PROGRAMA UNIVERSITARIO DE IMAGINATION

Práctica 10 RVfpga
Buses Serie

1. INTRODUCCIÓN

En esta práctica se describe en primer lugar cómo funcionan los buses serie y las características principales de uno de los buses serie más típicos que se utilizan actualmente, el bus SPI (Sección 2). A continuación, nos centramos en el acelerómetro SPI disponible en la placa Nexys A7: se analiza la especificación de alto nivel de este periférico y se proponen ejercicios básicos (Secciones 3 y 4), y luego se analiza su implementación de bajo nivel y se proponen algunos ejercicios avanzados (Secciones 5 y 6).

2. BUSES SERIE - EL BUS SPI

Los buses paralelos envían varios bits a la vez, mientras que los buses serie envían la información de bit en bit. En primer lugar, vamos a comparar estos dos esquemas de comunicación y luego se describirá el protocolo SPI (*serial peripheral interface*), que es uno de los buses serie más comunes que se utilizan actualmente. Puede encontrar mucha información en Internet para ampliar sus conocimientos sobre este importante protocolo de comunicación.

Como ya se ha demostrado en prácticas anteriores, el principal propósito de los sistemas electrónicos integrados es conectar procesadores y circuitos para crear las funciones deseadas. Para que los procesadores y los circuitos compartan información, deben compartir un protocolo de comunicación común. Se han definido cientos de protocolos de comunicación para lograr este intercambio de datos y, en general, se pueden separar en dos categorías principales: interfaces paralelas o serie.

Las interfaces paralelas transfieren múltiples bits en paralelo, es decir, al mismo tiempo. Requieren buses (múltiples cables) de datos. Por ejemplo, el protocolo puede transmitir ocho, dieciséis o más bits al mismo tiempo (véase la Figura 1). También requieren un reloj para determinar cuándo hay nuevos grupos de *N bits* de datos listos para ser transferidos.

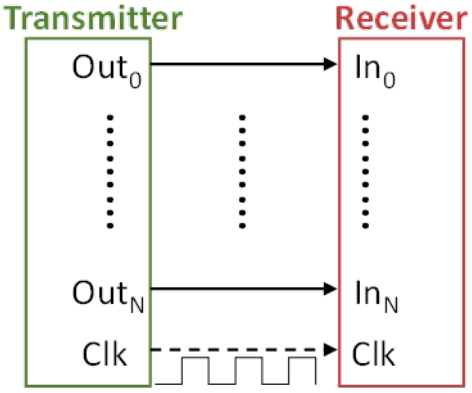

Figura 1. Ejemplo de un bus de datos paralelo de n bits.

A diferencia de la comunicación paralela, las interfaces serie transmiten sólo un bit de sus datos a la vez. Estas interfaces pueden funcionar usando tan sólo un cable y normalmente nunca más de cuatro. Figura 2 muestra un ejemplo de interfaz serie con un cable para datos y otro para un reloj. En cada nuevo flanco de reloj, se transfiere un nuevo bit de datos.

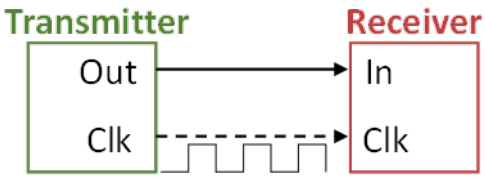

Figura 2. Ejemplo de un bus de datos serie de 1 bit.

La comunicación paralela posee los beneficios de ser rápida, directa y relativamente fácil de implementar. Sin embargo, requiere muchas más líneas de entrada/salida (E/S). Así que, debido a que los pines son limitados, los sistemas empotrados a menudo optan por la comunicación serie, sacrificando la potencial velocidad por el número real de pines.

Bus SPI:

El protocolo de Interfaz Periférica Serie (SPI por sus siglas en inglés) es una de las interfaces más utilizadas entre el microcontrolador y los circuitos integrados periféricos tales como sensores, ADCs, DACs, registros de desplazamiento, SRAM y otros. SPI es una interfaz síncrona y completamente dúplex basada en la comunicación controlador-periférico.

