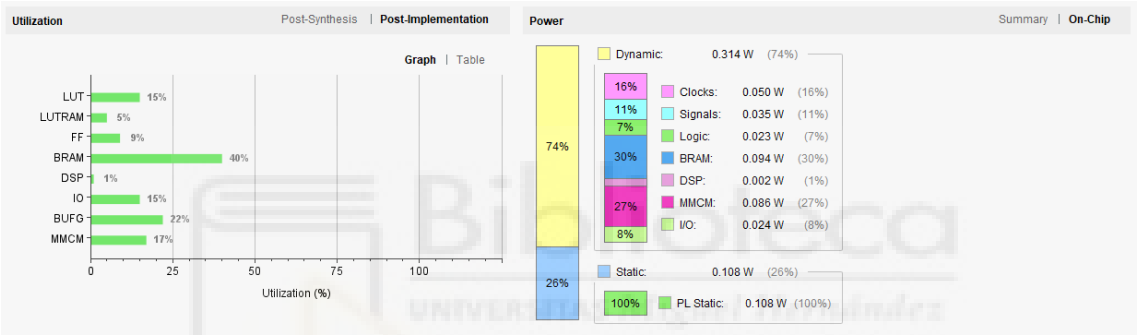

Figura 3.16: Utilización de recursos lógicos y consumo de energía del sistema MicroBlaze.

Por último, se exporta el *hardware* incluyendo el *bitstream* para poder desarrollar la aplicación en Vitis. Esto creará un archivo '.xsa' que contiene la información sobre la configuración de *hardware* que se va a utilizar en el desarrollo de *software*.

3.3. Conexiones

Las conexiones entre FPGA y acelerómetro se realizan mediante cables y siguen la asignación de la tabla 3.1. La función de cada pin se puede consultar en el esquemático de la placa EVAL-ADXL345Z-DB [19].

TE0725	**EVAL-ADXL345-DB**	**Descripción**
J1 P1	UART 1	GND
J1 P3	UART 3	TX UART ADuC7024
J1 P4	UART 2	RX UART ADuC7024
J1 P5	UART 4	VDD
J2 P21	J2 P6	SPI MicroSD MOSI
J2 P22	J2 P5	SPI MicroSD CS
J2 P27	J2 P10	SPI MicroSD SCK
J2 P28	J2 P9	SPI MicroSD MISO
J2 P43	J2 P12	I2C ADXL345 SCL
J2 P44	J2 P11	I2C ADXL345 SDA

Tabla 3.1: Conexiones entre el módulo TE0725 y la placa EVAL-ADXL345Z-DB.

4. Aplicación *software*

Para crear la aplicación se utiliza el *software* 'Vitis Classic' [20]. Vitis es una plataforma de software desarrollada por Xilinx usada para desarrollar aplicaciones destinadas a ejecutarse en sistemas embebidos. Nos permite poder trabajar con nuestra FPGA a un nivel de abstracción más alto mediante el uso de C o C++. Además, incluye una gran cantidad de bibliotecas para poder trabajar con los bloques IP de Xilinx incluidos en el diseño.

Se decidió usar esta versión para programar el sistema MicroBlaze frente a otra más moderna puesto que era la herramienta explicada en el tutorial. Además, existen más referencias en la web para resolver dudas que puedan surgir durante el desarrollo.

Para crear un nuevo proyecto, se selecciona en el menú '*New Platform Project*' y se indica la ruta hasta el archivo '.xsa' generado anteriormente.

4.1. Pruebas de *hardware*

La primera acción que se realiza antes de comenzar a desarrollar la aplicación *software* para comunicarnos con el ADXL345 es crear dos aplicaciones independientes:

- La primera se encarga de verificar que el controlador de HyperBus que estamos usando funciona y se detecta la memoria HyperRAM.
- La segunda se encarga de verificar que todos los periféricos desarrollados por Xilinx usados en nuestro diseño funcionan correctamente. Realiza una auto prueba y verifica que cada periférico puede generar una interrupción en el subsistema de interrupciones.

Ambas aplicaciones son generadas automáticamente por Vitis basándose en el hardware especificado en el archivo '.xsa', por lo que no es necesario modificar nada, simplemente añadir las plantillas. Se llaman *'Memory Tests'* y *'Peripheral Tests'* respectivamente.

Es importante cambiar la memoria que se va a utilizar para almacenar el código y las variables del programa. Para ello se elimina la selección por defecto que usa la memoria del MicroBlaze y se pasa a usar la memoria HyperRAM. Este procedimiento se realizará en todas las aplicaciones sucesivas que creemos. La modificación se realiza en el menú '*Generate Linker Script*' de la aplicación (Figura 4.1).

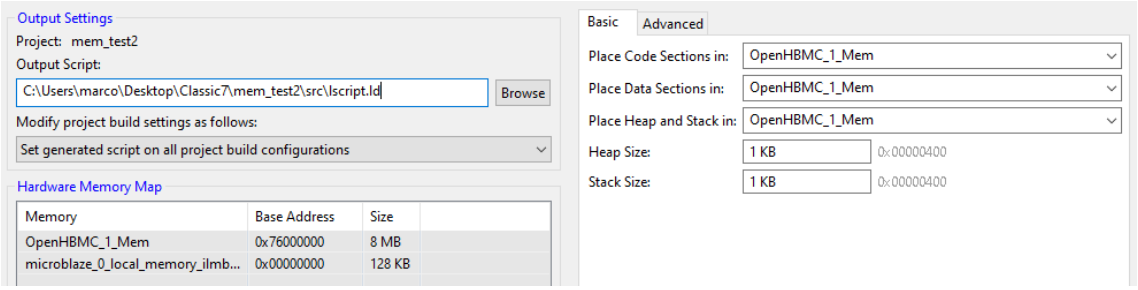

Figura 4.1: Asignación de memoria en el menú '*Generate Linker Script*'.

A continuación, creamos una nueva configuración de depuración para cada aplicación. Los programas imprimirán mediante la USB UART los resultados por puerto serie en el ordenador. Para poder recibir los datos se usa la versión gratuita de MobaXterm. La sesión configurada debe ser comunicación serie con velocidad 9600 bps, 8 bits de datos, 1 de parada y ninguno de paridad, la misma configuración que la UartLite de nuestro diagrama de bloques. Además, se selecciona el puerto al que está conectado el dispositivo FPGA, en este caso COM4. Ya podemos ejecutar la depuración y programar el módulo.

Como se puede apreciar en las figuras 4.2 y 4.3, los resultados son favorables. Se puede verificar que la memoria está siendo utilizada, además de su capacidad. Los periféricos también funcionan correctamente y están conectados al sistema de interrupciones.

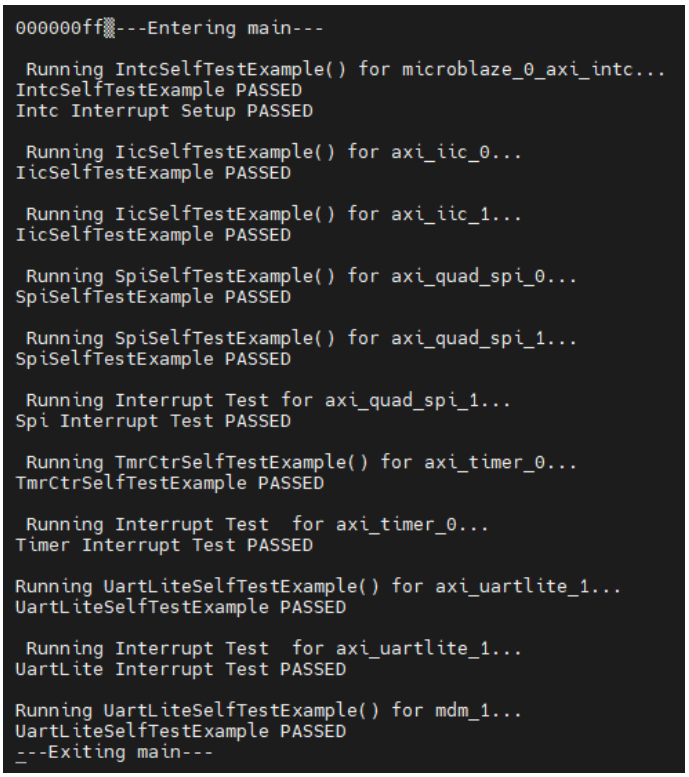

Figura 4.2: Resultado de '*Peripheral Tests*' en el terminal serie.

Figura 4.3: Resultado de '*Memory Tests*' en el terminal serie.

4.2. Comunicación con el ADXL345

Para la comunicación con el acelerómetro, se crea una nueva aplicación. Como código inicial y de referencia, se parte de un ejemplo en C proporcionado por Analog Devices [21] donde ya están implementadas varias funcionalidades. Éste está pensado para trabajar con el acelerómetro en modo SPI, por lo que deben ser cambiadas todas las funciones a I^2C. Además, los periféricos usados no son los proporcionados por Vivado, por lo que también es necesario modificarlos.

Para usar nuestros periféricos, se parte de los ejemplos del GitHub de Xilinx. En concreto, el ejemplo 'xiic_eeprom_example.c' [22] para el controlador I^2C y el ejemplo

'xtmrctr_intr_example.c' [23] para el temporizador. También se añaden las respectivas bibliotecas de cada componente, disponibles en la misma página.

El programa de ejemplo consta de un menú principal donde se puede elegir entre varias opciones:

- Habilitar las mediciones del ADXL345.
- Desactivar las mediciones del ADXL345.
- Imprimir por pantalla la aceleración.
- Seleccionar el rango de medida.
- Cambiar la tasa de adquisición.
- Activar/desactivar las interrupciones por toque.
- Detener todas las acciones.

Como en parte se ha expresado antes, se modificó el código para añadir las siguientes funcionalidades:

- Cambio de la comunicación SPI con el ADXL345 por I^2C.
- Nueva función que imprime mediciones de tiempo y aceleración en los tres ejes con la posibilidad de parar o retomar la adquisición. Esto se hace con un toque al acelerómetro. Además, se añade la opción de volver al menú principal con un doble toque.
- Una función que imprime por pantalla los valores de *offset* para calibrar el ADXL345 a nivel de registro.
- Función de auto test para verificar si el ADXL345 se encuentra dentro de los rangos permitidos por el fabricante para funcionar de manera confiable.

Las partes principales del código son:

- 'main.c', donde se elige la acción a ejecutar.
- 'ADXL345.h'. Se definen todas las direcciones de memoria de los periféricos del sistema MicroBlaze, las direcciones de memoria de todos los registros del ADXL345 y se declaran las funciones.
- 'ADXL345.c'. Se implementan las funciones declaradas en 'ADXL345.h'.

A continuación, se explican las funciones más relevantes del código, sin tener en cuenta la parte de comunicación con la tarjeta MicroSD. El código completo se puede consultar en el anexo C.

· **Función main**. Código 4.1.

Inicia la plataforma de hardware del sistema embebido e inicializa los cachés de datos y de instrucciones. Se inicializa el ADXL345 y se imprime por pantalla el menú principal. Dependiendo de la selección por teclado en el terminal, se ejecuta la función correspondiente del switch. Cuando la función se termine de ejecutar, se podrá seleccionar otra.

Las elecciones posibles son:

- Activar la adquisición de datos del acelerómetro.
- Parar la adquisición de datos.
- Imprimir la aceleración por pantalla.
- Seleccionar el rango de medida.
- Cambiar la tasa de adquisición.
- Seleccionar interrupciones por toque .
- Imprimir por pantalla los valores de calibración de offset.
- Ejecutar el auto test en el ADXL345.

- Imprimir la aceleración en bucle.

La figura 4.4 muestra el menú de selección.

```
int main()
{
  init_platform();
  Xil_ICacheEnable();
  Xil_DCacheEnable();

  // Inicializa el ADXL345
  ADXL345_Init();

  // Imprime por pantalla el menú principal
  ADXL345_DisplayMainMenu();

  while(1)
  {

    switch(rxData)
    {
      case 'e':
        ADXL345_Run();
        break;
      case 'd':
        ADXL345_Stop();
        break;
      case 'a':
        ADXL345_DisplayAccel();
        break;
      case 's':
        ADXL345_SelectRange();
        break;
      case 'r':
        ADXL345_AcquisitionRate();
        break;
      case 't':
        ADXL345_SelectTap();
        break;
      case 'o':
        ADXL345_DisplayCalibrationValue();
        break;
      case 'h':
        ADXL345_DisplaySelftestDelta();
        break;
      case 'l':
        ADXL345_DisplayAccelLoop();
        break;
      case 0:
        break;
      default:
        xil_printf("\n> Error! Por favor, selecciona una de las opciones
          ↪ enumeradas a continuación\r");
        ADXL345_DisplayMenu();
        break;
    }
  }

  Xil_DCacheDisable();
  Xil_ICacheDisable();

  return 0;
}
```

Código 4.1: Función main.

```
Programa del ADXL345

Opciones:
    [e] Habilitar Mediciones
    [d] Deshabilitar Mediciones
    [a] Imprimir Aceleración
    [s] Seleccionar Rango de Medida
    [r] Cambiar Tasa de Adquisición
    [t] Seleccionar Interrupciones por Toque
    [o] Calcular el valor de calibración de Offset
    [h] Auto-test
    [l] Imprimir aceleración en bucle
Por favor, selecciona una opción
```

Figura 4.4: Menú principal de la aplicación *software* en el terminal serie.

· **Función ADXL345_WriteReg**. Código 4.2.

Recibe como parámetros la dirección de registro del ADXL345 a la que se ha de acceder y los datos que se van a escribir.

1. Inicializa el periférico I^2C.
2. Envía la dirección del ADXL345 y el buffer de datos.
3. Espera a que finalice la transmisión.
4. Detiene el periférico I^2C.

```
1  void ADXL345_WriteReg(u8 RegAddress, u8 txData)
2  {
3
4    u8 WriteBuffer[2] = {RegAddress, txData};
5
6    TransmitComplete = 1;   // Valor por defecto
7    XIic_Start(&IicInstance);             // Inicializa el dispositivo IIC
8    XIic_MasterSend(&IicInstance, WriteBuffer, 2);  // Envía los datos
9
10   //Espera a que finalice la transmisión
11   while ((TransmitComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {
12
13   }
14
15   XIic_Stop(&IicInstance);  // Detiene el dispositivo IIC
16
17 }
```

Código 4.2: Función ADXL345_WriteReg.

· **Función ADXL345_ReadReg**. Código 4.3.

Recibe como parámetro la dirección de registro del ADXL345 que se desea leer.

1. Inicializa el periférico I^2C.
2. Envía la dirección del dispositivo en modo escritura y la dirección del registro de memoria del ADXL345.
3. Espera a que finalice la transmisión y detiene el periférico I^2C.
4. Realiza la secuencia de inicialización del periférico y envía la dirección del dispositivo en modo lectura.
5. Espera a recibir los datos y desconecta el dispositivo I^2C.
6. Devuelve el byte recibido.

```
int ADXL345_ReadReg(u8 RegAddress)
{

  u8 WriteBuffer[1] = {RegAddress};
  u8 ReadBuffer[1] = {0x00};

  TransmitComplete = 1;               // Valor por defecto
  XIic_Start(&IicInstance);           // Inicializa el dispositivo IIC
  XIic_MasterSend(&IicInstance, WriteBuffer, 1);  // Envía los datos

  //Espera a que finalice la transmisión
  while ((TransmitComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {

  }

  XIic_Stop(&IicInstance);  // Detiene el dispositivo IIC
  ReceiveComplete = 1;    // Valor por defecto

  XIic_Start(&IicInstance);   // Inicializa el dispositivo IIC
  XIic_MasterRecv(&IicInstance, ReadBuffer, 1); // Recibe los datos

  //Espera a que finalice la recepción
  while ((ReceiveComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {

  }

  XIic_Stop(&IicInstance);  // Detiene el dispositivo IIC
  rxData = ReadBuffer[0];

  return(rxData);
}
```

Código 4.3: Función ADXL345_ReadReg.

· **Función ADXL345_BurstReadReg**. Código 4.4.

Recibe como parámetro la dirección del primer registro que se desea leer.

Funciona igual que la función anterior solo que mantiene la comunicación con el acelerómetro hasta recibir dos bytes de datos. Éstos se leen de registros consecutivos.

Es útil a la hora de medir la aceleración, ya que el valor de cada eje está representado en dos registros contiguos. Es el caso de los registros que van del 0x32 al 0x37.

Devuelve el valor de los dos registros en formato int, siendo el primer registro leído el menos significativo.

La figura 4.5 muestra un ejemplo de lectura de los registros de aceleración del eje X.

```
int ADXL345_BurstReadReg(u8 RegAddress)
{

  u8 WriteBuffer[1] = {RegAddress};
  u8 ReadBuffer[2] = {0x00, 0x00};

  TransmitComplete = 1;   // Valor por defecto
  XIic_Start(&IicInstance);             // Inicializa el dispositivo IIC
  XIic_MasterSend(&IicInstance, WriteBuffer, 1);  // Envía los datos

  //Espera a que finalice la transmisión
  while ((TransmitComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {

  }

  XIic_Stop(&IicInstance);  // Detiene el dispositivo IIC
  ReceiveComplete = 1;    // Valor por defecto

  XIic_Start(&IicInstance);             // Inicializa el dispositivo IIC
  XIic_MasterRecv(&IicInstance, ReadBuffer, 2); // Recibe los datos

  //Espera a que finalice la recepción
  while ((ReceiveComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {

  }

  XIic_Stop(&IicInstance); // Detiene el dispositivo IIC

  // Organiza los datos u8 recibidos en u16
  BurstRead = ((ReadBuffer[1] & 0xFF) << 8) | (ReadBuffer[0] & 0xFF);

  return(BurstRead);
}
```

Código 4.4: Función ADXL345_BurstReadReg.

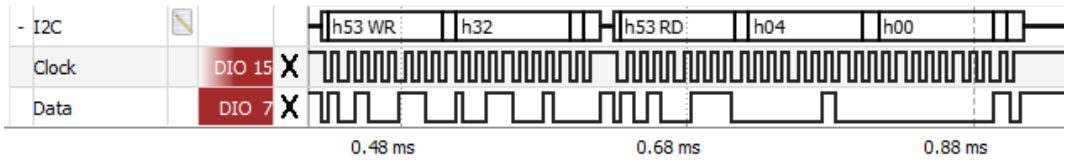

Figura 4.5: Lectura de dos registros consecutivos del ADXL345 en modo I^2C. Captura tomada con el *software* WaveForms.

· **Función ADXL345_Init**. Código 4.5.