El bus SPI suele comunicarse a través de 4 puertos (véase la Figura 3):

- **SDO** - Salida de datos en serie: Salida del controlador al dispositivo periférico
- **SDI** - Entrada de datos en serie: Entrada del controlador desde el dispositivo periférico
- **SCK** - Reloj en serie: Enviado desde el controlador al dispositivo periférico
- **CS** - Chip Select: Señal activa en bajo; El controlador envía una señal al periférico (0 cuando se selecciona el periférico)

Nota: históricamente, a SDO también se le ha llamado MOSI (salida de datos del maestro, entrada de datos del esclavo) y a SDI se le ha llamado MISO (entrada de datos del maestro, salida de datos de esclavo). Esos términos están obsoletos y se les considera ofensivos, pero todavía existen en la literatura y en la documentación.

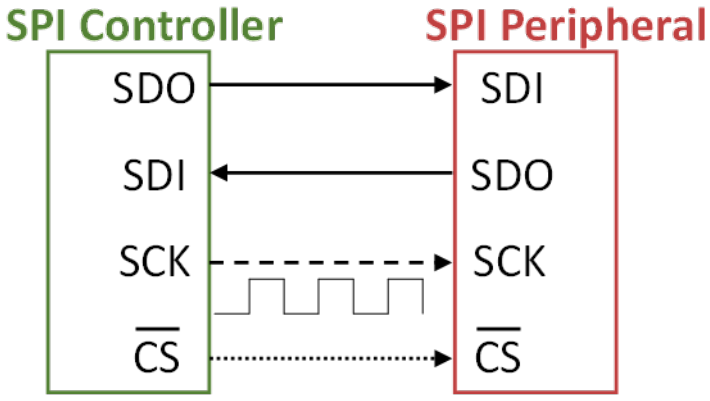

Figura 3. Ejemplo de un sistema con un controlador de SPI y un periférico de SPI.

Los datos serie se sincronizan con el flanco ascendente o descendente del reloj. SPI es una interfaz full-duplex; el controlador y el periférico pueden enviar datos al mismo tiempo a través de las líneas SDO y SDI, respectivamente. Las interfaces SPI sólo tienen un controlador, pero pueden tener múltiples periféricos. Cuando se conecta más de un periférico, se utilizan múltiples señales de selección de chip activas en bajo (CS barra) del

controlador para seleccionar el periférico al que se accede. SDO y SDI son las líneas de datos en serie: SDO (*serial data out*) son los datos de salida del controlador al periférico y SDI (*serial data in*) son los datos de entrada del periférico al controlador.

Para iniciar la comunicación SPI, el controlador debe seleccionar el dispositivo periférico (activando la señal CSbar, es decir, CSbar = 0) y luego enviar la señal del reloj al periférico. Durante la comunicación SPI, los datos se transmiten simultáneamente desde y hacia el controlador a través de las señales SDO y SDI, respectivamente. El flanco del reloj serie (SCK) sincroniza el muestreo de los datos.

La interfaz SPI también proporciona señales adicionales, CPOL y CPHA, para seleccionar el estado inactivo del reloj y la fase de muestreo de la señal. La señal de polaridad del reloj (CPOL) es 0 cuando el reloj (SCK) está inactivo en 0, y 1 cuando está inactivo en 1. La señal de fase de reloj (CPHA) selecciona la fase del reloj para enviar y muestrear datos. Cuando CPHA = 0, los datos (en SDI o SDO) se muestrean en el flanco destacado (es decir, el primer flanco después de que SCK deje de estar inactivo - y en cada ciclo posterior); por lo tanto, los datos (SDI y SDO) deben cambiar en el flanco posterior, como se muestra en los dos diagramas de temporización superiores de la Figura 4. CPHA = 1 hace lo contrario: los datos se muestrean en el flanco posterior y los datos cambian en el flanco destacado, como se muestra en las dos figuras inferiores de la Figura 4. El flanco en el que se transmiten nuevos datos también se llama *flanco de desplazamiento*, porque esta comunicación en serie se implementa típicamente utilizando un registro de desplazamiento.

La interfaz SPI que se utilizará en esta práctica es CPHA = 0 y CPOL = 0, por lo que SCK está inactivo en 0 y el controlador y el periférico muestrean los datos en el flanco de subida y desplaza los nuevos datos en la línea (SDO o SDI) justo después de cada flanco de bajada, como se muestra en el diagrama de tiempo superior de la Figura 4. Tenga en cuenta que cuando SCK está inactivo, y justo antes de que suba, SDO y SDI deben acarrear el bit más significativo del siguiente byte de datos.