No recibe ningún parámetro ni devuelve ningún valor. Inicia todos los periféricos

conectándolos al sistema de interrupciones y manda la secuencia de inicialización al ADXL345. Los pasos seguidos son:

1. Inicializa el sistema de interrupciones y los dispositivos conectados.
2. Se establece la dirección de 7 bits del ADXL345 para las posteriores comunicaciones I^2C, en este caso 0x53.
3. Se configura la tasa de salida del ADXL345 a 12,5Hz.
4. Se configura el formato de datos de salida a ±16g y resolución de 13 bits.
5. Se establece el umbral de aceleración a partir del cual se detectan los toques en el dispositivo. También la máxima duración de un evento para que pueda ser considerado como toque.
6. Se establece el tiempo de latencia, tiempo que transcurre hasta que otro evento pueda ser detectado como segundo toque. Asimismo, la cantidad de tiempo que puede pasar desde la expiración del tiempo de latencia para que un segundo toque pueda ser válido.
7. Se habilita la posibilidad de que se detecten toques en todos los ejes.
8. Se establecen los umbrales de actividad e inactividad.
9. Se habilita la detección de actividad en todos los ejes.
10. Se establece el umbral de detección de caída libre y el tiempo mínimo para que se considere el evento.
11. Se habilitan las interrupciones por toque doble y simple.
12. Se escriben en registro los valores de corrección del error de offset si se han declarado al comienzo de la función.

```
void ADXL345_Init(void)
{
  signed char X_CALIB = 0, Y_CALIB = 0, Z_CALIB = 0;  // Dejar en 0 si el offset
          ↪ del dispositivo no ha sido calibrado

  // Inicializa el Subsistema de Interrupciones y los dispositivos conectados
  SetupIntrSystem(&IntcInstance, &IicInstance, &TimerCounterInst,
          ↪ TMRCTR_DEVICE_ID, TMRCTR_INTERRUPT_ID);
  XIic_SetAddress(&IicInstance, XII_ADDR_TO_SEND_TYPE, ADXL345_ADDRESS);  //
          ↪ Establece la dirección del ADXL345

  ADXL345_WriteReg(ADXL345_BW_RATE, (0x07 << ADXL345_RATE));      // Tasa de Datos
          ↪ de Salida = 12.5 Hz
  ADXL345_WriteReg(ADXL345_DATA_FORMAT, ((1 << ADXL345_FULL_RES)  | // Formato de
          ↪ Datos = +- 16g, Resolución Completa
                    (3 << ADXL345_Range)));
  ADXL345_WriteReg(ADXL345_THRESH_TAP, 0x20);             // Umbral de Toque
  ADXL345_WriteReg(ADXL345_DUR, 0x0D);                  // Duración de Toque
  ADXL345_WriteReg(ADXL345_Latent, 0x50);                 // Latencia de Toque
  ADXL345_WriteReg(ADXL345_Window, 0xF0);                 // Ventana de Toque
  ADXL345_WriteReg(ADXL345_TAP_AXES, ((1 << ADXL345_TAP_X_en)   | // Habilitar
          ↪ Toques en todos los ejes
                  (1 << ADXL345_TAP_Y_en)   |
                  (1 << ADXL345_TAP_Z_en)));
  ADXL345_WriteReg(ADXL345_THRESH_ACT, 0x08);             // Umbral de Actividad
  ADXL345_WriteReg(ADXL345_ACT_INACT_CTL, ((1 << ADXL345_ACT_X_en)| // Ejes de
          ↪ Actividad
                    (1 << ADXL345_ACT_Y_en) |
                    (1 << ADXL345_ACT_Z_en)));
  ADXL345_WriteReg(ADXL345_THRESH_FF, 0x08);              // Umbral de Caída Libre
  ADXL345_WriteReg(ADXL345_TIME_FF, 0x0A);              // Ventana de Caída Libre
  ADXL345_WriteReg(ADXL345_FIFO_CTL, 0x01);
  ADXL345_WriteReg(ADXL345_INT_ENABLE, ((1 << ADXL345_SINGLE_TAP) | // Habilitar
          ↪ Toques Simples y Dobles
                  (1 << ADXL345_DOUBLE_TAP) |
                  (0 << ADXL345_Activity) |
                  (0 << ADXL345_FREE_FALL)  |
                  (0 << ADXL345_DATA_READY)));
  ADXL345_WriteReg(ADXL345_OFSX, X_CALIB);                // Escribir Offset del eje
          ↪ X
  ADXL345_WriteReg(ADXL345_OFSY, Y_CALIB);                // Escribir Offset del eje
          ↪ Y
  ADXL345_WriteReg(ADXL345_OFSZ, Z_CALIB);                // Escribir Offset del eje
          ↪ Z
  ADXL345_ReadAllAxes();
}
```

Código 4.5: Función ADXL345_Init.

· **Función ADXL345_Display_G_Force**. Código 4.6.

Imprime por pantalla la aceleración del eje seleccionado en el parámetro 'axis'. La secuencia es la siguiente:

1. Lee la aceleración en un eje con la función ADXL345_BurstReadReg.
2. Pasa las medidas de Ca2 a *unsigned*.
3. Pasa las medidas de LSB a gravedades, representando cada bit menos significativo 3,9 mg.

4. Imprime por pantalla el valor de aceleración con signo y tres decimales.

Para entender esta función es necesario conocer cómo se organizan los datos de aceleración en los registros (Figura 4.6). La medida de la aceleración en cada eje está representada por 2 bytes. Los datos están justificados a la derecha en Ca2 y cada LSB representa 3,9 mg. Dependiendo del rango seleccionado ±[16, 8, 4, 2]g, la magnitud de la aceleración en los dos registros ocupará [13, 12, 11, 10] bits. Los tres bits más significativos representan el signo. En los modos de rango ±[8, 4, 2]g, los bits extras serán '0' para valores de aceleración positivos y '1' para negativos.

```
void ADXL345_Display_G_Force(char axis)
{
  u32   gForce    = 0;
  u32   gForceCalc  = 0;
  float   value     = 0;
  u32   whole     = 0;
  u32   thousands   = 0;
  int   negative  = 0;

  // Selecciona el eje a leer
  switch(axis)
  {
    case 'x':
      gForce = ADXL345_BurstReadReg(ADXL345_DATA_X0);
      xil_printf("[X = ");
      break;
    case 'y':
      gForce = ADXL345_BurstReadReg(ADXL345_DATA_Y0);
      xil_printf("[Y = ");
      break;
    case 'z':
      gForce = ADXL345_BurstReadReg(ADXL345_DATA_Z0);
      xil_printf("[Z = ");
      break;
    default:
      break;
  }

  // Si el resultado de la lectura es negativo, aplicar relleno con 0xFFFFF000
  if((gForce & 0x1000) == 0x1000)
  {
    negative = 1;
    gForceCalc = 0xFFFFF000 | gForce;
    gForceCalc = gForceCalc - 1;
    gForceCalc = ~gForceCalc;

  }
  // Si el resultado es positivo, no modificar
  else
  {
    gForceCalc = gForce & 0x0FFF;
  }

  // Convierte a decimal
  value = ((float)gForceCalc * 0.0039);

  // Prepara e imprime los datos
  // Si son positivos
  if(negative == 0)
  {
    whole = value;
    thousands = (value - whole) * 1000;
```

```
53      if(thousands > 99)
54      {
55        xil_printf("+%d.%3d] ", whole, thousands);
56      }
57      else if(thousands > 9)
58      {
59        xil_printf("+%d.0%2d] ", whole, thousands);
60      }
61      else
62      {
63        xil_printf("+%d.00%1d] ", whole, thousands);
64      }
65    }
66    // Si son negativos
67    else
68    {
69      whole = value;
70      thousands = (value - whole) * 1000;
71
72      if(thousands > 99)
73      {
74        xil_printf("-%d.%3d] ", whole, thousands);
75      }
76      else if(thousands > 9)
77      {
78        xil_printf("-%d.0%2d] ", whole, thousands);
79      }
80      else
81      {
82        xil_printf("-%d.00%1d] ", whole, thousands);
83      }
84    }
85  }
```

Código 4.6: Función ADXL345_Display_G_Force.

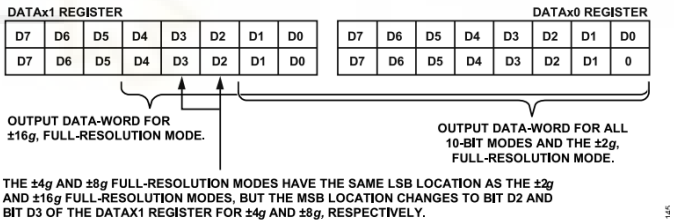

Figura 4.6: Formato de datos de aceleración en el ADXL345

· **Función ADXL345_Run**. Código 4.7.

Habilita las mediciones en el ADXL345.

Comprueba si las mediciones están habilitadas ya con la variable global 'isRunning'. Si no lo están, pone a '1' el bit que habilita las mediciones en el registro POWER_CTL.

```
void ADXL345_Run(void)
{
  if(isRunning == 0)
  {
    ADXL345_WriteReg(ADXL345_POWER_CTL, (1 << ADXL345_Measure));
    isRunning = 1;
  }
  xil_printf("\n\r>Mediciones del ADXL345 Habilitadas!\n\r\n\r");
  ADXL345_DisplayMenu();
}
```

Código 4.7: Función ADXL345_Run.

· **Función ADXL345_Stop**. Código 4.8.

Detiene las mediciones del ADXL345.

Comprueba si las mediciones están habilitadas ya con la variable global 'isRunning'. Si lo están, pone a '0' el bit que habilita las mediciones en el registro POWER_CTL.

```
void ADXL345_Stop(void)
{
  if(isRunning == 1)
  {
    ADXL345_WriteReg(ADXL345_POWER_CTL, (0 << ADXL345_Measure));
    isRunning = 0;
  }
  xil_printf("\n\r>Mediciones del ADXL345 Deshabilitadas!\n\r\n\r");
  ADXL345_DisplayMenu();
}
```

Código 4.8: Función ADXL345_Stop.

· **Función ADXL345_DisplayAccel**. Código 4.9.

Imprime por pantalla el valor de aceleración en los tres ejes.

1. Si las mediciones del registro están habilitadas, lee el registro de interrupciones INT_SOURCE para verificar si hay una nueva medida para leer.
2. Si el resultado es verdadero, lee los valores de aceleración de todos los ejes y los imprime por pantalla.

En registro de interrupciones, los bits en '1' indican que sus respectivas funciones han desencadenado un evento, mientras que un bit en '0' indica que el evento correspondiente no ha ocurrido. La figura 4.7 muestra la organización de los distintos eventos dentro del registro.

Los eventos que pueden desencadenar interrupciones, en caso de que éstas estén activadas son: los toques simples, toques dobles, medidas disponibles, caída libre, la actividad, la inactividad, el desbordamiento de la memoria FIFO o su detector de nivel.

```
void ADXL345_DisplayAccel(void)
{
  if(isRunning == 1)
  {
    if(ADXL345_ReadReg(ADXL345_INT_SOURCE) & 0x80)  // Comprueba si los datos
          ↪ están listos
    {
      ADXL345_DisplayAllAxes();
    }
  }
  else if(isRunning == 0)
  {
    xil_printf("\n\r!!! Mediciones no habilitadas. Por favor habilítalas antes de
          ↪ realizar la acción !!!\n\r");
  }
  ADXL345_DisplayMenu();
}
```

Código 4.9: Función ADXL345_DisplayAccel.

Table 32. Register 0x30—Bits[D7:D4]

D7	D6	D5	D4
DATA_READY	SINGLE_TAP	DOUBLE_TAP	Activity

Table 33. Register 0x30—Bits[D3:D0]

D3	D2	D1	D0
Inactivity	FREE_FALL	Watermark	Overrun

Figura 4.7: Registro de interrupciones del ADXL345.

· **Función ADXL345_SelectRange**. Código 4.10.

Selecciona el nuevo rango de medida del ADXL345 dependiendo del carácter introducido por teclado. No está expresado en la función pero [1, 2, 3, 4] seleccionan ±[2, 4, 8, 16] g respectivamente (Figura 4.8).

1. Se lee el valor introducido por teclado.
2. Se lee el registro DATA_FORMAT.
3. Dependiendo del valor introducido, se cambian los dos bits menos significativos y se vuelve a escribir el registro. Esto se hace para no perder la información del resto de bits al sobrescribir.

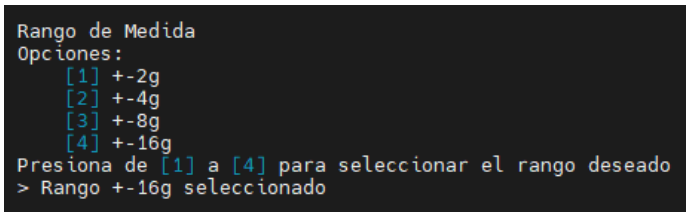

Figura 4.8: Menú de rango de medida.

```
void ADXL345_SelectRange(void)
{
  u8 rx    = 0;
  u8 rxReg  = 0;

  ADXL345_DisplayRangeMenu();

  scanf(" %c", &rx);

  // Convierte el formato ASCII
  rx = rx - 0x31;

  if ((rx >= 0x00) && (rx <= 0x03))
    {
    rxReg = (ADXL345_ReadReg(ADXL345_DATA_FORMAT) & 0xFC) | rx;

    // Cambia el valor del registro DATA_FORMAT
    ADXL345_WriteReg(ADXL345_DATA_FORMAT, rxReg);
    }

  // Imprime la nueva configuración
  switch(rx)
  {
    case 0:
      xil_printf("> Rango +-2g seleccionado\n\r");
      break;
    case 1:
      xil_printf("> Rango +-4g seleccionado\n\r");
      break;
    case 2:
      xil_printf("> Rango +-8g seleccionado\n\r");
      break;
    case 3:
      xil_printf("> Rango +-16g seleccionado\n\r");
      break;
    default:
      xil_printf("> Rango de medida erróneo\n\r");
      break;
  }

  ADXL345_DisplayMenu();
}
```

Código 4.10: Función ADXL345_SelectRange.

· **Función ADXL345_AcquisitionRate**. Código 4.11.

Selecciona la nueva tasa de medida del ADXL345 dependiendo del carácter introducido por teclado. [1, 2, 3, 4, 5, 6, 7, 8, 9] selecciona [6.25, 12.5, 25, 50, 100, 200, 400, 800, 1600] Hz (Figura 4.9).

1. Se lee el valor introducido por teclado.

2. Se lee el registro BW_RATE.

3. Dependiendo del valor introducido, se cambian los cuatro bits menos significativos y se escribe el registro de vuelta.

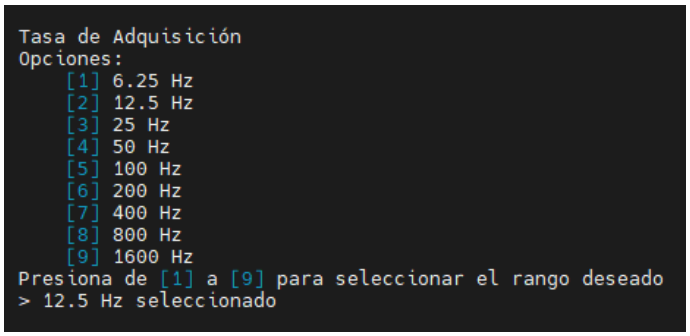

Figura 4.9: Menú de tasa de adquisición.

```c
void ADXL345_AcquisitionRate(void)
{
  u8 rx    = 0;
  u8 rxReg  = 0;

  ADXL345_DisplayRateMenu();

  scanf(" %c", &rx);

  // Convierte el formato ASCII
  rx = rx - 0x30 + 0x05;

  if ((rx >= 0x06) && (rx <= 0x0E))
  {
    rxReg = (ADXL345_ReadReg(ADXL345_BW_RATE) & 0xF0) | rx;

    // Cambia el valor del registro BW_RATE
    ADXL345_WriteReg(ADXL345_BW_RATE, rxReg);
  }

  // Imprime la nueva configuración
  switch(rx)
  {
    case 0x06:
      xil_printf("> 6.25 Hz seleccionado\n\r");
      break;
    case 0x07:
      xil_printf("> 12.5 Hz seleccionado\n\r");
      break;
    case 0x08:
      xil_printf("> 25 Hz seleccionado\n\r");
      break;
    case 0x09:
      xil_printf("> 50 Hz seleccionado\n\r");
      break;
    case 0x0A:
      xil_printf("> 100 Hz seleccionado\n\r");
      break;
    case 0x0B:
      xil_printf("> 200 Hz seleccionado\n\r");
      break;
```

```
42      case 0x0C:
43        xil_printf("> 400 Hz seleccionado\n\r");
44        break;
45      case 0x0D:
46        xil_printf("> 800 Hz seleccionado\n\r");
47        break;
48      case 0x0E:
49        xil_printf("> 1600 Hz seleccionado\n\r");
50        break;
51      default:
52        xil_printf("> Rango de adquisición erroneo\n\r");
53        break;
54    }
55
56    ADXL345_DisplayMenu();
57  }
```

Código 4.11: Función ADXL345_AcquisitionRate.

· **Función ADXL345_SelectTap**. Código 4.12.

Habilita o deshabilita las interrupciones por toques simples o dobles.

1. Se lee el valor introducido por teclado.
2. Se lee el registro INT_ENABLE
3. Dependiendo del valor introducido, se cambian los bits 5 y 6 del registro sin modificar el resto de bits.

```
1  void ADXL345_SelectTap(void)
2  {
3    u8 rx    = 0;
4    u8 rxReg  = 0;
5
6    ADXL345_DisplayTapMenu();
7
8    scanf(" %c", &rx);
9
10   // Habilita e imprime las opciones deseadas
11   switch(rx)
12   {
13     case '1':
14       rxReg = (ADXL345_ReadReg(ADXL345_INT_ENABLE) & 0x9F) | (1 <<
           ↪ ADXL345_SINGLE_TAP);
15       ADXL345_WriteReg(ADXL345_INT_ENABLE, rxReg);
16       xil_printf("> Toque Simple seleccionado\n\r");
17       break;
18     case '2':
19       rxReg = (ADXL345_ReadReg(ADXL345_INT_ENABLE) & 0x9F) | (1 <<
           ↪ ADXL345_DOUBLE_TAP);
20       ADXL345_WriteReg(ADXL345_INT_ENABLE, rxReg);
21       xil_printf("> Toque Doble seleccionado\n\r");
22       break;
23     case '3':
24       rxReg = (ADXL345_ReadReg(ADXL345_INT_ENABLE) & 0x9F) | (1 <<
           ↪ ADXL345_SINGLE_TAP) | (1 << ADXL345_DOUBLE_TAP);
25       ADXL345_WriteReg(ADXL345_INT_ENABLE, rxReg);
26       xil_printf("> Toques Simple y Doble seleccionados\n\r");
27       break;
```

```
28      case '4':
29        rxReg = ADXL345_ReadReg(ADXL345_INT_ENABLE) & 0x9F;
30        ADXL345_WriteReg(ADXL345_INT_ENABLE, rxReg);
31        xil_printf("> Toques deshabilitados\n\r");
32        break;
33      default:
34        xil_printf("> Opción de toque erronea\n\r");
35        break;
36    }
37
38    ADXL345_DisplayMenu();
39  }
```

Código 4.12: Función ADXL345_SelectTap

· **Función ADXL345_CalculateAverage**. Código 4.13.

Toma 100 medidas de un eje y obtiene el valor medio. Funciona de manera similar a ADXL345_Display_G_Force. Devuelve la media.

```
1  long int ADXL345_CalculateAverage(char axis)
2  {
3    u32 gForce, gForceCalc;
4    long int Avg = 0;
5    int negative, i;
6
7    for(i=0;i<100;i++){   //Toma 100 mediciones para hacer la media
8      WaitFor(10000);
9      switch(axis)
10     {
11       case 'x':
12         gForce = ADXL345_BurstReadReg(ADXL345_DATA_X0);
13         break;
14       case 'y':
15         gForce = ADXL345_BurstReadReg(ADXL345_DATA_Y0);
16         break;
17       case 'z':
18         gForce = ADXL345_BurstReadReg(ADXL345_DATA_Z0);
19         break;
20       default:
21         break;
22     }
23
24     // Si el resultado de la lectura es negativo, aplicar relleno con 0xFFFFF000
25     if((gForce & 0x1000) == 0x1000)
26     {
27       negative = 1;
28       gForceCalc = 0xFFFFF000 | gForce;
29       gForceCalc = gForceCalc - 1;
30       gForceCalc = ~gForceCalc;
31
32     }
33     // Si el resultado es positivo, no modificar
34     else
35     {
36       negative = 0;
37       gForceCalc = gForce & 0x0FFF;
38     }
39     // Sumatorio de los valores obtenidos
40     if(negative == 0) {
41       Avg += gForceCalc;
42     }
43         else
44     {
45       Avg -= gForceCalc;
```

```
46      }
47    }
48
49    Avg = (long int)round((double)Avg / 100); //Calcula la media
50
51    return Avg;
52  }
```

Código 4.13: Función ADXL345_CalculateAverage

· **Función ADXL345_CalculateOffset**. Código 4.14.

Calcula el offset en los ejes a partir del valor medio. Se asume que el acelerómetro se encuentra en orientación X=0g, Y=0g, Z=1g.

El valor de *offset* es el mismo que el valor medio para los ejes X e Y. Al valor del eje Z se le debe restar una gravedad. Al ser la sensibilidad 3.9 mg/LSB, se restan 256 LSBs.

```
1  long int ADXL345_CalculateOffset(char axis)
2  {
3    long int Offset;
4
5    Offset = ADXL345_CalculateAverage(axis);
6
7    if(axis == 'z')
8    {
9      Offset -= 256;    //Descontamos 1g del eje z (1/0.0039)
10   }
11
12   return Offset;
13 }
```

Código 4.14: Función ADXL345_CalculateOffset

· **Función ADXL345_DisplayCalibrationValue**. Código 4.15.

Calcula los valores de calibración a partir de los valores de offset de los distintos ejes. La forma de calcularlos está explicada en la 'ADXL345 Quick Start Guide' de Analog Devices [24]. La figura 4.10 muestra el protocolo.

Una vez calculados, debemos de reprogramar la FPGA cambiando las variables X_CALIB, Y_CALIB, Z_CALIB de la función ADXL345_Init para actualizar los registros de calibración. En la figura 4.11 se puede apreciar el cambio en las medidas de aceleración antes y después de corregir el *offset*.

```c
void ADXL345_DisplayCalibrationValue(void)
{
  long int Offset, X_CALIB, Y_CALIB, Z_CALIB;

  if(isRunning == 1)
  {
    // Esperar 11.1ms después de la inicialización del ADXL345. No necesario en este código.
    // Para una calibración más afinada corregir el offset a nivel de software
    xil_printf("\nEjecuta de nuevo si el sensor no está posicionado en la orientación X = 0g, Y = 0g, Z = +1g");
    xil_printf(" o si las variables XYZ_CALIB en ADXL345_Init() no están inicializadas a '0'.\n\r");
    xil_printf("El formato de datos debe ser: rango +-16g y modo 13 bits (por defecto)\n\r");

    // Cálculo de XYZ_CALIB basado en la fórmula de la guía de Analog Devices (an-1077)
    Offset = ADXL345_CalculateOffset('x');
    X_CALIB = -((long int)round((double)Offset / 4));
    Offset = ADXL345_CalculateOffset('y');
    Y_CALIB = -((long int)round((double)Offset / 4));
    Offset = ADXL345_CalculateOffset('z');
    Z_CALIB = -((long int)round((double)Offset / 4));

    // Cada LSB del resultado de calibración representa 15.6 mg
    xil_printf("Valores de calibración:\n\r");
    xil_printf("    > X_CALIB = %ld\n\r", X_CALIB);
    xil_printf("    > Y_CALIB = %ld\n\r", Y_CALIB);
    xil_printf("    > Z_CALIB = %ld\n\r", Z_CALIB);

  }
  else if(isRunning == 0)
  {
    xil_printf("\n\r!!! Mediciones no habilitadas. Por favor, habilítalas antes de realizar la acción !!!\n\r");
  }
  ADXL345_DisplayMenu();
}
```

Código 4.15: Función ADXL345_DisplayCalibrationValue

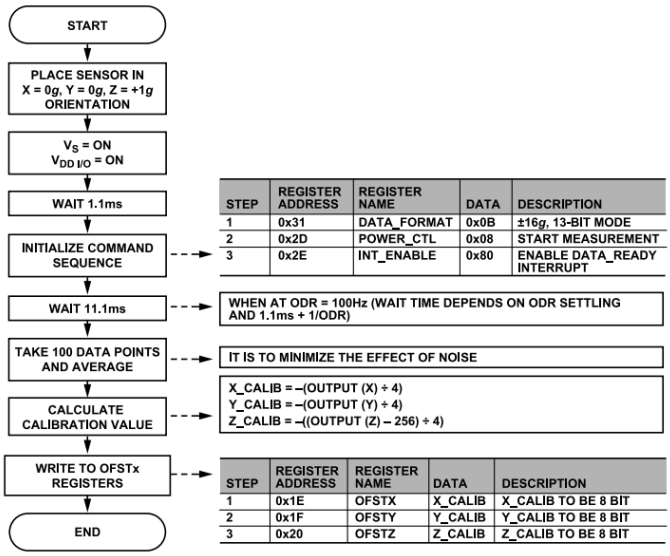

Figura 4.10: Secuencia para la calibración del error de *offset* en el ADXL345.

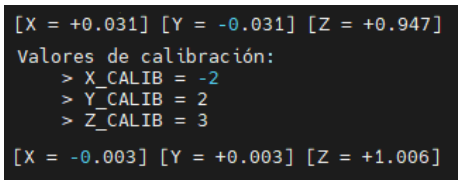

Figura 4.11: Medidas de aceleración en reposo antes y después de la corrección del error de *offset*.

· **Función ADXL345_DisplaySelftestDelta**. Código 4.16.

Calcula y verifica si los valores de auto test son correctos. En la 'ADXL345 Quick Start Guide' se explica la secuencia a seguir (Figura 4.12).

1. Se calcula en valor medio de aceleración en los tres ejes.
2. Se activa el bit de auto test en el registro DATA_FORMAT. A partir de ese momento, el sensor aplica sobre sí mismo una fuerza electrostática controlada.
3. Se vuelve a calcular el valor medio de aceleración en los tres ejes.
4. La diferencia entre el valor medio con el auto test activado y el valor medio sin activar representa el resultado de auto test.
5. Si el resultado está comprendido entre los límites representados en la figura 4.13, se toma como resultado satisfactorio y el dispositivo se considera fiable.

Un ejemplo de la ejecución de la función se da en la figura 4.14.

```
1  void ADXL345_DisplaySelftestDelta(void)
2  {
3    long int X_ST_OFF, Y_ST_OFF, Z_ST_OFF, X_ST_ON, Y_ST_ON, Z_ST_ON;
4    long int X_ST, Y_ST, Z_ST;
5
6    if(isRunning == 1)
7    {
8      // Esperar 11.1ms después de la inicialización del ADXL345. No necesario en
           ↪ este código.
9      xil_printf("\nEl sensor debe ser colocado en un ambiente estable mientras se
           ↪ ejecuta la secuencia de auto-test.\n\r");
10
11     X_ST_OFF = ADXL345_CalculateAverage('x');
12     Y_ST_OFF = ADXL345_CalculateAverage('y');
13     Z_ST_OFF = ADXL345_CalculateAverage('z');
14
15     ADXL345_WriteReg(ADXL345_DATA_FORMAT, ((1 << ADXL345_SELF_TEST) |   // Activa
           ↪ el auto-test
16                                  (1 << ADXL345_FULL_RES) | // Formato de datos =
           ↪ +- 16g, Resolución Completa
```

```
17                          (3 << ADXL345_Range)));
18
19      X_ST_ON = ADXL345_CalculateAverage('x');
20      Y_ST_ON = ADXL345_CalculateAverage('y');
21      Z_ST_ON = ADXL345_CalculateAverage('z');
22
23      ADXL345_WriteReg(ADXL345_DATA_FORMAT, ((0 << ADXL345_SELF_TEST) |   //
            ↪ Desactiva el auto-test
24                          (1 << ADXL345_FULL_RES)  | // Formato de datos = +- 16g,
            ↪ Resolución Completa
25                          (3 << ADXL345_Range)));
26
27      X_ST = X_ST_ON - X_ST_OFF;                          // Calcula los valores de
            ↪ auto-test
28      Y_ST = Y_ST_ON - Y_ST_OFF;
29      Z_ST = Z_ST_ON - Z_ST_OFF;
30
31      xil_printf("Los valores de auto-test son:\n\r");
32      xil_printf("    > X_ST = %d LSBs\n\r", X_ST);
33      xil_printf("    > Y_ST = %d LSBs\n\r", Y_ST);
34      xil_printf("    > Z_ST = %d LSBs\n\r", Z_ST);
35
36      /* Para Vs = 3.3V, los valores de auto-test son: X_ST = [0.35, 3.70]g, Y_ST =
            ↪ [-3.70, -0.35]g, Z_ST = [0.45, 5]g
37         Multiplica por 256 para cambiar de LSB a g */
38      if(X_ST < 0.35*256 || X_ST > 3.7*256 || Y_ST < -3.7*256 || Y_ST > -0.35*256 ||
            ↪ Z_ST < 0.45*256 || Z_ST > 5*256)
39      {
40        xil_printf("Auto-test no superado\n\r");
41      }
42      else
43      {
44        xil_printf("Auto-test satisfactorio\n\r");
45      }
46    }
47    else if(isRunning == 0)
48    {
49      xil_printf("\n\r!!! Mediciones no habilitadas. Por favor, habilítalas antes de
            ↪ realizar la acción !!!\n\r");
50    }
51    ADXL345_DisplayMenu();
52  }
```

Código 4.16: Función ADXL345_DisplaySelftestDelta

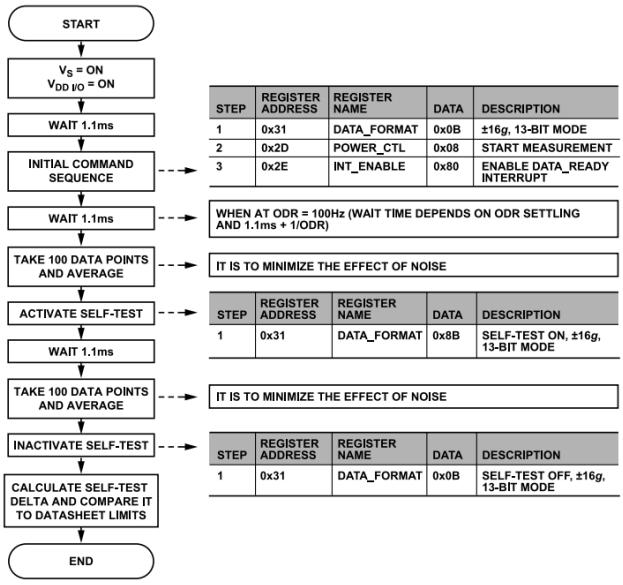

Figura 4.12: Secuencia para la realización del auto test en el ADXL345.

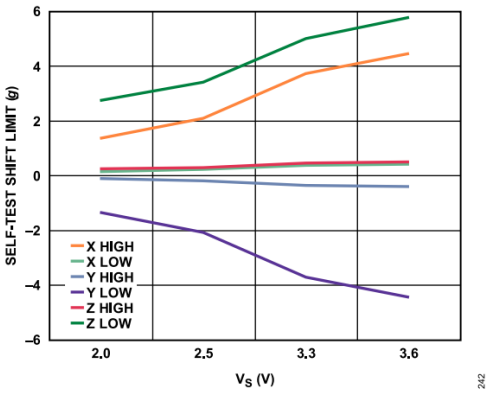

Figura 4.13: Límites de los valores de auto test en el ADXL345 dependiendo del voltaje de alimentación.

```
Los valores de auto-test son:
    > X_ST = 158 LSBs
    > Y_ST = -280 LSBs
    > Z_ST = 554 LSBs
Auto-test satisfactorio
```

Figura 4.14: Resultado de los valores de auto test en el terminal.

· **Función GetTime**. Código 4.17.

Se imprime por pantalla el tiempo transcurrido desde que se inicializó el contador. El contador es de 64 bits y está formado por dos contadores de 32 bits. Al usar en el sistema un reloj de 100MHz, cada LSB del contador menos significativo representa 10 ns mientras que cada bit del contador más significativo representa $2^{32} \cdot 10$ ns.

1. Se conecta el contador al sistema de interrupciones.
2. Se leen los valores de los dos contadores.
3. Se calcula el tiempo en segundos y se imprime por pantalla.

```
1  void GetTime(XTmrCtr *TmrCtrInstancePtr, u16 TmrCtrIntrId, INTC *IntcInstancePtr)
2  {
3    /* Configurado para un reloj de 100MHz. El contador está formado por dos
4       contadores de 32 bits. */
5
6    u32 Tiempo_s;
7    int Tiempo_ms;
8      u32 Contador1, Contador0;
9
10     // Conecta las interrupciones del contador
```

```
11    XIntc_Connect(IntcInstancePtr, TmrCtrIntrId,
                ↪ (XInterruptHandler)XTmrCtr_InterruptHandler, (void
                ↪ *)TmrCtrInstancePtr);
12
13    // Lee los registros del contador
14      Contador1 = XTmrCtr_GetValue(TmrCtrInstancePtr, (u32)TIMER_CNTR_1);
15      Contador0 = XTmrCtr_GetValue(TmrCtrInstancePtr, (u32)TIMER_CNTR_0);
16
17      if(ClockRead == 0){   // Habilita flag de primera lectura realizada
18        TimerOffset = Contador1;  // Offset del contador al reiniciar
19        ClockRead = 1;
20      }
21
22      // Calcula el tiempo en segundos y milisegundos
23      Tiempo_s = 42.94967296 * (Contador1 - TimerOffset) + 0.00000001 * Contador0;
24      Tiempo_ms = ((42.94967296 * (Contador1 - TimerOffset) + 0.00000001 *
                ↪ Contador0) * 1000) - Tiempo_s * 1000;
25
26      xil_printf("[t = %lu.", Tiempo_s);
27    if (Tiempo_ms < 10) {
28      xil_printf("00%d] ", Tiempo_ms);
29    } else if (Tiempo_ms < 100) {
30      xil_printf("0%d] ", Tiempo_ms);
31    } else {
32      xil_printf("%d] ", Tiempo_ms);
33    }
34
35    XIntc_Disconnect(IntcInstancePtr, TmrCtrIntrId);
36  }
```

Código 4.17: Función GetTime

· **Función ADXL345_DisplayAccelLoop**. Código 4.18.

Imprime en bucle la aceleración y el tiempo transcurrido desde que se llamó a la función. Si el acelerómetro detecta un toque simple, se detendrá la adquisición y se retomará cuando reciba otro toque. Si detecta un toque doble, se parará la adquisición y se volverá al menú principal.

1. Se lee el registro de interrupciones para borrar los eventos que pudiesen haber aparecido antes del inicio de la función.

2. Se imprime por pantalla el encabezado [Tiempo(s)] [ax(g)] [ay(g)] [az(g)] y se inicializa el contador.

3. Ahora en bucle y por orden de prioridad, se verifica si se ha activado la interrupción por doble toque, la interrupción por toque simple o la interrupción por medición disponible.

4. Si no se detecta ningún toque, se imprime la aceleración de manera normal.

Nótese que se añade una función de retardo para evitar que se lea el registro de interrupciones demasiado rápido.

Un ejemplo de los resultados por el terminal serie se da en la figura 4.15.

```
void ADXL345_DisplayAccelLoop(void)
{
  u8 INT_SOURCE;
  int Stop = 0;

    if(isRunning == 1)
    {

      xil_printf("\n\rSi está habilitado, un toque simple pausa la adquisición de datos. ");
      xil_printf("Un doble toque devuelve al menú principal.\n\r");

      ADXL345_ReadReg(ADXL345_INT_SOURCE);  // Limpia el registro de interrupciones

      xil_printf("\n\r[Tiempo(s)] [ax(g)] [ay(g)] [az(g)]\n\n\r");
      InitializeTimer(&TimerCounterInst, TMRCTR_INTERRUPT_ID, &IntcInstance); // Inicializa el contador
      while(1)
      {
        // Espera para evitar que se verifique demasiado rápido si hay nuevos datos disponibles
        WaitFor(1000000);
        INT_SOURCE = ADXL345_ReadReg(ADXL345_INT_SOURCE);
        if(INT_SOURCE & 0x20)   // Verifica si ha habido una interrupción por doble toque
        {
          break;
        }
        else if(INT_SOURCE & 0x40)  // Verifica si ha habido una interrupción por toque simple
        {
          Stop = ~Stop;       // Pausa o retoma la adquisición
        }
        else if ((INT_SOURCE & 0x80) && (Stop == 0))  // Verifica si hay datos de aceleración disponibles
        {
          WaitFor(1000000);
          GetTime(&TimerCounterInst, TMRCTR_INTERRUPT_ID, &IntcInstance);  // Imprime el tiempo
          ADXL345_DisplayAllAxes(); // Imprime la aceleración
        }
      }
    }
  else if(isRunning == 0)
  {
    xil_printf("\n\r!!! Mediciones no habilitadas. Por favor habilítalas antes de realizar la acción !!!\n\r");
  }

  ClockRead = 0;
  ADXL345_ReadReg(ADXL345_INT_SOURCE);  // Limpia el registro de interrupciones
  ADXL345_DisplayMenu();
}
```

Código 4.18: Función ADXL345_DisplayAccelLoop

```
[Tiempo(s)] [ax(g)] [ay(g)] [az(g)]

[t = 0.260] [X = -0.070] [Y = +0.000] [Z = +1.006]
[t = 0.559] [X = -0.078] [Y = +0.003] [Z = +0.978]
[t = 0.858] [X = -0.066] [Y = +0.003] [Z = +1.010]
[t = 1.157] [X = -0.070] [Y = +0.000] [Z = +0.994]
[t = 1.457] [X = -0.066] [Y = +0.007] [Z = +0.990]
[t = 1.756] [X = -0.074] [Y = -0.003] [Z = +0.994]
[t = 2.055] [X = -0.085] [Y = +0.019] [Z = +0.986]
[t = 2.354] [X = -0.081] [Y = +0.058] [Z = +0.932]
```

Figura 4.15: Ejecución de la función ADXL345_DisplayAccelLoop.