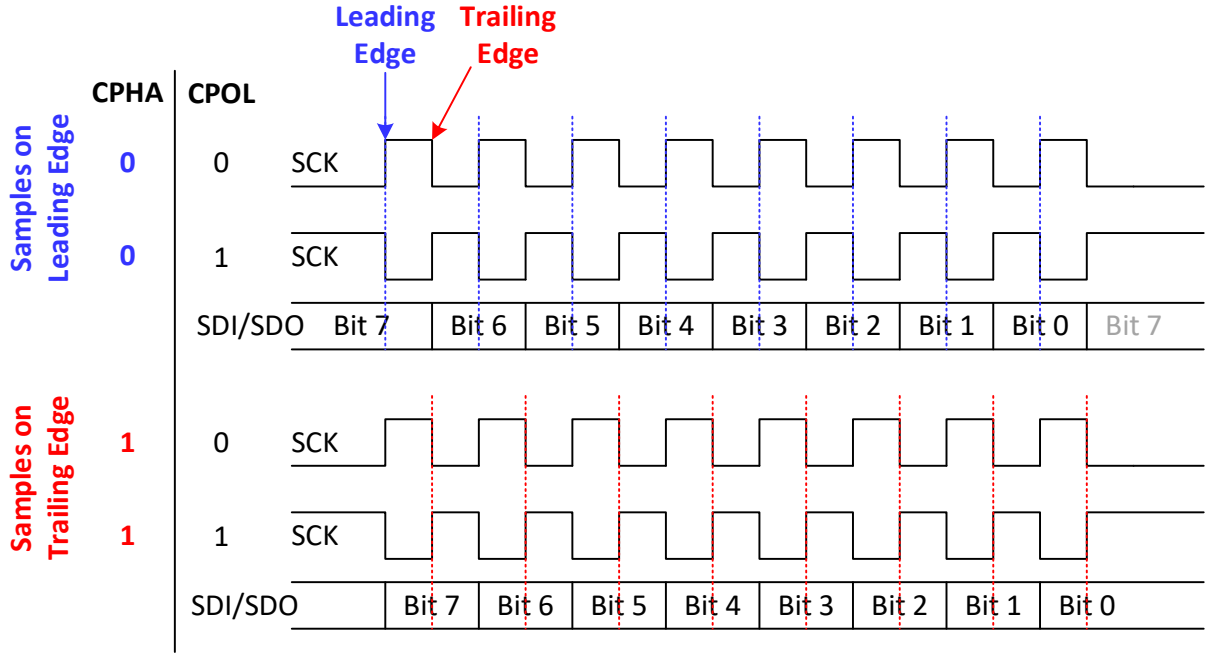

Figura 4. Relación de la CPHA/CPOL con el muestreo/envío de datos

3. ACELERÓMETRO SPI: ESPECIFICACIÓN DE ALTO NIVEL

Muchos periféricos incluyen una interfaz SPI. Por ejemplo, el acelerómetro de la placa Nexys A7 tiene una interfaz SPI. En esta sección se describe la especificación de alto nivel del controlador SPI del Sistema RVfpga y se introduce el acelerómetro ADXL362 incluido en la placa Nexys A7. También se introduce un ejercicio que utiliza el acelerómetro.

A. Especificación del controlador SPI

El módulo SPI del Sistema RVfpga proviene de OpenCores (https://opencores.org/projects/simple_spi). Si se descarga el paquete, se proporciona un documento que describe la especificación de alto nivel del módulo. Este documento también se proporciona aquí:

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/spi/docs/simple_spi.pdf

A continuación, se resume el funcionamiento y las características principales del módulo SPI; sin embargo, para más información, véase el documento anterior.

Este módulo tiene las siguientes características principales:

- Es compatible con las especificaciones SPI de Motorola.
- Utiliza la interfaz clásica WISHBONE RevB.3 de 8 bits.
- Contiene un buffer FIFO de lectura de 4 entradas y un buffer FIFO de escritura de 4 entradas.
- Permite la generación de interrupciones después de 1, 2, 3, o 4 bytes transferidos.
- Puede funcionar con una amplia gama de frecuencias de reloj de entrada.
- Es completamente sintetizable.