5. MicroSD

Aunque la idea principal era usar la ranura para MicroSD presente en la EVAL-ADXL345Z-DB, con la finalidad de evitar problemas de ruido derivados del uso de cables largos y minimizar potenciales fallos durante la fase de prueba, se optó por replicar el sistema MicroBlaze en el módulo FPGA NEXYS A7-100T de Digilent [25] (Figura 5.1). La ventaja que ofrece esta placa es que incluye una ranura para tarjeta MicroSD, por lo que evitamos las conexiones externas.

La MicroSD está conectada como representa la figura 5.2. Al usar el modo de comunicación SPI, deberemos conectar CS con DAT3, MOSI con CMD, MISO con DAT0 y SCK con CLK. Además, para alimentar la tarjeta, SD_RESET debe ser conectado a valor lógico bajo.

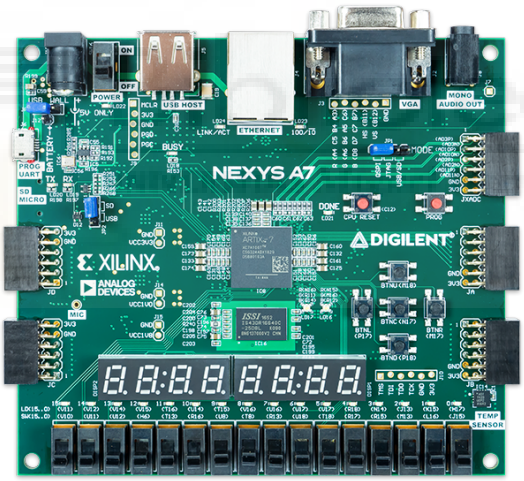

Figura 5.1: Vista frontal del módulo FPGA NEXYS A7.

La ventaja de usar esta placa es que, al estar basada también en una Artix-7, podremos reutilizar prácticamente entero todo el diagrama de bloques anterior. Simplemente hará falta cambiar el archivo de placa [26] y el controlador de memoria externa, ya que ésta usa una memoria DDR2 de 128 MB y no una HyperRAM.

Sin embargo, la idea final es seguir utilizando la TE0725, ya que la NEXYS A7 es bastante grande y tiene un consumo energético mayor. Esta FPGA está pensada como plataforma para el ámbito educativo e iniciación en el ámbito del diseño digital.

Debido a esto, incorpora muchos periféricos que no se usan y que encarecen el coste con respecto al uso de la TE0725.

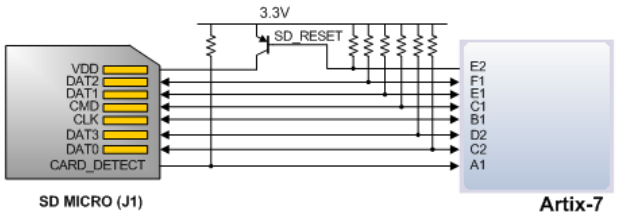

Figura 5.2: Conexión entre la FPGA Artix-7 en la NEXYS A7 y la tarjeta MicroSD.

5.1. Sistema embebido NEXYS-A7

El sistema embebido es igual al de la TE0725, pero se sustituye el controlador de memoria HyperBus por el bloque *'Memory Interface Generator'*. Éste forma parte de las IPs proporcionadas por Vivado, por lo que no hay necesidad de recurrir a una IP externa.

La función principal del *'Memory Interface Generator'* es la de generar un controlador de memoria basándose en el archivo de placa. En concreto, creará un controlador para memoria DDR2 SDRAM. El bloque se configura automáticamente por lo que no hay que preocuparse por definir los parámetros.

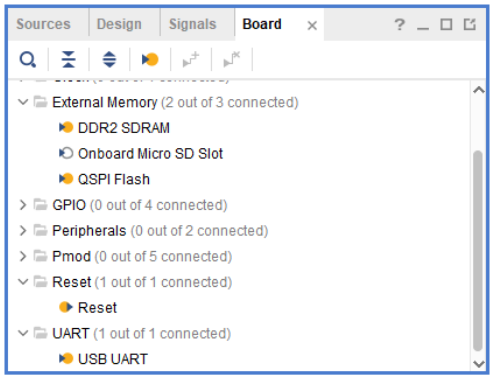

Figura 5.3: Asignación automática de pines en la placa NEXYS A7.

Sí será necesario cambiar las asignaciones automáticas y añadir las específicas de

esta placa. La figura 5.3 muestra las nuevas asignaciones. El diagrama de bloques se puede consultar en el anexo D.

Por último, se añade un nuevo archivo de *constraints* (anexo E). Los pines que se conecten al ADXL345 saldrán por los conectores Pmod del borde de la placa y la IP de SPI para la MicroSD se conectará directamente a la ranura de la Nexys A7.

El proceso a continuación es igual que en la placa TE0725. Una vez generado el bitstream, se exporta el hardware y se prueba en Vitis. Todas las pruebas y códigos explicados anteriormente funcionan también en este dispositivo.

5.2. Problemas con el controlador SPI

Para usar la tarjeta MicroSD, se añade la biblioteca 'xilffs' mediante las opciones de Vitis. Ésta sirve para poder manejar sistemas de archivos FAT, como es el caso de las tarjetas SD.

FAT (*File Allocation Table*) [27] es un sistema de archivos que los organiza por directorios en una estructura jerárquica. Utiliza una tabla de asignación que contiene entradas que indican qué unidades básicas de almacenamiento están ocupadas por cada archivo y dónde se encuentran en el dispositivo de almacenamiento.

Existen varias versiones de FAT, diferenciadas por el tamaño máximo de partición y la cantidad de unidades básicas de almacenamiento que pueden manejar. En nuestro caso se usa una tarjeta formateada en FAT32. Su peculiaridad principal respecto a otros sistemas de archivos como FAT16 es que usa una tabla de asignación de archivos de 32 bits (Figura 5.4). De todas formas, la biblioteca se encarga de detectar el tipo de sistema FAT mediante las funciones de inicialización de la tarjeta MicroSD. Esta biblioteca consiste principalmente en dos partes:

- La primera representa al sistema de archivos y tiene funciones específicas de cómo se organizan los datos.
- La segunda parte es la capa de unión. Esta capa de software actúa como interfaz entre los drivers del periférico controlador y el sistema de archivos.

El problema principal es que la capa de unión está pensada para un controlador para SD que no usa el modo SPI, sino SDIO, y no está disponible de forma gratuita en el repositorio de Vivado. Sería necesario cambiar todas las funciones para poder usar nuestro controlador genérico de SPI. Además, la versión de Vitis que se está utilizando presenta un error de directorio con la carpeta de la biblioteca.

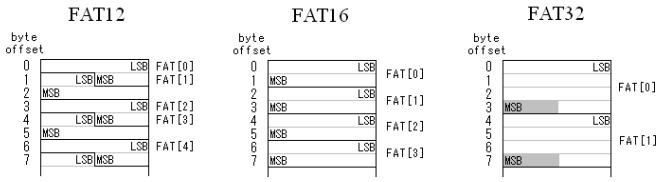

Figura 5.4: Tabla de asignación de archivos para FAT12, FAT16 y FAT32.

Se consideró la opción de migrar al nuevo Vitis, pero las desventajas superaban a las ventajas, ya que presentaba otros errores distintos, como la falta de ciertas bibliotecas de Xilinx que eran necesarias para la aplicación previa. La opción más sencilla era la de buscar otra IP.

5.3. Nuevos controladores para la memoria MicroSD

La primera idea fue la de usar la IP del controlador para SD pensado para un dispositivo Pmod de Digilent (Figura 5.5) [28]. El modo de funcionamiento es SPI por lo que podría seguir siendo usado en la aplicación final. Además, no está basado en la biblioteca 'xilffs' pero utiliza una muy similar.

Por lo general, los conectores Pmod tienen 2 pines de tierra, dos de alimentación y 8 de datos. Al disponer de pines extra, deben ser conectados a la MicroSD aunque no se usen. La tabla 5.1 muestra la asignación de pines entre la IP y la ranura para MicroSD de la NEXYS A7.

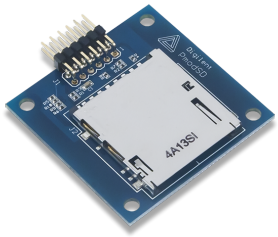

Figura 5.5: Pmod para SD de Digilent.

Sin embargo, a la hora de probar el *software* de ejemplo proporcionado, no se conseguía inicializar la tarjeta. Esto puede ser por varios factores, como por ejemplo, una versión no compatible de Vivado.

IP PMOD SD	**MicroSD NEXYS A7**
CS	DAT3
MOSI	CMD
MISO	DAT0
SCK	CLK
DAT1	DAT1
DAT2	DAT2
CD	CARD_DETECT
WP	-
-	SD_RESET

Tabla 5.1: Conexiones entre la IP Pmod de SD y la tarjeta MicroSD de la NEXYS A7.

La segunda opción fue la de usar una IP de Github que consiste en un controlador de SD mediante SPI [29]. Aunque no tenía muchas valoraciones positivas, se decidió probar ya que si funcionaba resultaba ser la opción más sencilla.

Tiene como pines externos los básicos del protocolo SPI y requiere usar la biblioteca genérica para FAT de ChaN [30] con la capa de unión que se proporciona.

Aunque esta IP aparentemente conseguía inicializar el sistema, no se consiguió crear un archivo.

La tercera opción fue probar con las IPs de la página de GitHub de Dan Gisselquist [31]. Incluye dos distintas:

- Una con protocolo SPI que usa el tipo de conexión bus Wishbone, por lo que no es compatible con AXI.

- Otra opción que funciona mediante protocolo SDIO. Aunque no es el modo que necesitamos, soporta tanto AXI como Wishbone, por lo que se decide probar.

El problema es que, a la hora de empaquetar la IP usando los archivos de descripción de hardware, aparecen más entradas y salidas de las que requiere una tarjeta SD. Al estar las descripciones escritas en Verilog, no poseo los suficientes conocimientos como para investigar el fallo.

Finalmente se opta por abandonar el objetivo.

6. Programación mediante memoria Flash

Tanto la TE0725 como la NEXYS A7 cuentan con una memoria Flash que se programa mediante Quad SPI, una configuración de SPI que permite transferir 4 bits por ciclo de reloj. Al ser no volátil, la principal función de la memoria es almacenar el *bitstream* de configuración y el software que se ejecuta en el procesador embebido. Al encender la placa, estos se cargan en la FPGA.

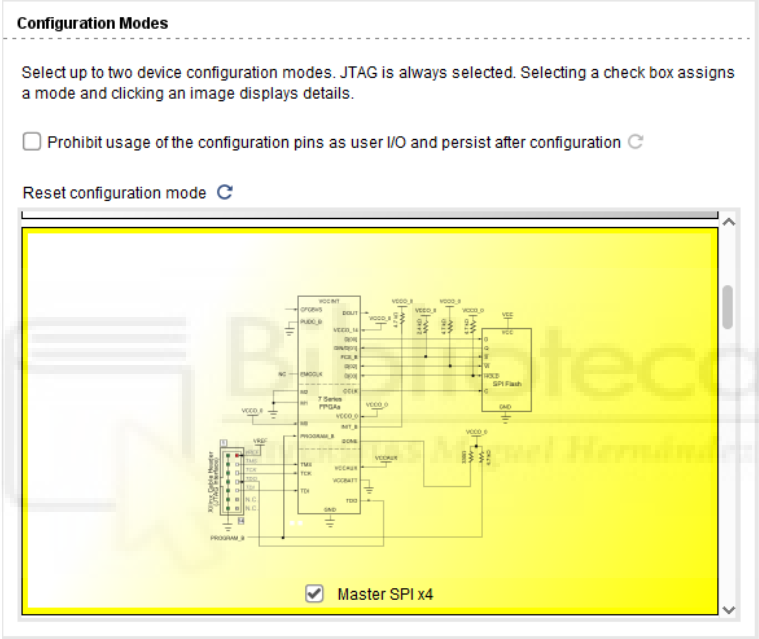

Figura 6.1: Método de programación 'Master SPI x4' en el menú *'Edit Device Properties'*.

En este caso, se explicará el procedimiento específico para cargar los archivos en la memoria Flash de la NEXYS A7, si bien es similar para el módulo TE0725.

Para que este funcionamiento sea posible, primero se deben configurar los proyectos tanto de Vivado como de Vitis. Se siguen dos tutoriales [32] [33].

En Vivado, se abre el diseño sintetizado, donde se pueden agregar nuevos métodos de programación. Por defecto, el proyecto está configurado para usar el modo JTAG. En la pestaña 'Edit Device Properties', se añade el nuevo método, Master SPI x4 (Figura 6.1). Además, se incrementa la velocidad de configuración a 33MHz y se

habilita la compresión del bitstream; esto acelerará los procesos de programación futuros. Luego, se genera el bitstream y se exporta el hardware.

Para los pasos siguientes, se reinstala Vitis Classic versión 2024.1, ya que la versión 2023.2 presenta un fallo que impide la programación de la memoria Flash.

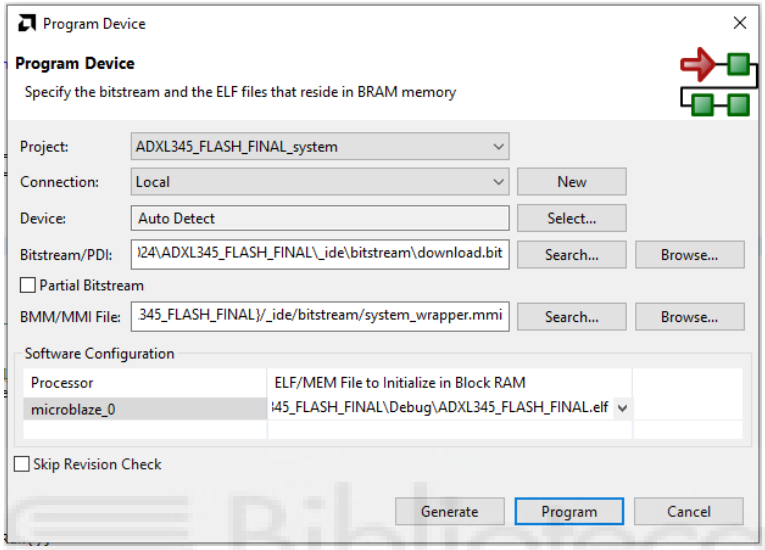

Figura 6.2: Ventana *'Program Device'* en Vitis.

Se crea una nueva aplicación utilizando el proyecto anterior del ADXL345. A continuación, en la ventana *'Program Device'* (Figura 6.2), se seleccionan los archivos necesarios:

- El *bitstream* (renombrado como 'download.bit').
- El archivo '.mmi'. Contiene la información del mapa de memoria.
- El archivo '.elf'. Entre otras cosas, contiene el código de la aplicación compilada, así como la configuración para cargar y ejecutar el programa en la memoria del MicroBlaze.

Se programa el dispositivo en modo JTAG con esta configuración.

Una vez programada la FPGA, se abre la ventana *'Program Flash Memory'*. Nuevamente, se selecciona el archivo de bitstream y se elige el dispositivo Flash específico

del menú desplegable (Figura 6.3). Este proceso incluye el borrado y la programación de los contenidos de la memoria Flash. Se realiza gracias al periférico 'AXI Quad SPI' añadido anteriormente en el diagrama de bloques.

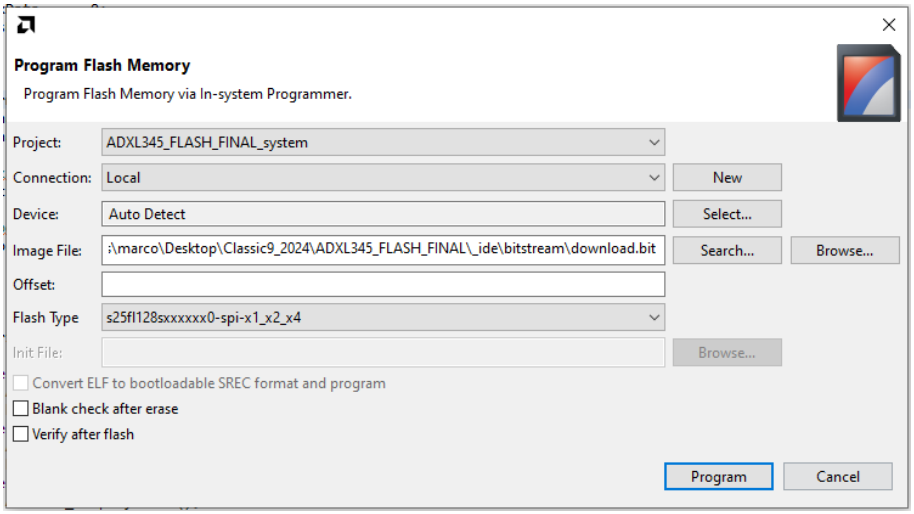

Figura 6.3: Ventana *'Program Flash Memory'* en Vitis.

Por último, se interrumpe la alimentación. A partir de este momento, siempre y cuando el jumper JP1 de la figura 6.4 esté colocado en la posición 'SPI Flash', cuando se vuelva a alimentar el dispositivo, éste se programará directamente desde la memoria Flash.

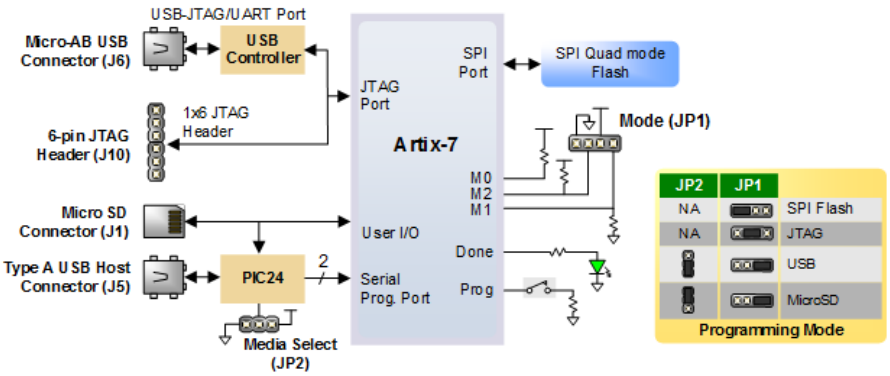

Figura 6.4: Modos de programación en la NEXYS A7.

7. Conclusión y líneas futuras

Aunque no se ha conseguido implementar la parte del sistema de almacenamiento, la adquisición de datos en tiempo real muestra resultados satisfactorios. El único problema detectado se relaciona los cables que conectan las líneas SDA y SCL del ADXL345 con la FPGA. En esta configuración, donde los cables son relativamente largos, la proximidad entre las líneas de la comunicación serie I^2C provoca errores debido a interferencias electromagnéticas y diafonía. Pese a que esto se corrige separando los cables, también podría hacerse mediante el uso de cables blindados o diseñando una PCB que se pueda insertar directamente en el módulo TE0725.

Si bien, sería interesante llegar a implementar la funcionalidad que ofrece la memoria MicroSD, esto no es crítico, ya que los programas de terminal serie como MobaXterm permiten guardar automáticamente los resultados en un documento de texto. Sin embargo, si no se usa un ordenador, esta opción no está disponible.

Como continuación del proyecto y líneas futuras, se podría valorar la idea de añadir un módulo de comunicación inalámbrica y enviar los datos a un servidor web. Esto eliminaría la necesidad de una tarjeta MicroSD o incluso del terminal serie, proporcionando acceso a los valores de aceleración en tiempo real sin requerir la conexión física de la FPGA.

En resumen, aunque no se hayan cumplido todos los objetivos, se logró implementar la parte principal de manera satisfactoria y existen alternativas para abordar las necesidades pendientes.

ANEXOS

A. *Constraints* TE0725

```
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 50 [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
set_property BITSTREAM.CONFIG.SPI_32BIT_ADDR YES [current_design]
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
set_property BITSTREAM.CONFIG.M1PIN PULLNONE [current_design]
set_property BITSTREAM.CONFIG.M2PIN PULLNONE [current_design]
set_property BITSTREAM.CONFIG.M0PIN PULLNONE [current_design]
set_property BITSTREAM.CONFIG.USR_ACCESS TIMESTAMP [current_design]
```

Código A.1: _i_bitgen_common.xdc

```
#UART ACCELEROMETER#

set_property -dict { PACKAGE_PIN K2   IOSTANDARD LVCMOS33 } [get_ports {
          ↪ UART_ACCEL_rxd }]; #Sch=J1 P4 RX
set_property -dict { PACKAGE_PIN K1   IOSTANDARD LVCMOS33 } [get_ports {
          ↪ UART_ACCEL_txd }]; #Sch=J1 P3 TX


#SPI MicroSD#

set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {
          ↪ spi_MicroSD_MOSI }]; #Sch=J2 P21 MOSI
set_property -dict { PACKAGE_PIN U3   IOSTANDARD LVCMOS33 } [get_ports {
          ↪ spi_MicroSD_SS }]; #Sch=J2 P22 SS
set_property -dict { PACKAGE_PIN N5   IOSTANDARD LVCMOS33 } [get_ports {
          ↪ spi_MicroSD_sck }]; #Sch=J2 P27 SCK
set_property -dict { PACKAGE_PIN P5   IOSTANDARD LVCMOS33 } [get_ports {
          ↪ spi_MicroSD_MISO }]; #Sch=J2 P28 MISO


#AXI I2C ACL#

set_property -dict { PACKAGE_PIN M1   IOSTANDARD LVCMOS33  PULLUP true} [get_ports
          ↪ { IIC_ACL_scl_io }]; #Sch=J2 P43 SCL
set_property -dict { PACKAGE_PIN L1   IOSTANDARD LVCMOS33  PULLUP true} [get_ports
          ↪ { IIC_ACL_sda_io }]; #Sch=J2 P44 SDA
```

Código A.2: accel.xdc

```
set_property PACKAGE_PIN A13 [get_ports HB_CLK0_0]
set_property PACKAGE_PIN A14 [get_ports HB_CLK0n_0]

set_property PACKAGE_PIN E17 [get_ports {HB_dq_0[0]}]
set_property PACKAGE_PIN B17 [get_ports {HB_dq_0[1]}]
set_property PACKAGE_PIN F18 [get_ports {HB_dq_0[2]}]
set_property PACKAGE_PIN F16 [get_ports {HB_dq_0[3]}]
set_property PACKAGE_PIN G17 [get_ports {HB_dq_0[4]}]
set_property PACKAGE_PIN D18 [get_ports {HB_dq_0[5]}]
set_property PACKAGE_PIN B18 [get_ports {HB_dq_0[6]}]
set_property PACKAGE_PIN A16 [get_ports {HB_dq_0[7]}]
```

```
12 set_property PACKAGE_PIN E18 [get_ports HB_RWDS_0]
13 set_property PACKAGE_PIN D17 [get_ports HB_CS1n_0]
14 set_property PACKAGE_PIN J17 [get_ports HB_RSTn_0]
15 #set_property PACKAGE_PIN A18 [get_ports HB_CS0n_0 ]
16 #set_property PACKAGE_PIN J18 [get_ports HB_INTn_0 ]
17 #set_property PACKAGE_PIN C17 [get_ports HB_RSTOn_0]
18
19 #
20 # FPGA Pin Voltage assignment
21 #
22 set_property IOSTANDARD LVCMOS18 [get_ports HB_CLK0_0]
23 set_property IOSTANDARD LVCMOS18 [get_ports HB_CLK0n_0]
24 set_property IOSTANDARD LVCMOS18 [get_ports {HB_dq_0[*]}]
25 set_property IOSTANDARD LVCMOS18 [get_ports HB_CS1n_0]
26 set_property IOSTANDARD LVCMOS18 [get_ports HB_RSTn_0]
27 set_property IOSTANDARD LVCMOS18 [get_ports HB_RWDS_0]
28
29 #set_property IOSTANDARD LVCMOS18 [get_ports HB_CS0n_0]
30 #set_property IOSTANDARD LVCMOS18 [get_ports HB_INTn_0]
31 #set_property IOSTANDARD LVCMOS18 [get_ports HB_RSTOn_0]
32
33 #set_property PULLUP true [get_ports HB_RSTOn_0]
34 #set_property PULLUP true [get_ports HB_INTn_0]
35
36 #
37 #Hyperbus Clock - change according to clk pin on PLL
38 #
39 create_generated_clock -name clk_0   -source [get_pins
          ↪ msys_i/clk_wiz_0/inst/mmcm_adv_inst/CLKIN1] -master_clock sys_clock
          ↪ [get_pins msys_i/clk_wiz_0/inst/mmcm_adv_inst/CLKOUT0]
40 create_generated_clock -name clk_90  -source [get_pins
          ↪ msys_i/clk_wiz_0/inst/mmcm_adv_inst/CLKIN1] -master_clock sys_clock
          ↪ [get_pins msys_i/clk_wiz_0/inst/mmcm_adv_inst/CLKOUT1]
41 create_generated_clock -name clk_180 -source [get_pins
          ↪ msys_i/clk_wiz_0/inst/mmcm_adv_inst/CLKIN1] -master_clock sys_clock
          ↪ [get_pins msys_i/clk_wiz_0/inst/mmcm_adv_inst/CLKOUT2]
42
43 #
44 #100Mhz clock freqeuncy - change accordingly
45 #
46 set hbus_freq_ns   10
47
48 set dqs_in_min_dly -0.5
49 set dqs_in_max_dly  0.5
50
51 set HB_dq_ports    [get_ports HB_dq_0[*]]
52
53 #
54 #Create RDS clock and RDS virtual clock
55 #
56 create_clock -period $hbus_freq_ns -name rwds_clk      [get_ports HB_RWDS_0]
57 create_clock -period $hbus_freq_ns -name virt_rwds_clk
58
59 #
60 #Input Delay Constraint -  HB_RWDS-HB_DQ
61 #
62 set_input_delay -clock [get_clocks virt_rwds_clk]             -max
          ↪ ${dqs_in_max_dly} ${HB_dq_ports}
63 set_input_delay -clock [get_clocks virt_rwds_clk] -clock_fall -max
          ↪ ${dqs_in_max_dly} ${HB_dq_ports} -add_delay
64
65 set_input_delay -clock [get_clocks virt_rwds_clk]             -min
          ↪ ${dqs_in_min_dly} ${HB_dq_ports} -add_delay
66 set_input_delay -clock [get_clocks virt_rwds_clk] -clock_fall -min
          ↪ ${dqs_in_min_dly} ${HB_dq_ports} -add_delay
67
68 set_multicycle_path -setup -end -rise_from [get_clocks virt_rwds_clk] -rise_to
          ↪ [get_clocks rwds_clk] 0
69 set_multicycle_path -setup -end -fall_from [get_clocks virt_rwds_clk] -fall_to
          ↪ [get_clocks rwds_clk] 0
```

```
70 set_false_path  -fall_from [get_clocks virt_rwds_clk] -rise_to [get_clocks
           ↪ rwds_clk] -setup
71 set_false_path  -rise_from [get_clocks virt_rwds_clk] -fall_to [get_clocks
           ↪ rwds_clk] -setup
72 set_false_path  -fall_from [get_clocks virt_rwds_clk] -fall_to [get_clocks
           ↪ rwds_clk] -hold
73 set_false_path  -rise_from [get_clocks virt_rwds_clk] -rise_to [get_clocks
           ↪ rwds_clk] -hold
74
75 set_false_path -from [get_clocks clk_0] -to [get_clocks rwds_clk]
76 set_false_path -from [get_clocks rwds_clk] -to [get_clocks clk_0]
77
78 #
79 #Output Delay Constraint -  HB_CLK0-HB_DQ
80 #
81
82 create_generated_clock -name HB_CLK0_0 -source [get_pins
           ↪ */*/*/U_IO/U_CLK0/dq_idx_[0].ODDR_inst/C] -multiply_by 1 -invert
           ↪ [get_ports HB_CLK0_0]
83
84 set_output_delay -clock [get_clocks HB_CLK0_0] -min -1.000 ${HB_dq_ports}
85 set_output_delay -clock [get_clocks HB_CLK0_0] -max  1.000 ${HB_dq_ports}
86 set_output_delay -clock [get_clocks HB_CLK0_0] -min -1.000 ${HB_dq_ports}
           ↪ -clock_fall -add_delay
87 set_output_delay -clock [get_clocks HB_CLK0_0] -max  1.000 ${HB_dq_ports}
           ↪ -clock_fall -add_delay
88
89
90 set_false_path -from [get_pins */*/*/U_HBC/*/dq_io_tri_reg/C] -to ${HB_dq_ports}
91
92 set_false_path -from * -to [get_pins */*/inst/*/i_iavs0_270_rstn_1_reg/CLR]
93 set_false_path -from * -to [get_pins */*/inst/*/i_iavs0_270_rstn_2_reg/CLR]
94 set_false_path -from * -to [get_pins */*/inst/*/i_iavs0_270_rstn_3_reg/CLR]
95
96
97 set_false_path -from [get_clocks rwds_clk] -to [get_clocks -of_objects [get_pins
           ↪ msys_i/clk_wiz_1/inst/mmcm_adv_inst/CLKOUT0]]
98 set_false_path -from [get_clocks virt_rwds_clk] -to [get_clocks rwds_clk]
```

Código A.3: _i_hyperram.xdc

B. Diagrama de bloques TE0725

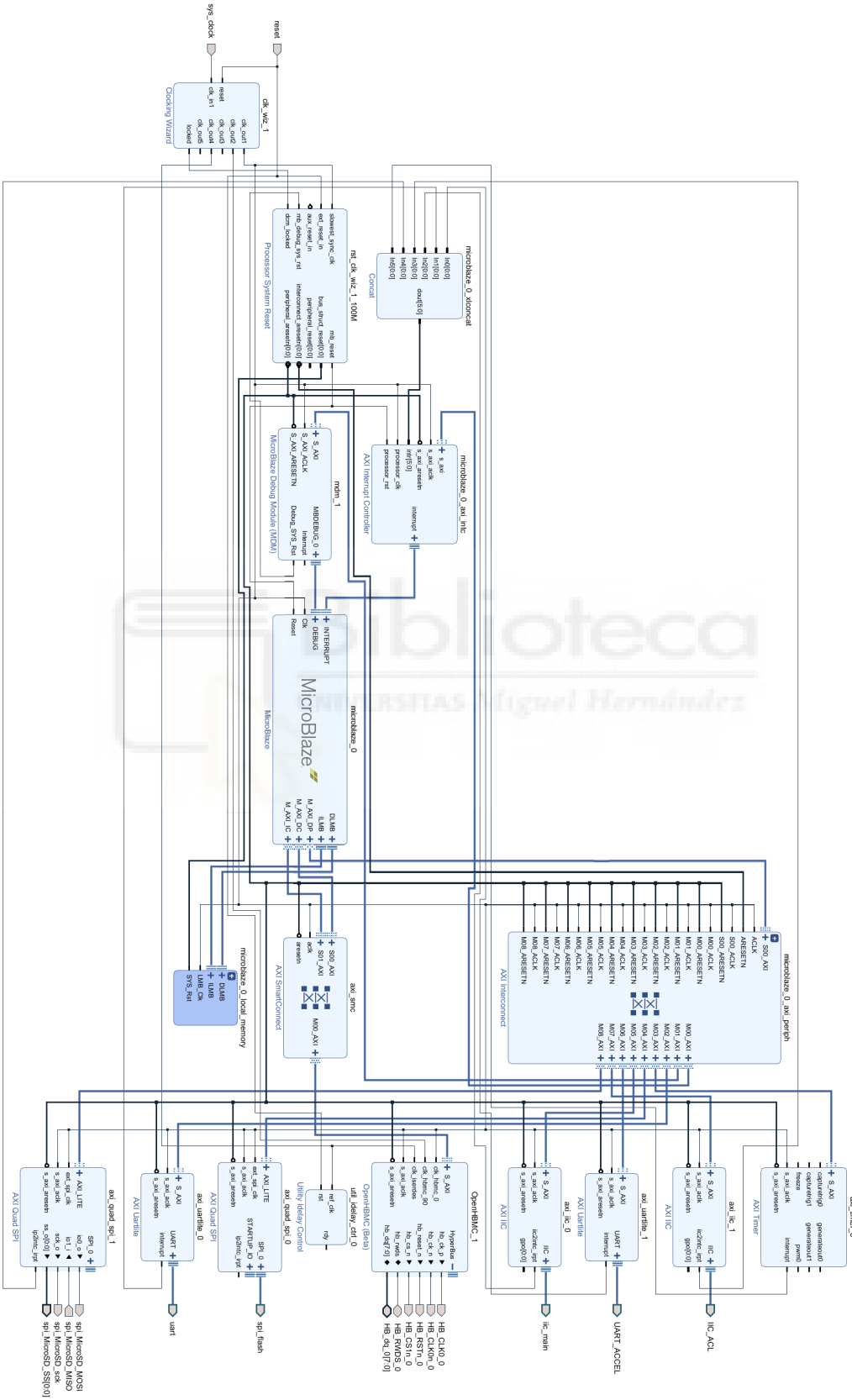

Figura B.1: Diagrama de bloques de la TE0725.

C. Código completo

```c
#include <stdio.h>
#include "xparameters.h"
#include "xil_io.h"
#include "xil_cache.h"
#include "adxl345.h"
#include "xiic.h"
#include "platform.h"
#include "xil_printf.h"
#include "xil_exception.h"
#include "xintc.h"
#include "xtmrctr.h"


volatile char rxData  = 0;
volatile char isRunning = 0;

int main()
{
  init_platform();
  Xil_ICacheEnable();
  Xil_DCacheEnable();

  // Inicializa el ADXL345
  ADXL345_Init();

  // Imprime por pantalla el menú principal
  ADXL345_DisplayMainMenu();

  while(1)
  {

    switch(rxData)
    {
      case 'e':
        ADXL345_Run();
        break;
      case 'd':
        ADXL345_Stop();
        break;
      case 'a':
        ADXL345_DisplayAccel();
        break;
      case 's':
        ADXL345_SelectRange();
        break;
      case 'r':
        ADXL345_AcquisitionRate();
        break;
      case 't':
        ADXL345_SelectTap();
        break;
      case 'o':
        ADXL345_DisplayCalibrationValue();
        break;
      case 'h':
        ADXL345_DisplaySelftestDelta();
        break;
      case 'l':
        ADXL345_DisplayAccelLoop();
        break;
      case 0:
        break;
      default:
        xil_printf("\n> Error! Por favor, selecciona una de las opciones enumeradas a continuación\r");
        ADXL345_DisplayMenu();
        break;
    }
  }

  Xil_DCacheDisable();
  Xil_ICacheDisable();

  return 0;
}
```