En la sección 3 de la especificación básica del SPI se describen los registros de control y estado disponibles dentro del módulo SPI, cada uno de los cuales está asignado a una dirección diferente (véase la Tabla 1). La dirección base del controlador SPI es **0x80001100**. Estos registros se describen en detalle a continuación.

Tabla 1. Registros del SPI

Nombre	Dirección	Ancho	Acceso	Descripción
SPCR	0x80001100	8	R/W	Registro de control
SPSR	0x80001108	8	R/W	Registro de estado
SPDR	0x80001110	8	R/W	Registro de datos
SPER	0x80001118	8	R/W	Registro de extensiones
SPCS	0x80001120	8	R/W	Registro CS

El Registro de Control de SPI (SPCR) controla el módulo SPI; la Tabla 2 muestra la función de cada uno de sus bits.

Tabla 2. Bits del SPCR

Bit	Acceso	Nombre y descripción
0:1	R/W	**SPR** Velocidad de reloj SPI: Estos bits seleccionan la velocidad del reloj del SPI.
2	R/W	**CPHA** Fase del reloj: Determina la fase de muestreo y envío de datos. Cuando CPHA = 1, los nuevos datos se desplazan en el flanco destacado y los datos se muestrean en el flanco posterior. Cuando CPHA = 0, los nuevos datos se desplazan en el flanco posterior y se muestrean en el flanco destacado.

3	R/W	**CPOL** Polaridad del reloj: Determina el estado inactivo del reloj SPI (SCK). Cuando CPOL = 0, SCK está inactivo en 0, cuando CPOL = 1, SCK está inactivo en 1.
4	R/W	**MSTR** Selección de modo: Cuando MSTR = 1, el núcleo SPI es un dispositivo controlador. Este es el único modo soportado para este controlador.
6	R/W	**SPE** Activación de SPI: Cuando SPE = 1, el núcleo SPI está habilitado. Cuando SPE = 0 el núcleo SPI está desactivado.
7	R/W	**SPIE** Habilitación de la interrupción del SPI: Si SPIE = 1, cuando el flag de interrupción SPI en el registro de estado está activo, el host es interrumpido.

El Registro de Estado SPI (SPSR) proporciona el estado del módulo SPI; la Tabla 3 muestra la función de cada uno de sus bits.

Tabla 3. Bits del SPSR

Bit	Acceso	Descripción
0	R/W	**RFEMPTY** FIFO de lectura vacío: Si RFEMPTY = 1, el FIFO de lectura está vacío.
1	R/W	**RFFULL** FIFO de lectura lleno: Si RFFULL = 1, el FIFO de lectura está lleno.
2	R/W	**WFEMPTY** FIFO de escritura vacío: SI WFEMPTY = 1, el FIFO de escritura está vacío.
3	R/W	**WFFULL** FIFO de escritura lleno: SI WFFULL = 1, el FIFO de escritura está lleno.
6	R/W	**WCOL** Flag de colisión de escritura: Cuando WCOL = 1, el registro SPDATA fue escrito mientras el FIFO de escritura estaba lleno. Al escribir un 1 a WCOL se borra este bit.
7	R/W	**SPIF** Bandera de Interrupción SPI: SPIF = 1 al completar la transferencia de un bloque. Si SPIF está a 1 y SPIE activado, se genera una interrupción. Al escribir un 1 en SPIF se borra el flag.

El Registro de Datos SPI (SPDR) proporciona los datos a leer o escribir. El controlador SPI incluye un Buffer de Escritura 4 x 8-bit y un Buffer de Lectura 4x 8-bit.

El Registro Extendido SPI (SPER) proporciona alguna funcionalidad adicional; Tabla 4 describe los diferentes campos que contiene.

Tabla 4. Bits del SPER

Bit	Acceso	Descripción
0:1	R/W	**ESPR** Selección de la velocidad de reloj del SPI extendido: Añade dos bits al SPR (Velocidad de reloj SPI).
6:7	R/W	**ICNT** Conteo de Interrupción: Determinar el tamaño del bloque de transferencia. El bit SPIF se activa después de ICNT transferencias de 8 bits. Así, es posible reducir la carga del kernel debido a la reducción de la frecuencia de las llamadas a la rutina de servicio de interrupción.

Finalmente, el Registro de Selección de Chip SPI (SPCS) selecciona el periférico a utilizar. El ancho de esta señal es configurable a través del parámetro SS_WIDTH (*SPI Select*

Width). En el Sistema RVfpga, sólo existe un periférico para cada interfaz SPI, por lo que SS_WIDTH = 1.