Código C.4: main.c

```
#ifndef ADXL345_H_
#define ADXL345_H_

#include "xparameters.h"
#include "xil_cache.h"
#include "xil_io.h"
//#include "xuartlite.h"
#include "xiic.h"
#include "xtmrctr.h"
#include "xintc.h"

//#define UARTLITE_DEVICE_ID        XPAR_UARTLITE_1_DEVICE_ID
#define IIC_DEVICE_ID       XPAR_IIC_0_DEVICE_ID
#define TMRCTR_DEVICE_ID      XPAR_TMRCTR_0_DEVICE_ID
#define INTC_DEVICE_ID        XPAR_INTC_0_DEVICE_ID
//#define UARTLITE_IRPT_INTR      XPAR_INTC_0_UARTLITE_1_VEC_ID
#define IIC_INTR_ID         XPAR_INTC_0_IIC_0_VEC_ID
#define TMRCTR_INTERRUPT_ID     XPAR_INTC_0_TMRCTR_0_VEC_ID


#define TIMER_CNTR_0   0
#define TIMER_CNTR_1   1
#define RESET_VALUE_CNTR_0   0x00000000
#define RESET_VALUE_CNTR_1   0x00000000

#define INTC      XIntc
#define INTC_HANDLER  XIntc_InterruptHandler
/***************************************************************************/
/* ADXL345                                                                 */
/***************************************************************************/
#define ADXL345_ADDRESS       0x53

#define ADXL345_RnW         7
#define ADXL345_MB          6
// ADXL345 Registers
#define ADXL345_DEVID       0x00
#define ADXL345_THRESH_TAP      0x1D
#define ADXL345_OFSX        0x1E
#define ADXL345_OFSY        0x1F
#define ADXL345_OFSZ        0x20
#define ADXL345_DUR         0x21
#define ADXL345_Latent        0x22
#define ADXL345_Window        0x23
#define ADXL345_THRESH_ACT      0x24
#define ADXL345_THRESH_INACT      0x25
#define ADXL345_TIME_INACT      0x26
#define ADXL345_ACT_INACT_CTL   0x27
#define ADXL345_THRESH_FF     0x28
#define ADXL345_TIME_FF       0x29
#define ADXL345_TAP_AXES      0x2A
#define ADXL345_ACT_TAP_STATUS    0x2B
#define ADXL345_BW_RATE       0x2C
#define ADXL345_POWER_CTL     0x2D
#define ADXL345_INT_ENABLE      0x2E
#define ADXL345_INT_MAP       0x2F
#define ADXL345_INT_SOURCE      0x30
#define ADXL345_DATA_FORMAT     0x31
#define ADXL345_DATA_X0       0x32
#define ADXL345_DATA_X1       0x33
#define ADXL345_DATA_Y0       0x34
#define ADXL345_DATA_Y1       0x35
#define ADXL345_DATA_Z0       0x36
#define ADXL345_DATA_Z1       0x37
#define ADXL345_FIFO_CTL      0x38
#define ADXL345_FIFO_STATUS     0x39

// Register specific bits
// Device ID Register 0x00 - Read value should be 0xE5
// Tap Threshold Register 0x1D - Holds Threshold values for tap interrupts
// Offset X Register 0x1E |
// Offset Y Register 0x1F |- 15.6 mg/LSB, added to the acceleration data
// Offset Z Register 0x20 |
// Tap Duration Register 0x21 - 625us/LSB. Holds time that an event must be above THRESH_TAP to qualify as
                ↪ event
// Latent Register 0x22 - Time from detection of tap to time window. 1.25ms/LSB
// Window Register 0x23 - Time when a second tap can be detected. 1.25ms/LSB
// Activity Threshold Register 0x24 - Threshold value for detecting activity. 62.5mg/LSB
// Inactivity Threshold Register 0x25 - Threshold value for detecting inactivity. 62.5mg/LSB
// Inactivity Time Register 0x26 - Time for which acceleration must be less than THRESH_INACT. 1s/LSB
// Activity/Inactivity Control Register 0x27
#define ADXL345_ACT_ac_dc     7
#define ADXL345_ACT_X_en      6
#define ADXL345_ACT_Y_en      5
#define ADXL345_ACT_Z_en      4
#define ADXL345_INACT_ac_dc     3
#define ADXL345_INACT_X_en      2
#define ADXL345_INACT_Y_en      1
#define ADXL345_INACT_Z_en      0
// Threshold Free Fall Register 0x28 - Value for free fall detection. 62.5mg/LSB 0x05 to 0x09 are recommended
// Free Fall Time Register 0x29 - Time for which axes value must be lower than THRESH_FF to generate fall
                ↪ interrupt. 5ms/LSB. 0x14 to 0x46 recommended
// Axes Control For Single/Double Tap Register 0x2A
#define ADXL345_Suppress      3
#define ADXL345_TAP_X_en      2
#define ADXL345_TAP_Y_en      1
#define ADXL345_TAP_Z_en      0
// Source Of Single/Double Tap Register 0x2B
#define ADXL345_ACT_X_Source    6
#define ADXL345_ACT_Y_Source    5
#define ADXL345_ACT_Z_Source    4
```

```
99  #define ADXL345_ASleep        3
100 #define ADXL345_TAP_X_Source    2
101 #define ADXL345_TAP_Y_Source    1
102 #define ADXL345_TAP_Z_Source    0
103 // Data Rate and Power Mode Control Register 0x2C
104 #define ADXL345_LOW_POWER     4
105 #define ADXL345_RATE        0
106 // Power Savings Features Control Register 0x2D
107 #define ADXL345_Link        5
108 #define ADXL345_AUTO_SLEEP      4
109 #define ADXL345_Measure       3
110 #define ADXL345_Sleep       2
111 #define ADXL345_Wakeup        0
112 // Interrupt Enable Control Register 0x2E
113 // Interrupt Mapping Control Register 0x2F
114 // Interrupt Source Register 0x30
115 #define ADXL345_DATA_READY      7
116 #define ADXL345_SINGLE_TAP      6
117 #define ADXL345_DOUBLE_TAP      5
118 #define ADXL345_Activity      4
119 #define ADXL345_Inactivity      3
120 #define ADXL345_FREE_FALL     2
121 #define ADXL345_Watermark     1
122 #define ADXL345_Overrun       0
123 // Data Format Control Register 0x31
124 #define ADXL345_SELF_TEST     7
125 #define ADXL345_SPI         6
126 #define ADXL345_INT_INVERT      5
127 #define ADXL345_FULL_RES      3
128 #define ADXL345_Justify       2
129 #define ADXL345_Range       0
130 // X-Axis Data0 Register 0x32
131 // X-Axis Data1 Register 0x33
132 // Y-Axis Data0 Register 0x34
133 // Y-Axis Data1 Register 0x35
134 // Z-Axis Data0 Register 0x36
135 // Z-Axis Data1 Register 0x37
136 // FIFO Control Register 0x38
137 #define ADXL345_FIFO_MODE     6
138 #define ADXL345_Trigger       5
139 #define ADXL345_Samples       0
140 // FIFO Status Register 0x39
141 #define ADXL345_FIFO_TRIG     7
142 #define ADXL345_Entries       0
143
144 void ADXL345_DisplayMainMenu(void);
145 void ADXL345_DisplayMenu(void);
146 void ADXL345_DisplayRangeMenu(void);
147 void ADXL345_DisplayRateMenu(void);
148 void ADXL345_DisplayTapMenu(void);
149 void ADXL345_WriteReg(unsigned char addr, unsigned char txData);
150 int ADXL345_ReadReg(unsigned char addr);
151 int ADXL345_BurstReadReg(unsigned char addr);
152 void ADXL345_Init(void);
153 void ADXL345_Display_G_Force(char axis);
154 void ADXL345_DisplayAllAxes(void);
155 void ADXL345_ReadAllAxes(void);
156 void ADXL345_Run(void);
157 void ADXL345_Stop(void);
158 void ADXL345_DisplayAccel(void);
159 void ADXL345_SelectRange(void);
160 void ADXL345_AcquisitionRate(void);
161 void ADXL345_SelectTap(void);
162 void ADXL345_DisplayCalibrationValue(void);
163 void ADXL345_DisplaySelftestDelta(void);
164 long int ADXL345_CalculateOffset(char axis);
165 long int ADXL345_CalculateAverage(char axis);
166 void ADXL345_DisplayAccelLoop(void);
167 void WaitFor(int);
168 void GetTime(XTmrCtr *TmrCtrInstancePtr, u16 TmrCtrIntrId, INTC *IntcInstancePtr);
169 void InitializeTimer(XTmrCtr *TmrCtrInstancePtr, u16 TmrCtrIntrId, INTC *IntcInstancePtr);
170
171 #endif /* ADXL345_H_ */
```