TAREA: Localizar la declaración de los registros SPCR, SPSR, SPDR, SPER y SPCS en el módulo SPI, así como la definición de sus direcciones. El módulo SPI está disponible dentro de la carpeta *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Periféricos/spi*.

B. Especificación del acelerómetro ADXL362

La placa Nexys A7 incluye el acelerómetro ADXL362 de Analog Devices. Puede encontrar la información completa del dispositivo en su hoja de datos, localizada aquí: https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf

El ADXL362 es un acelerómetro MEMS de 3 ejes de baja potencia (>2µA) con velocidad de datos de salida de 100 transacciones por segundo. En modo de activación por movimiento consume muy baja potencia (270 nA). Proporciona una resolución de salida de 12 bits, aunque también cuando es suficiente una resolución menor, se proporcionan datos en formato de 8 bits. Dispone de rangos de medición de ±2 g (1 g = 9.8 m/s^2) , ±4 g y ±8 g, con una resolución de 1 mg/LSB (mili-g por bit) en el rango de ±2 g. Mientras el ADXL362 está en el Modo de Medición, mide continuamente y almacena los datos de aceleración en los registros de datos X, Y y Z.

El acelerómetro ADXL362 incluye varios registros (Tabla 5) que permiten al usuario configurarlo y leer los datos de aceleración. El dispositivo se configura escribiendo en los registros de control, y se accede a los datos del acelerómetro leyendo los registros del dispositivo. Toda comunicación con el dispositivo debe especificar una dirección de registro y una bandera (*flag*) que indique si la comunicación es de lectura o de escritura. La transferencia de datos se produce después de que la dirección del registro y el *flag* de comunicación se envían al dispositivo.

Este acelerómetro actúa como un dispositivo periférico usando un esquema de comunicación SPI. La interfaz entre la FPGA y el acelerómetro se muestra en la Figura 5.

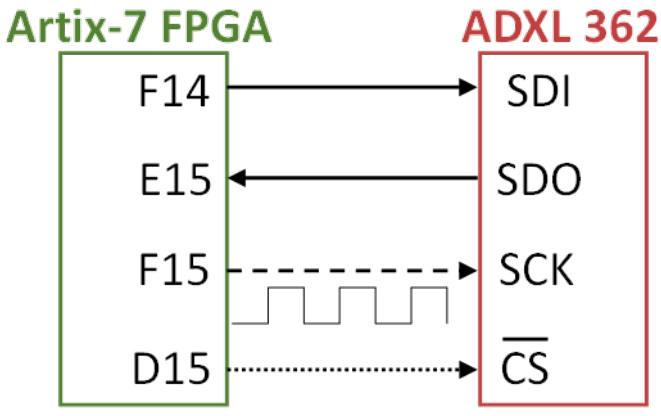

Figura 5. Interfaz del acelerómetro ADXL362 con la placa Nexys A7

La frecuencia de reloj recomendada para la comunicación SPI con el ADXL362 va entre 1 y 5 MHz. El SPI del ADXL362 funciona en el modo SPI 0 (CPOL = 0 y CPHA = 0) y utiliza una estructura multi-byte en la que el primer byte indica si la comunicación realiza una lectura de registro (0x0B) o una escritura de registro (0x0A).

A continuación, la secuencia de eventos del protocolo SPI:

```
<CS down> <Write/Read (0x0A/0x0B)> <address byte> <data byte> <CS up>
```

Figura 6 y la Figura 7 ilustran dos ejemplos de la comunicación entre el controlador SPI (controlador) y el acelerómetro (periférico): Figura 6 muestra la lectura de un registro y Figura 7 la escritura de un registro.

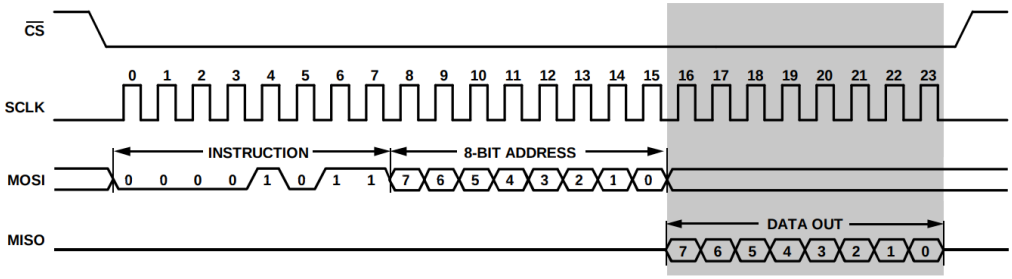

Figura 6. Lectura de registro

(Figura extraída de https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf)

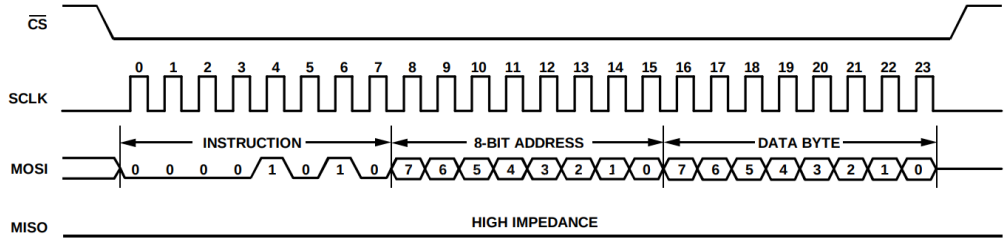

Figura 7. Escritura de registro

(Figura extraída de https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf)

Tabla 5 muestra los registros disponibles en el acelerómetro ADXL362. Para una descripción completa de los registros, consulte la hoja de datos del ADXL362: https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf.

Reg	Name	Bits	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset	RW
0x00	DEVID_AD	[7:0]	DEVID_AD[7:0]								0xAD	R
0x01	DEVID_MST	[7:0]	DEVID_MST[7:0]								0x1D	R
0x02	PARTID	[7:0]	PARTID[7:0]								0xF2	R
0x03	REVID	[7:0]	REVID[7:0]								0x01	R
0x08	XDATA	[7:0]	XDATA[7:0]								0x00	R
0x09	YDATA	[7:0]	YDATA[7:0]								0x00	R
0x0A	ZDATA	[7:0]	ZDATA[7:0]								0x00	R
0x0B	STATUS	[7:0]	ERR_USER_REGS	AWAKE	INACT	ACT	FIFO_OVER-RUN	FIFO_WATER-MARK	FIFO_READY	DATA_READY	0x40	R
0x0C	FIFO_ENTRIES_L	[7:0]	FIFO_ENTRIES_L[7:0]								0x00	R
0x0D	FIFO_ENTRIES_H	[7:0]	UNUSED						FIFO_ENTRIES_H[1:0]		0x00	R
0x0E	XDATA_L	[7:0]	XDATA_L[7:0]								0x00	R
0x0F	XDATA_H	[7:0]	SX				XDATA_H[3:0]				0x00	R
0x10	YDATA_L	[7:0]	YDATA_L[7:0]								0x00	R
0x11	YDATA_H	[7:0]	SX				YDATA_H[3:0]				0x00	R
0x12	ZDATA_L	[7:0]	ZDATA_L[7:0]								0x00	R
0x13	ZDATA_H	[7:0]	SX				ZDATA_H[3:0]				0x00	R
0x14	TEMP_L	[7:0]	TEMP_L[7:0]								0x00	R
0x15	TEMP_H	[7:0]	SX				TEMP_H[3:0]				0x00	R
0x16	Reserved	[7:0]	Reserved[7:0]								0x00	R
0x17	Reserved	[7:0]	Reserved[7:0]								0x00	R
0x1F	SOFT_RESET	[7:0]	SOFT_RESET[7:0]								0x00	W
0x20	THRESH_ACT_L	[7:0]	THRESH_ACT_L[7:0]								0x00	RW
0x21	THRESH_ACT_H	[7:0]	UNUSED					THRESH_ACT_H[2:0]			0x00	RW
0x22	TIME_ACT	[7:0]	TIME_ACT[7:0]								0x00	RW
0x23	THRESH_INACT_L	[7:0]	THRESH_INACT_L[7:0]								0x00	RW
0x24	THRESH_INACT_H	[7:0]	UNUSED					THRESH_INACT_H[2:0]			0x00	RW
0x25	TIME_INACT_L	[7:0]	TIME_INACT_L[7:0]								0x00	RW
0x26	TIME_INACT_H	[7:0]	TIME_INACT_H[7:0]								0x00	RW
0x27	ACT_INACT_CTL	[7:0]	RES		LINKLOOP		INACT_REF	INACT_EN	ACT_REF	ACT_EN	0x00	RW
0x28	FIFO_CONTROL	[7:0]	UNUSED				AH	FIFO_TEMP	FIFO_MODE		0x00	RW
0x29	FIFO_SAMPLES	[7:0]	FIFO_SAMPLES[7:0]								0x80	RW
0x2A	INTMAP1	[7:0]	INT_LOW	AWAKE	INACT	ACT	FIFO_OVER-RUN	FIFO_WATER-MARK	FIFO_READY	DATA_READY	0x00	RW
0x2B	INTMAP2	[7:0]	INT_LOW	AWAKE	INACT	ACT	FIFO_OVER-RUN	FIFO_WATER-MARK	FIFO_READY	DATA_READY	0x00	RW
0x2C	FILTER_CTL	[7:0]	RANGE		RES	HALF_BW	EXT_SAMPLE	ODR			0x13	RW
0x2D	POWER_CTL	[7:0]	RES	EXT_CLK	LOW_NOISE		WAKEUP	AUTOSLEEP	MEASURE		0x00	RW
0x2E	SELF_TEST	[7:0]	UNUSED							ST	0x00	RW