Código C.5: adxl345.h

```c
#include <stdio.h>
#include <math.h>
#include "adxl345.h"

extern volatile char rxData;         // Almacena la opción seleccionada
extern volatile char isRunning;      // '1' si el ADXL345 está realizando mediciones. '0' indica modo reposo
static u32 TimerOffset;          // Valor de offset al reiniciar el contador
static int ClockRead = 0;          // '0' si es la primera lectura del contador, '1' las consecutivas
static int FirstTimeInitializingClk = 0; // '0' si no se ha inicializado todavía el contador
u16 BurstRead;

volatile u8 TransmitComplete;   // Flag para verificar la finalización de la transmisión
volatile u8 ReceiveComplete;    // Flag para verificar la finalización de la recepción

static INTC IntcInstance;     // Instancia del controlador de interrupciones
//static XUartLite UartLiteInst;  // Instancia del dispositivo UartLite
XIic IicInstance;         // Instancia del dispositivo IIC
XTmrCtr TimerCounterInst;       // Instancia del contador AXI Timer

static void SendHandler(XIic *InstancePtr);         // Controlador de envíos del dispositivo IIC
static void ReceiveHandler(XIic *InstancePtr);        // Controlador de recepciones del dispositivo IIC
static void StatusHandler(XIic *InstancePtr, int Event);  // Controlador de estado del dispositivo IIC
// Función de inicialización del sistema de interrupciones
static int SetupIntrSystem(INTC *IntcInstancePtr, XIic *IicInstPtr, XTmrCtr *TmrCtrInstancePtr, u16
              ↪ TmrCtrDeviceId, u16 TmrCtrIntrId);

/*****************************************************************************
* @brief  Muestra por panatalla el menú principal.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_DisplayMainMenu(void)
{
  xil_printf("\n\r\n\rPrograma del ADXL345\n\r");
  xil_printf("\n\r");
  xil_printf("Opciones:\n\r");
  xil_printf("    [e] Habilitar Mediciones\n\r");
  xil_printf("    [d] Deshabilitar Mediciones \n\r");
  xil_printf("    [a] Imprimir Aceleración\n\r");
  xil_printf("    [s] Seleccionar Rango de Medida\n\r");
  xil_printf("    [r] Cambiar Tasa de Adquisición\n\r");
  xil_printf("    [t] Seleccionar Interrupciones por Toque\n\r");
  xil_printf("    [o] Calcular el valor de calibración de Offset\n\r");
  xil_printf("    [h] Auto-test\n\r");
  xil_printf("    [l] Imprimir aceleración en bucle\n\r");
  xil_printf("Por favor, selecciona una opción\n\r");

  scanf(" %c", &rxData);
}

/*****************************************************************************
* @brief  Muestra el menú.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_DisplayMenu(void)
{
  xil_printf("\n\rOpciones:\n\r");
  xil_printf("    [e] Habilitar Mediciones\n\r");
  xil_printf("    [d] Deshabilitar Mediciones\n\r");
  xil_printf("    [a] Imprimir Aceleración\n\r");
  xil_printf("    [s] Seleccionar Rango de Medida\n\r");
  xil_printf("    [r] Cambiar Tasa de Adquisición\n\r");
  xil_printf("    [t] Seleccionar Interrupciones por Toque\n\r");
  xil_printf("    [o] Calcular el valor de calibración de Offset\n\r");
  xil_printf("    [h] Auto-test\n\r");
  xil_printf("    [l] Imprimir aceleración en bucle\n\r");
  xil_printf("Por favor, selecciona una opción\n\r");

  scanf(" %c", &rxData);
}

/*****************************************************************************
* @brief  Muestra el menú de rango de medida.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_DisplayRangeMenu(void)
{
  xil_printf("\n\rRango de Medida\n\r");
  xil_printf("Opciones:\n\r");
  xil_printf("    [1] +-2g\n\r");
  xil_printf("    [2] +-4g\n\r");
  xil_printf("    [3] +-8g\n\r");
  xil_printf("    [4] +-16g\n\r");
  xil_printf("Presiona de [1] a [4] para seleccionar el rango deseado\n\r");
}

/*****************************************************************************
* @brief  Muestra el menú de tasa de adquisición.
*
* @param  None.
*
* @return   None.
```

```
100 ******************************************************************************/
101 void ADXL345_DisplayRateMenu(void)
102 {
103   xil_printf("\n\rTasa de Adquisición\n\r");
104   xil_printf("Opciones:\n\r");
105   xil_printf("    [1] 6.25 Hz\n\r");
106   xil_printf("    [2] 12.5 Hz\n\r");
107   xil_printf("    [3] 25 Hz\n\r");
108   xil_printf("    [4] 50 Hz\n\r");
109   xil_printf("    [5] 100 Hz\n\r");
110   xil_printf("    [6] 200 Hz\n\r");
111   xil_printf("    [7] 400 Hz\n\r");
112   xil_printf("    [8] 800 Hz\n\r");
113   xil_printf("    [9] 1600 Hz\n\r");
114   xil_printf("Presiona de [1] a [9] para seleccionar el rango deseado\n\r");
115 }
116
117 /*****************************************************************************
118 * @brief  Muestra el menú de interrupción por toques.
119 *
120 * @param  None.
121 *
122 * @return   None.
123 ******************************************************************************/
124 void ADXL345_DisplayTapMenu(void)
125 {
126   xil_printf("\n\rInterrupciones por Toque\n\r");
127   xil_printf("Opciones:\n\r");
128   xil_printf("    [1] Toque Simple\n\r");
129   xil_printf("    [2] Toque Doble\n\r");
130   xil_printf("    [3] Toques Simple y Doble\n\r");
131   xil_printf("    [4] Toques Deshabilitados\n\r");
132   xil_printf("Presiona de [1] a [4] para seleccionar el rango deseado\n\r");
133 }
134
135 /*****************************************************************************
136 * @brief  Escribe al registro interno del ADXL345.
137 *
138 * @param  RegAdress   - Dirección de registro.
139 *
140 * @param  txData - Datos a escribir.
141 *
142 * @return   None.
143 ******************************************************************************/
144 void ADXL345_WriteReg(u8 RegAddress, u8 txData)
145 {
146
147   u8 WriteBuffer[2] = {RegAddress, txData};
148
149   TransmitComplete = 1;   // Valor por defecto
150   XIic_Start(&IicInstance);           // Inicializa el dispositivo IIC
151   XIic_MasterSend(&IicInstance, WriteBuffer, 2);  // Envía los datos
152
153   //Espera a que finalice la transmisión
154   while ((TransmitComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {
155
156   }
157
158   XIic_Stop(&IicInstance);  // Detiene el dispositivo IIC
159
160 }
161
162 /*****************************************************************************
163 * @brief    Lee el registro interno del ADXL345.
164 *
165 * @param    RegAddress - Dirección de registro.
166 *
167 * @return   rxData - Datos leídos.
168 ******************************************************************************/
169 int ADXL345_ReadReg(u8 RegAddress)
170 {
171
172   u8 WriteBuffer[1] = {RegAddress};
173   u8 ReadBuffer[1] = {0x00};
174
175   TransmitComplete = 1;               // Valor por defecto
176   XIic_Start(&IicInstance);             // Inicializa el dispositivo IIC
177   XIic_MasterSend(&IicInstance, WriteBuffer, 1);  // Envía los datos
178
179   //Espera a que finalice la transmisión
180   while ((TransmitComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {
181
182   }
183
184   XIic_Stop(&IicInstance);  // Detiene el dispositivo IIC
185   ReceiveComplete = 1;    // Valor por defecto
186
187   XIic_Start(&IicInstance);   // Inicializa el dispositivo IIC
188   XIic_MasterRecv(&IicInstance, ReadBuffer, 1); // Recibe los datos
189
190   //Espera a que finalice la recepción
191   while ((ReceiveComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {
192
193   }
194
195   XIic_Stop(&IicInstance);  // Detiene el dispositivo IIC
196   rxData = ReadBuffer[0];
197
198   return(rxData);
199 }
```

```
/*****************************************************************************
* @brief  Lee dos registros internos del ADXL345 consecutivos.
*
* @param  RegAddress - Dirección del primer registro.
*
* @return   rxData - Datos Leídos.
*****************************************************************************/
int ADXL345_BurstReadReg(u8 RegAddress)
{

  u8 WriteBuffer[1] = {RegAddress};
  u8 ReadBuffer[2] = {0x00, 0x00};

  TransmitComplete = 1;   // Valor por defecto
  XIic_Start(&IicInstance);            // Inicializa el dispositivo IIC
  XIic_MasterSend(&IicInstance, WriteBuffer, 1);  // Envía los datos

  //Espera a que finalice la transmisión
  while ((TransmitComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {

  }

  XIic_Stop(&IicInstance);  // Detiene el dispositivo IIC
  ReceiveComplete = 1;    // Valor por defecto

  XIic_Start(&IicInstance);            // Inicializa el dispositivo IIC
  XIic_MasterRecv(&IicInstance, ReadBuffer, 2); // Recibe los datos

  //Espera a que finalice la recepción
  while ((ReceiveComplete) || (XIic_IsIicBusy(&IicInstance) == TRUE)) {

  }

  XIic_Stop(&IicInstance); // Detiene el dispositivo IIC

  // Organiza los datos u8 recibidos en u16
  BurstRead = ((ReadBuffer[1] & 0xFF) << 8) | (ReadBuffer[0] & 0xFF);

  return(BurstRead);
}

/*****************************************************************************
* @brief  Inicializa el ADXL345.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_Init(void)
{
  signed char X_CALIB = 0, Y_CALIB = 0, Z_CALIB = 0;  // Dejar en 0 si el offset del dispositivo no ha sido
      ↪ calibrado

  // Inicializa el Subsistema de Interrupciones y los dispositivos conectados
  SetupIntrSystem(&IntcInstance, &IicInstance, &TimerCounterInst, TMRCTR_DEVICE_ID, TMRCTR_INTERRUPT_ID);
  XIic_SetAddress(&IicInstance, XII_ADDR_TO_SEND_TYPE, ADXL345_ADDRESS);  // Establece la dirección del ADXL345

  ADXL345_WriteReg(ADXL345_BW_RATE, (0x07 << ADXL345_RATE));      // Tasa de Datos de Salida = 12.5 Hz
  ADXL345_WriteReg(ADXL345_DATA_FORMAT, ((1 << ADXL345_FULL_RES)  | // Formato de Datos = +- 16g, Resolución
      ↪ Completa
                   (3 << ADXL345_Range)));
  ADXL345_WriteReg(ADXL345_THRESH_TAP, 0x20);             // Umbral de Toque
  ADXL345_WriteReg(ADXL345_DUR, 0x0D);                // Duración de Toque
  ADXL345_WriteReg(ADXL345_Latent, 0x50);               // Latencia de Toque
  ADXL345_WriteReg(ADXL345_Window, 0xF0);               // Ventana de Toque
  ADXL345_WriteReg(ADXL345_TAP_AXES, ((1 << ADXL345_TAP_X_en)   | // Habilitar Toques en todos los ejes
                   (1 << ADXL345_TAP_Y_en)   |
                   (1 << ADXL345_TAP_Z_en)));
  ADXL345_WriteReg(ADXL345_THRESH_ACT, 0x08);             // Umbral de Actividad
  ADXL345_WriteReg(ADXL345_ACT_INACT_CTL, ((1 << ADXL345_ACT_X_en)| // Ejes de Actividad
                    (1 << ADXL345_ACT_Y_en) |
                    (1 << ADXL345_ACT_Z_en)));
  ADXL345_WriteReg(ADXL345_THRESH_FF, 0x08);              // Umbral de Caída Libre
  ADXL345_WriteReg(ADXL345_TIME_FF, 0x0A);              // Ventana de Caída Libre
  ADXL345_WriteReg(ADXL345_FIFO_CTL, 0x01);
  ADXL345_WriteReg(ADXL345_INT_ENABLE, ((1 << ADXL345_SINGLE_TAP) | // Habilitar Toques Simples y Dobles
                    (1 << ADXL345_DOUBLE_TAP) |
                    (0 << ADXL345_Activity) |
                    (0 << ADXL345_FREE_FALL)  |
                    (0 << ADXL345_DATA_READY)));
  ADXL345_WriteReg(ADXL345_OFSX, X_CALIB);              // Escribir Offset del eje X
  ADXL345_WriteReg(ADXL345_OFSY, Y_CALIB);              // Escribir Offset del eje Y
  ADXL345_WriteReg(ADXL345_OFSZ, Z_CALIB);              // Escribir Offset del eje Z
  ADXL345_ReadAllAxes();
}

/*****************************************************************************
* @brief  Muestra la fuerza g para el eje deseado.
*
* @param  axis - Eje leído.
*
* @return   None.
*****************************************************************************/
void ADXL345_Display_G_Force(char axis)
{
  u32   gForce    = 0;
  u32   gForceCalc  = 0;
  float   value     = 0;
  u32   whole     = 0;
  u32   thousands   = 0;
```

```
298   int   negative  = 0;
299
300   // Selecciona el eje a leer
301   switch(axis)
302   {
303     case 'x':
304       gForce = ADXL345_BurstReadReg(ADXL345_DATA_X0);
305       xil_printf("[X = ");
306       break;
307     case 'y':
308       gForce = ADXL345_BurstReadReg(ADXL345_DATA_Y0);
309       xil_printf("[Y = ");
310       break;
311     case 'z':
312       gForce = ADXL345_BurstReadReg(ADXL345_DATA_Z0);
313       xil_printf("[Z = ");
314       break;
315     default:
316       break;
317   }
318
319   // Si el resultado de la lectura es negativo, aplicar relleno con 0xFFFFF000
320   if((gForce & 0x1000) == 0x1000)
321   {
322     negative = 1;
323     gForceCalc = 0xFFFFF000 | gForce;
324     gForceCalc = gForceCalc - 1;
325     gForceCalc = ~gForceCalc;
326
327   }
328   // Si el resultado es positivo, no modificar
329   else
330   {
331     gForceCalc = gForce & 0x0FFF;
332   }
333
334   // Convierte a decimal
335   value = ((float)gForceCalc * 0.0039);
336
337   // Prepara e imprime los datos
338   // Si son positivos
339   if(negative == 0)
340   {
341     whole = value;
342     thousands = (value - whole) * 1000;
343
344     if(thousands > 99)
345     {
346       xil_printf("+%d.%3d] ", whole, thousands);
347     }
348     else if(thousands > 9)
349     {
350       xil_printf("+%d.0%2d] ", whole, thousands);
351     }
352     else
353     {
354       xil_printf("+%d.00%1d] ", whole, thousands);
355     }
356   }
357   // Si son negativos
358   else
359   {
360     whole = value;
361     thousands = (value - whole) * 1000;
362
363     if(thousands > 99)
364     {
365       xil_printf("-%d.%3d] ", whole, thousands);
366     }
367     else if(thousands > 9)
368     {
369       xil_printf("-%d.0%2d] ", whole, thousands);
370     }
371     else
372     {
373       xil_printf("-%d.00%1d] ", whole, thousands);
374     }
375   }
376 }
377
378 /*****************************************************************************
379 * @brief  Imprime todos los ejes.
380 *
381 * @param  None.
382 *
383 * @return   None.
384 *****************************************************************************/
385 void ADXL345_DisplayAllAxes(void)
386 {
387   ADXL345_Display_G_Force('x');
388   ADXL345_Display_G_Force('y');
389   ADXL345_Display_G_Force('z');
390   xil_printf("\n\r");
391 }
392
393 /*****************************************************************************
394 * @brief  Lee todos los ejes.
395 *
396 * @param  None.
397 *
```

```
* @return   None.
*****************************************************************************/
void ADXL345_ReadAllAxes(void)
{
  ADXL345_BurstReadReg(ADXL345_DATA_X0);
  ADXL345_BurstReadReg(ADXL345_DATA_Y0);
  ADXL345_BurstReadReg(ADXL345_DATA_Z0);
}

/*****************************************************************************
* @brief  Habilita mediciones en el ADXL345.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_Run(void)
{
  if(isRunning == 0)
  {
    ADXL345_WriteReg(ADXL345_POWER_CTL, (1 << ADXL345_Measure));
    isRunning = 1;
  }
  xil_printf("\n\r>Mediciones del ADXL345 Habilitadas!\n\r\n\r");
  ADXL345_DisplayMenu();
}

/*****************************************************************************
* @brief  Detiene las mediciones en el ADXL345.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_Stop(void)
{
  if(isRunning == 1)
  {
    ADXL345_WriteReg(ADXL345_POWER_CTL, (0 << ADXL345_Measure));
    isRunning = 0;
  }
  xil_printf("\n\r>Mediciones del ADXL345 Deshabilitadas!\n\r\n\r");
  ADXL345_DisplayMenu();
}

/*****************************************************************************
* @brief  Imprime la aceleración en los tres ejes.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_DisplayAccel(void)
{
  if(isRunning == 1)
  {
    if(ADXL345_ReadReg(ADXL345_INT_SOURCE) & 0x80)  // Comprueba si los datos están listos
    {
      ADXL345_DisplayAllAxes();
    }
  }
  else if(isRunning == 0)
  {
    xil_printf("\n\r!!! Mediciones no habilitadas. Por favor habilítalas antes de realizar la acción !!!\n\r");
  }
  ADXL345_DisplayMenu();
}

/*****************************************************************************
* @brief  Selecciona el rango de medida.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_SelectRange(void)
{
  u8 rx   = 0;
  u8 rxReg  = 0;

  ADXL345_DisplayRangeMenu();

  scanf(" %c", &rx);

  // Convierte el formato ASCII
  rx = rx - 0x31;

  if ((rx >= 0x00) && (rx <= 0x03))
    {
    rxReg = (ADXL345_ReadReg(ADXL345_DATA_FORMAT) & 0xFC) | rx;

    // Cambia el valor del registro DATA_FORMAT
    ADXL345_WriteReg(ADXL345_DATA_FORMAT, rxReg);
    }

  // Imprime la nueva configuración
  switch(rx)
  {
    case 0:
      xil_printf("> Rango +-2g seleccionado\n\r");
```

```
498       break;
499     case 1:
500       xil_printf("> Rango +-4g seleccionado\n\r");
501       break;
502     case 2:
503       xil_printf("> Rango +-8g seleccionado\n\r");
504       break;
505     case 3:
506       xil_printf("> Rango +-16g seleccionado\n\r");
507       break;
508     default:
509       xil_printf("> Rango de medida erróneo\n\r");
510       break;
511   }
512
513   ADXL345_DisplayMenu();
514 }
515
516 /*****************************************************************************
517 * @brief  Selecciona ratio de adquisición.
518 *
519 * @param  None.
520 *
521 * @return   None.
522 *****************************************************************************/
523 void ADXL345_AcquisitionRate(void)
524 {
525   u8 rx   = 0;
526   u8 rxReg  = 0;
527
528   ADXL345_DisplayRateMenu();
529
530   scanf(" %c", &rx);
531
532   // Convierte el formato ASCII
533   rx = rx - 0x30 + 0x05;
534
535   if ((rx >= 0x06) && (rx <= 0x0E))
536   {
537     rxReg = (ADXL345_ReadReg(ADXL345_BW_RATE) & 0xF0) | rx;
538
539     // Cambia el valor del registro BW_RATE
540     ADXL345_WriteReg(ADXL345_BW_RATE, rxReg);
541   }
542
543   // Imprime la nueva configuración
544   switch(rx)
545   {
546     case 0x06:
547       xil_printf("> 6.25 Hz seleccionado\n\r");
548       break;
549     case 0x07:
550       xil_printf("> 12.5 Hz seleccionado\n\r");
551       break;
552     case 0x08:
553       xil_printf("> 25 Hz seleccionado\n\r");
554       break;
555     case 0x09:
556       xil_printf("> 50 Hz seleccionado\n\r");
557       break;
558     case 0x0A:
559       xil_printf("> 100 Hz seleccionado\n\r");
560       break;
561     case 0x0B:
562       xil_printf("> 200 Hz seleccionado\n\r");
563       break;
564     case 0x0C:
565       xil_printf("> 400 Hz seleccionado\n\r");
566       break;
567     case 0x0D:
568       xil_printf("> 800 Hz seleccionado\n\r");
569       break;
570     case 0x0E:
571       xil_printf("> 1600 Hz seleccionado\n\r");
572       break;
573     default:
574       xil_printf("> Rango de adquisición erroneo\n\r");
575       break;
576   }
577
578   ADXL345_DisplayMenu();
579 }
580
581 /*****************************************************************************
582 * @brief  Selecciona el tipo de interrupciones por toque habilitadas.
583 *
584 * @param  None.
585 *
586 * @return   None.
587 *****************************************************************************/
588 void ADXL345_SelectTap(void)
589 {
590   u8 rx   = 0;
591   u8 rxReg  = 0;
592
593   ADXL345_DisplayTapMenu();
594
595   scanf(" %c", &rx);
596
597   // Habilita e imprime las opciones deseadas
```

```
598   switch(rx)
599   {
600     case '1':
601       rxReg = (ADXL345_ReadReg(ADXL345_INT_ENABLE) & 0x9F) | (1 << ADXL345_SINGLE_TAP);
602       ADXL345_WriteReg(ADXL345_INT_ENABLE, rxReg);
603       xil_printf("> Toque Simple seleccionado\n\r");
604       break;
605     case '2':
606       rxReg = (ADXL345_ReadReg(ADXL345_INT_ENABLE) & 0x9F) | (1 << ADXL345_DOUBLE_TAP);
607       ADXL345_WriteReg(ADXL345_INT_ENABLE, rxReg);
608       xil_printf("> Toque Doble seleccionado\n\r");
609       break;
610     case '3':
611       rxReg = (ADXL345_ReadReg(ADXL345_INT_ENABLE) & 0x9F) | (1 << ADXL345_SINGLE_TAP) | (1 <<
                ↪ ADXL345_DOUBLE_TAP);
612       ADXL345_WriteReg(ADXL345_INT_ENABLE, rxReg);
613       xil_printf("> Toques Simple y Doble seleccionados\n\r");
614       break;
615     case '4':
616       rxReg = ADXL345_ReadReg(ADXL345_INT_ENABLE) & 0x9F;
617       ADXL345_WriteReg(ADXL345_INT_ENABLE, rxReg);
618       xil_printf("> Toques deshabilitados\n\r");
619       break;
620     default:
621       xil_printf("> Opción de toque erronea\n\r");
622       break;
623   }
624
625   ADXL345_DisplayMenu();
626 }
627
628
629 /*****************************************************************************
630 * @brief  Calcula el valor medio de la aceleración del eje seleccionado.
631 *
632 * @param  axis - Eje seleccionado.
633 *
634 * @return   Avg - Media de aceleración.
635 *****************************************************************************/
636 long int ADXL345_CalculateAverage(char axis)
637 {
638   u32 gForce, gForceCalc;
639   long int Avg = 0;
640   int negative, i;
641
642   for(i=0;i<100;i++){   //Toma 100 mediciones para hacer la media
643     WaitFor(10000);
644     switch(axis)
645     {
646       case 'x':
647         gForce = ADXL345_BurstReadReg(ADXL345_DATA_X0);
648         break;
649       case 'y':
650         gForce = ADXL345_BurstReadReg(ADXL345_DATA_Y0);
651         break;
652       case 'z':
653         gForce = ADXL345_BurstReadReg(ADXL345_DATA_Z0);
654         break;
655       default:
656         break;
657     }
658
659     // Si el resultado de la lectura es negativo, aplicar relleno con 0xFFFFF000
660     if((gForce & 0x1000) == 0x1000)
661     {
662       negative = 1;
663       gForceCalc = 0xFFFFF000 | gForce;
664       gForceCalc = gForceCalc - 1;
665       gForceCalc = ~gForceCalc;
666
667     }
668     // Si el resultado es positivo, no modificar
669     else
670     {
671       negative = 0;
672       gForceCalc = gForce & 0x0FFF;
673     }
674     // Sumatorio de los valores obtenidos
675     if(negative == 0) {
676       Avg += gForceCalc;
677     }
678     else
679     {
680       Avg -= gForceCalc;
681     }
682   }
683
684   Avg = (long int)round((double)Avg / 100); //Calcula la media
685
686   return Avg;
687 }
688
689
690 /*****************************************************************************
691 * @brief  Calcula el valor de offset de la aceleración en los distintos ejes.
692 *
693 * @param  axis - Eje a leer.
694 *
695 * @return   Offset - Offset del eje seleccionado.
696 *****************************************************************************/
```

```
long int ADXL345_CalculateOffset(char axis)
{
  long int Offset;

  Offset = ADXL345_CalculateAverage(axis);

  if(axis == 'z')
  {
    Offset -= 256;    //Descontamos 1g del eje z (1/0.0039)
  }

  return Offset;
}

/*****************************************************************************
* @brief  Imprime los valores para la calibración de aceleración de los distintos ejes.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_DisplayCalibrationValue(void)
{
  long int Offset, X_CALIB, Y_CALIB, Z_CALIB;

  if(isRunning == 1)
  {
    // Esperar 11.1ms después de la inicialización del ADXL345. No necesario en este código.
    // Para una calibración más afinada corregir el offset a nivel de software
    xil_printf("\nEjecuta de nuevo si el sensor no está posicionado en la orientación X = 0g, Y = 0g, Z =
        ↪ +1g");
    xil_printf(" o si las variables XYZ_CALIB en ADXL345_Init() no están inicializadas a '0'.\n\r");
    xil_printf("El formato de datos debe ser: rango +-16g y modo 13 bits (por defecto)\n\r");

    // Cálculo de XYZ_CALIB basado en la fórmula de la guía de Analog Devices (an-1077)
    Offset = ADXL345_CalculateOffset('x');
    X_CALIB = -((long int)round((double)Offset / 4));
    Offset = ADXL345_CalculateOffset('y');
    Y_CALIB = -((long int)round((double)Offset / 4));
    Offset = ADXL345_CalculateOffset('z');
    Z_CALIB = -((long int)round((double)Offset / 4));

    // Cada LSB del resultado de calibración representa 15.6 mg
    xil_printf("Valores de calibración:\n\r");
    xil_printf("    > X_CALIB = %ld\n\r", X_CALIB);
    xil_printf("    > Y_CALIB = %ld\n\r", Y_CALIB);
    xil_printf("    > Z_CALIB = %ld\n\r", Z_CALIB);

  }
  else if(isRunning == 0)
  {
    xil_printf("\n\r!!! Mediciones no habilitadas. Por favor, habilítalas antes de realizar la acción
        ↪ !!!\n\r");
  }
  ADXL345_DisplayMenu();
}


/*****************************************************************************
* @brief  Calcula y verifica si los valores de auto-test son correctos.
*
* @param  None.
*
* @return   None.
*****************************************************************************/
void ADXL345_DisplaySelftestDelta(void)
{
  long int X_ST_OFF, Y_ST_OFF, Z_ST_OFF, X_ST_ON, Y_ST_ON, Z_ST_ON;
  long int X_ST, Y_ST, Z_ST;

  if(isRunning == 1)
  {
    // Esperar 11.1ms después de la inicialización del ADXL345. No necesario en este código.
    xil_printf("\nEl sensor debe ser colocado en un ambiente estable mientras se ejecuta la secuencia de
        ↪ auto-test.\n\r");

    X_ST_OFF = ADXL345_CalculateAverage('x');
    Y_ST_OFF = ADXL345_CalculateAverage('y');
    Z_ST_OFF = ADXL345_CalculateAverage('z');

    ADXL345_WriteReg(ADXL345_DATA_FORMAT, ((1 << ADXL345_SELF_TEST) |   // Activa el auto-test
                     (1 << ADXL345_FULL_RES)  | // Formato de datos = +- 16g, Resolución Completa
                     (3 << ADXL345_Range)));

    X_ST_ON = ADXL345_CalculateAverage('x');
    Y_ST_ON = ADXL345_CalculateAverage('y');
    Z_ST_ON = ADXL345_CalculateAverage('z');

    ADXL345_WriteReg(ADXL345_DATA_FORMAT, ((0 << ADXL345_SELF_TEST) |   // Desactiva el auto-test
                     (1 << ADXL345_FULL_RES)  | // Formato de datos = +- 16g, Resolución Completa
                     (3 << ADXL345_Range)));

    X_ST = X_ST_ON - X_ST_OFF;                    // Calcula los valores de auto-test
    Y_ST = Y_ST_ON - Y_ST_OFF;
    Z_ST = Z_ST_ON - Z_ST_OFF;

    xil_printf("Los valores de auto-test son:\n\r");
    xil_printf("    > X_ST = %d LSBs\n\r", X_ST);
    xil_printf("    > Y_ST = %d LSBs\n\r", Y_ST);
    xil_printf("    > Z_ST = %d LSBs\n\r", Z_ST);
```

```
794     /* Para Vs = 3.3V, los valores de auto-test son: X_ST = [0.35, 3.70]g, Y_ST = [-3.70, -0.35]g, Z_ST =
                ↪ [0.45, 5]g
795        Multiplica por 256 para cambiar de LSB a g */
796     if(X_ST < 0.35*256 || X_ST > 3.7*256 || Y_ST < -3.7*256 || Y_ST > -0.35*256 || Z_ST < 0.45*256 || Z_ST > 5
                ↪ *256)
797     {
798       xil_printf("Auto-test no superado\n\r");
799     }
800     else
801     {
802       xil_printf("Auto-test satisfactorio\n\r");
803     }
804   }
805   else if(isRunning == 0)
806   {
807     xil_printf("\n\r!!! Mediciones no habilitadas. Por favor, habilítalas antes de realizar la acción
                ↪ !!!\n\r");
808   }
809   ADXL345_DisplayMenu();
810 }
811
812
813 /*****************************************************************************
814 * @brief  Imprime en bucle el tiempo transcurrido y los valores de aceleración.
815 *
816 * @param  None.
817 *
818 * @return   None.
819 *****************************************************************************/
820
821 void ADXL345_DisplayAccelLoop(void)
822 {
823   u8 INT_SOURCE;
824   int Stop = 0;
825
826     if(isRunning == 1)
827     {
828
829       xil_printf("\n\rSi está habilitado, un toque simple pausa la adquisición de datos. ");
830       xil_printf("Un doble toque devuelve al menú principal.\n\r");
831
832       ADXL345_ReadReg(ADXL345_INT_SOURCE);  // Limpia el registro de interrupciones
833
834       xil_printf("\n\r[Tiempo(s)] [ax(g)] [ay(g)] [az(g)]\n\n\r");
835       InitializeTimer(&TimerCounterInst, TMRCTR_INTERRUPT_ID, &IntcInstance); // Inicializa el contador
836       while(1)
837       {
838         // Espera para evitar que se verifique demasiado rápido si hay nuevos datos disponibles
839         WaitFor(1000000);
840         INT_SOURCE = ADXL345_ReadReg(ADXL345_INT_SOURCE);
841         if(INT_SOURCE & 0x20)   // Verifica si ha habido una interrupción por doble toque
842         {
843           break;
844         }
845         else if(INT_SOURCE & 0x40)  // Verifica si ha habido una interrupción por toque simple
846         {
847           Stop = ~Stop;       // Pausa o retoma la adquisición
848         }
849         else if ((INT_SOURCE & 0x80) && (Stop == 0))  // Verifica si hay datos de aceleración disponibles
850         {
851           WaitFor(1000000);
852           GetTime(&TimerCounterInst, TMRCTR_INTERRUPT_ID, &IntcInstance);  // Imprime el tiempo
853           ADXL345_DisplayAllAxes(); // Imprime la aceleración
854         }
855       }
856     }
857   else if(isRunning == 0)
858   {
859     xil_printf("\n\r!!! Mediciones no habilitadas. Por favor habilítalas antes de realizar la acción !!!\n\r");
860   }
861
862   ClockRead = 0;
863   ADXL345_ReadReg(ADXL345_INT_SOURCE);  // Limpia el registro de interrupciones
864   ADXL345_DisplayMenu();
865 }
866
867 /*****************************************************************************
868 * @brief  Espera durante x iteraciones
869 *
870 * @param  cont - Número de iteraciones.
871 *
872 * @return   None.
873 *****************************************************************************/
874
875 void WaitFor(int cont)
876 {
877   int i;
878
879   for(i=0;i<cont;i++);
880 }
881
882
883
884 /*****************************************************************************
885 * @brief  Imprime el tiempo desde la inicialización del contador
886 *
887 * @param  *TmrCtrInstancePtr - Puntero a la instancia del contador
888 *
889 * @param  TmrCtrIntrId - ID del contador en el sistema de interrupciones
890 *
```

```
* @return None.
******************************************************************************/
void GetTime(XTmrCtr *TmrCtrInstancePtr, u16 TmrCtrIntrId, INTC *IntcInstancePtr)
{
  /* Configurado para un reloj de 100MHz. El contador está formado por dos
     contadores de 32 bits. */

  u32 Tiempo_s;
  int Tiempo_ms;
    u32 Contador1, Contador0;

    // Conecta las interrupciones del contador
  XIntc_Connect(IntcInstancePtr, TmrCtrIntrId, (XInterruptHandler)XTmrCtr_InterruptHandler, (void
              ↪ *)TmrCtrInstancePtr);

  // Lee los registros del contador
    Contador1 = XTmrCtr_GetValue(TmrCtrInstancePtr, (u32)TIMER_CNTR_1);
    Contador0 = XTmrCtr_GetValue(TmrCtrInstancePtr, (u32)TIMER_CNTR_0);

    if(ClockRead == 0){   // Habilita flag de primera lectura realizada
      TimerOffset = Contador1;  // Offset del contador al reiniciar
      ClockRead = 1;
    }

    // Calcula el tiempo en segundos y milisegundos
    Tiempo_s = 42.94967296 * (Contador1 - TimerOffset) + 0.00000001 * Contador0;
    Tiempo_ms = ((42.94967296 * (Contador1 - TimerOffset) + 0.00000001 * Contador0) * 1000) - Tiempo_s * 1000;

    xil_printf("[t = %lu.", Tiempo_s);
  if (Tiempo_ms < 10) {
    xil_printf("00%d] ", Tiempo_ms);
  } else if (Tiempo_ms < 100) {
    xil_printf("0%d] ", Tiempo_ms);
  } else {
    xil_printf("%d] ", Tiempo_ms);
  }

  XIntc_Disconnect(IntcInstancePtr, TmrCtrIntrId);
}


/*****************************************************************************
* @brief  Configura el sistema de interrupciones e inicializa sus periféricos.
*
* @param  *IntcInstancePtr - Puntero a la instancia del controlador de interrupciones
*
* @param  *IicInstPtr - Puntero a la instancia del dispositivo IIC
*
* @param  *TmrCtrInstancePtr - Puntero a la instancia del contador
*
* @param  TmrCtrDeviceId - ID del contador
*
* @param  TmrCtrIntrId - ID del contador en el sistema de interrupciones
*
* @return XST_SUCCESS or XST_FAILURE.
******************************************************************************/
int SetupIntrSystem(INTC *IntcInstancePtr, XIic *IicInstPtr, XTmrCtr *TmrCtrInstancePtr, u16 TmrCtrDeviceId,
              ↪ u16 TmrCtrIntrId)
{
  int Status;
  XIic_Config *ConfigPtrI2C; // Puntero a los datos de configuración del I2C

  // Inicializa los drivers del controlador de interrupciones
  Status = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  // I2C - Inicializa los drivers
  ConfigPtrI2C = XIic_LookupConfig(IIC_DEVICE_ID);
  if (ConfigPtrI2C == NULL) {
    return XST_FAILURE;
  }

  Status = XIic_CfgInitialize(&IicInstance, ConfigPtrI2C, ConfigPtrI2C->BaseAddress);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  // Contador - Inicializa los drivers
  Status = XTmrCtr_Initialize(TmrCtrInstancePtr, TmrCtrDeviceId);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  // I2C - Realiza un auto-test
  Status = XIic_SelfTest(&IicInstance);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  // Contador - Realiza un auto-test en ambos contadores de 32 bits
  Status = XTmrCtr_SelfTest(TmrCtrInstancePtr, TIMER_CNTR_0);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  Status = XTmrCtr_SelfTest(TmrCtrInstancePtr, TIMER_CNTR_1);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
```

```
989   }
990
991   // I2C - Conecta el controlador de drivers del dispositivo al sistema de interrupciones.
992   Status = XIntc_Connect(&IntcInstance, IIC_INTR_ID, (XInterruptHandler) XIic_InterruptHandler, IicInstPtr);
993   if (Status != XST_SUCCESS) {
994     return XST_FAILURE;
995   }
996
997   // Inicializa el controlador de interrupciones
998   Status = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);
999   if (Status != XST_SUCCESS) {
1000    return XST_FAILURE;
1001  }
1002
1003  // I2C - Habilita las interrupciones
1004  XIntc_Enable(&IntcInstance, IIC_INTR_ID);
1005
1006  // Contador - Habilita las interrupciones
1007  XIntc_Enable(IntcInstancePtr, TmrCtrIntrId);
1008
1009  // I2C - Habilita los controladores para recepción y transmisión
1010  XIic_SetSendHandler(&IicInstance, &IicInstance, (XIic_Handler) SendHandler);
1011  XIic_SetRecvHandler(&IicInstance, &IicInstance, (XIic_Handler) ReceiveHandler);
1012  XIic_SetStatusHandler(&IicInstance, &IicInstance, (XIic_StatusHandler) StatusHandler);
1013
1014  // Inicializa la tabla de excepciones
1015  Xil_ExceptionInit();
1016
1017  // Registra el controlador de interrupciones con la tabla de excepciones
1018  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT, (Xil_ExceptionHandler)INTC_HANDLER, IntcInstancePtr);
1019
1020  // Habilita las excepciones
1021  Xil_ExceptionEnable();
1022
1023  return XST_SUCCESS;
1024 }
1025
1026 /*****************************************************************************
1027 * @brief  Este controlador del dispositivo IIC es llamado de manera asíncrona
1028 *       e indica que los datos del buffer especificado han sido enviados
1029 *
1030 * @param  *InstancePtr - Puntero a la instancia del dispositivo IIC
1031 *
1032 * @return None.
1033 *****************************************************************************/
1034 static void SendHandler(XIic *InstancePtr)
1035 {
1036   TransmitComplete = 0;
1037 }
1038
1039 /*****************************************************************************
1040 * @brief  Este controlador del dispositivo IIC es llamado de manera asíncrona
1041 *       e indica que los datos del buffer especificado han sido recibidos
1042 *
1043 * @param  *InstancePtr - Puntero a la instancia del dispositivo IIC
1044 *
1045 * @return None.
1046 *****************************************************************************/
1047 static void ReceiveHandler(XIic *InstancePtr)
1048 {
1049   ReceiveComplete = 0;
1050 }
1051
1052 /*****************************************************************************
1053 * @brief  Este controlador del dispositivo IIC es llamado de manera asíncrona
1054 *       e indica los eventos que han ocurrido
1055 *
1056 * @param  *InstancePtr - Puntero a la instancia del dispositivo IIC
1057 *
1058 * @return None.
1059 *****************************************************************************/
1060 static void StatusHandler(XIic *InstancePtr, int Event)
1061 {
1062
1063 }
1064
1065 /*****************************************************************************
1066 * @brief  Inicializa el contador
1067 *
1068 * @param  *TmrCtrInstancePtr - Puntero a la instancia del contador
1069 *
1070 * @param  TmrCtrIntrId - ID del contador en el sistema de interrupciones
1071 *
1072 * @param  *IntcInstancePtr - Puntero a la instancia del controlador de interrupciones
1073 *
1074 * @return None.
1075 *****************************************************************************/
1076 void InitializeTimer(XTmrCtr *TmrCtrInstancePtr, u16 TmrCtrIntrId, INTC *IntcInstancePtr)
1077 {
1078   if(FirstTimeInitializingClk == 0){  //Verifica que es la primera vez que se inicializa
1079
1080     // Habilita los contadores simultáneamente, habilita las interrupciones, el modo cascada y el modo captura.
1081     XTmrCtr_SetOptions(TmrCtrInstancePtr, TIMER_CNTR_0, XTC_INT_MODE_OPTION | XTC_CASCADE_MODE_OPTION |
                ↪ XTC_ENABLE_ALL_OPTION | XTC_CAPTURE_MODE_OPTION);
1082
1083     // Establece el valor de reinicio y reinicia ambos contadores
1084     XTmrCtr_SetResetValue(TmrCtrInstancePtr, TIMER_CNTR_0, RESET_VALUE_CNTR_0);
1085     XTmrCtr_SetResetValue(TmrCtrInstancePtr, TIMER_CNTR_1, RESET_VALUE_CNTR_1);
1086     XTmrCtr_Reset(TmrCtrInstancePtr, TIMER_CNTR_0);
1087     XTmrCtr_Reset(TmrCtrInstancePtr, TIMER_CNTR_1);
```

```
1088      FirstTimeInitializingClk = 1; // Activa la flag
1089    }
1090
1091    // Conecta, inicializa y desconecta el contador para que pueda ser llamado
1092    XIntc_Connect(IntcInstancePtr, TmrCtrIntrId, (XInterruptHandler)XTmrCtr_InterruptHandler, (void
                ↪ *)TmrCtrInstancePtr);
1093    XTmrCtr_Start(TmrCtrInstancePtr, TIMER_CNTR_0);
1094    XIntc_Disconnect(IntcInstancePtr, TmrCtrIntrId);
1095  }
```