Tabla 5. Registros del acelerómetro ADXL362
(Tabla extraída de https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf)

4. EJERCICIOS BÁSICOS

Ejercicio 1. Crear un programa en l enguaje ensamblador de RISC-V que lea los ocho bits más significativos de los datos de aceleración de los ejes X, Y y Z y que luego muestre esos valores en los displays de 7 segmentos de 8 dígitos. Consulte la sección B para obtener información sobre la configuración y los registros. Utilice las siguientes subrutinas para acceder al módulo SPI. Antes de usar las subrutinas, intente comprenderlas basándose en la información proporcionada en la Sección A sobre el módulo SPI. A continuación, se presenta un breve resumen de cada subrutina:

- Función `spiInit`: Inicializa el módulo SPI.
- Función `spiCS`: Envía el estado del CS al registro SPCS.
- Función `spiCSUp`: Pone la línea CS a alto, invocando la subrutina `spiCS`.
- Función `spiCSDown`: Pone la línea CS a bajo, invocando la subrutina `spiCS`.
- Función `spiSendGetData`: Envía un byte a través de SPI y recupera los datos del periférico.

```
# Register addresses for SPI Peripheral

#define SPCR        0x80001100
#define SPSR        0x80001108
#define SPDR        0x80001110
#define SPER        0x80001118
#define SPCS        0x80001120
```

```
# Function: Initialize SPI peripheral
# call:  by call ra, spiInit
# inputs: None
# outputs: None
# destroys: t0, t1

spiInit:
    li t1, SPCR # control register
    li t0, 0x53 # 01010011 no ints, core enabled, reserved, controller,
                  cpol=0, cha=0, clock divisor 11 for 4096
    sb t0, 0(t1)
    li t1, SPER # extension register
    li t0, 0x02 # int count 00 (7:6), clock divisor 10 (1:0) for 4096
    sb t0, 0(t1)
ret
```

```
# Function: Pull CS Line to either high or low - Provides quick calls spiCSUp
   and spiCSDown
# call:  by call ra, spiCS
# inputs: CS status in a0 (0 is low, 1 is high)
# outputs: None
# destroys: t0

spiCS:
      li t0, SPCS  # CS register
      sb a0, 0(t0) # Send CS status
ret

spiCSUp:
      li a0, 0x00
      j spiCS

spiCSDown:
      li a0, 0xFF
      j spiCS
```

```
# Function: Send byte through SPI and get the peripheral data back
# call:  by call ra, spiSendGetData
# inputs: data byte to send in a0
# outputs: received data byte in a1
# destroys: t0, t1

spiSendGetData:
internalSpiClearIF: # internal clear interrupt flag
      li t1, SPSR  # status register
      lb t0, 0(t1) # clear SPIF by writing a 1 to bit 7
      ori t0,t0,0x80
      sb t0, 0(t1)
internalSpiActualSend:
      li t0, SPDR  # data register
      sb a0, 0(t0) # send the byte contained in a0 to spi
internalSpiTestIF:
      li t1, SPSR  # status register
      lb t0, 0(t1)
      andi t0, t0, 0x80
      li t1, 0x80
```

```
        bne t0,t1,internalSpiTestIF # loop while SPSR.bit7 == 0. (transmission
                                    in progress)
internalSpiReadData:
        li t0, SPDR  # data register
        lb a1, 0(t0) # read the message from SPI
ret
```