Código C.6: adxl345.c

D. Diagrama de bloques NEXYS A7

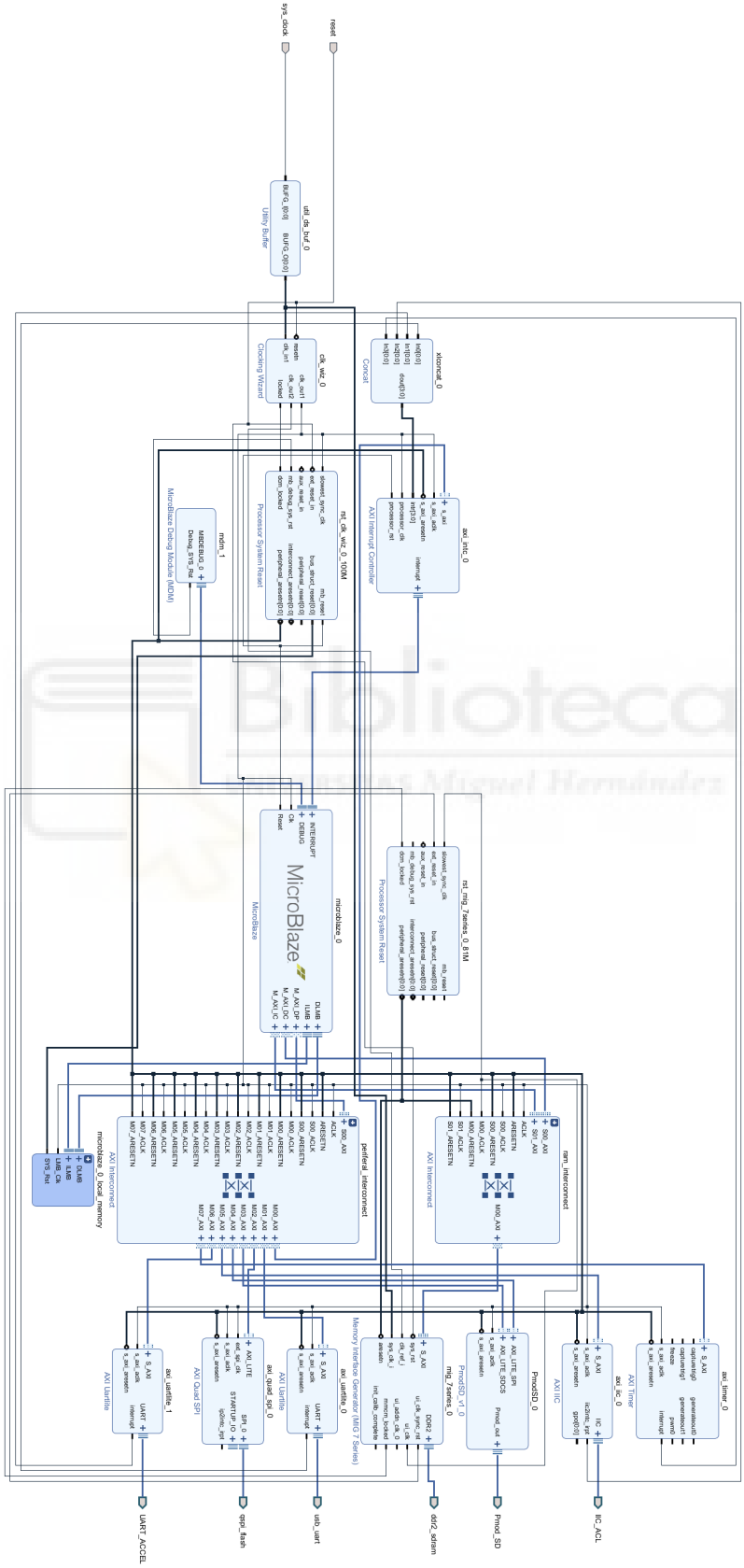

Figura D.2: Diagrama de bloques de la NEXYS A7.

E. *Constraints* NEXYS A7

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports {
          ↪ sys_clock }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
          ↪ {sys_clock}];

set_property CLOCK_DEDICATED_ROUTE TRUE [get_nets
          ↪ system_i/util_ds_buf_0/U0/BUFG_O[0]];

##Pmod Header JA

set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 PULLUP true }
          ↪ [get_ports { IIC_ACL_scl_io }]; #IO_L20N_T3_A19_15 Sch=ja[1]
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { JA[2]
          ↪ }]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]
set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 PULLUP true }
          ↪ [get_ports { IIC_ACL_sda_io }]; #IO_L21P_T3_DQS_15 Sch=ja[3]
set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports {
          ↪ UART_ACCEL_rxd }]; #IO_L18N_T2_A23_15 Sch=ja[4]
#set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports { JA[7]
          ↪ }]; #IO_L16N_T2_A27_15 Sch=ja[7]
#set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports { JA[8]
          ↪ }]; #IO_L16P_T2_A28_15 Sch=ja[8]
#set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports { JA[9]
          ↪ }]; #IO_L22N_T3_A16_15 Sch=ja[9]
set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports {
          ↪ UART_ACCEL_txd }]; #IO_L22P_T3_A17_15 Sch=ja[10]
```

Código E.7: NEXYS-A7-100T-Master.xdc

```
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]
```

Código E.8: synthesis_flash.xdc

BIBLIOGRAFÍA

[1] *UG-065: Evaluation Board User Guide for ADXL345*, Analog Devices, 2010.

[2] *ADXL345 (Rev. G)*, Analog Devices, Accedido en mayo 2024.

[3] *I^2C*, `https://en.wikipedia.org/?title=I2C&redirect=no`, Accedido en mayo 2024, Wikipedia.

[4] *Physical Layer Simplified Specification*, SD Association, 2023.

[5] *Serial Peripheral Interface*, `https://en.wikipedia.org/wiki/Serial_Peripheral_Interface`, Accedido en mayo 2024, Wikipedia.

[6] *ADuC7019/20/21/22/24/25/26/27/28/29 Data Sheet*, Analog Devices, 2018.

[7] *Programing the ADXL34x datalogger*, Accedido en junio 2024, Analog Devices.

[8] A. Lukats, *TE0725 Technical Reference Manual*, Accedido en marzo 2024, Trenz Electronic.

[9] *TE0725-100-2I9 Schematic*, Trenz Electronic, 2016.

[10] A. Naseri, *TE0709 Technical Reference Manual*, Accedido en marzo 2024, Trenz Electronic.

[11] *MicroBlaze Processor*, `https://www.xilinx.com/products/design-tools/microblaze.html`, Accedido en marzo 2024, AMD.

[12] *Vivado Overview*, `https://www.xilinx.com/products/design-tools/vivado.html`, Accedido en junio 2024, Xilinx.

[13] ryanvergel, *Programming an Embedded MicroBlaze Processor*, https://github.com/Xilinx/Embedded-Design-Tutorials/tree/master/docs/Getting_Started/microblaze-system, Accedido en marzo 2024, 2024.

[14] J. Hartfiel, *TE0725 Reference Designs*, https://wiki.trenz-electronic.de/display/PD/TE0725+Reference+Designs, Accedido en marzo 2024, 2018.

[15] *Trenz Electronic Board Files*, https://github.com/Xilinx/XilinxBoardStore/tree/2022.2/boards, Accedido en junio 2024, Xilinx.

[16] *Advanced eXtensible Interface*, https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface, Accedido en junio 2024, Wikipedia.

[17] *What You Need to Know about HyperRAM™ – An Alternative Memory Option*, https://blog.techdesign.com/hyperram-alternative-memory-option/, Accedido en junio 2024, TECHDesign.

[18] OVGN, *OpenHBMC*, https://github.com/OVGN/OpenHBMC, Última actualización octubre 2022. Accedido en marzo 2024.

[19] *EVAL-ADXL345Z-DB Layout*, Analog Devices, 2021.

[20] *The Vitis Software Platform Development Environment*, https://www.xilinx.com/products/design-tools/vitis.html, Accedido en junio 2024, Xilinx.

[21] *ADXL345 Pmod Xilinx FPGA Reference Design*, https://wiki.analog.com/resources/fpga/xilinx/pmod/adxl345, Última modificación enero 2021. Accedido en abril 2024, Analog Devices.

[22] *xiic_eeprom_example.c*, https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/iic/examples, Versión 14 de julio de 2023, Xilinx.

[23] *xtmrctr_intr_example.c*, https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/tmrctr/examples, Versión 1 de abril de 2024, Xilinx.

[24] T. Tusuzki, *ADXL345 Quick Start Guide*, Accedido en junio 2024, Analog Devices.

[25] *Nexys A7 Reference Manual*, Revisión 10 julio, 2019, Digilent.

[26] *Digilent Board Files*, https://github.com/Digilent/vivado-boards, Accedido en junio 2024, Digilent.

[27] ChaN, *FAT Filesystem*, http://elm-chan.org/docs/fat_e.html, Accedido en junio 2024.

[28] *Pmod SD Reference Manual*, Accedido en junio 2024, Digilent.

[29] filipamator, *SDCard_AXI4*, https://github.com/filipamator/SDCard_AXI4, Accedido en junio 2024.

[30] ChaN, *FatFs - Generic FAT Filesystem Module*, http://elm-chan.org/fsw/ff/00index_e.html, Accedido en junio 2024.

[31] D. Gisselquist, *SD-Card controller*, https://github.com/ZipCPU/sdspi, Accedido en junio 2024.

[32] *Nexys 4 DDR Programming Guide*, https://digilent.com/reference/learn/programmable-logic/tutorials/nexys-4-ddr-programming-guide/start, Accedido en julio 2024, Digilent.

[33] *Curing methods for Microblaze*, https://www.cnblogs.com/milianke/p/17693804.html, Accedido en julio 2024, Cnblogs.