5. IMPLEMENTACIÓN DE BAJO NIVEL

A. Implementación de bajo nivel del Acelerómetro SPI

En la primera parte de esta práctica hemos mostrado cómo usar los módulos SPI del Sistema RVfpga, y en esta última parte de la práctica se describe cómo se implementa el módulo SPI en RVfpga. De manera similar al formato de las prácticas anteriores, se divide el análisis del controlador de SPI en tres fases:

1. Conexión física entre el SoC y el acelerómetro (región sombreada a la izquierda en la Figura 8)
2. Integración del controlador SPI, que se incluye dentro del Controlador del Sistema de SweRVolfX (región sombreada en el medio en la Figura 8)
3. Conexión entre el controlador SPI y el core SweRV EH1 (región sombreada a la derecha en la Figura 8)

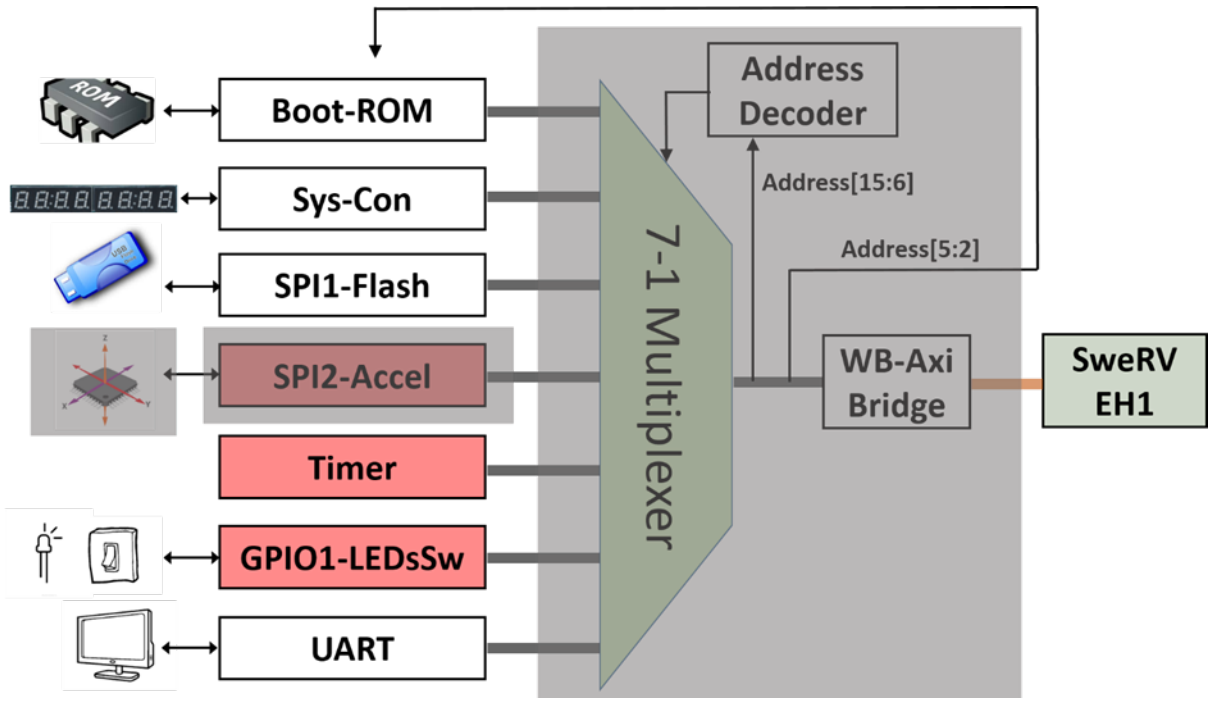

Figura 8. Controlador SPI integrado en el Sistema RVfpga

1. Conexión física del acelerómetro y el SoC

Como con otros periféricos, el archivo de restricciones de RVfpgaNexys debe incluir las conexiones físicas al acelerómetro. El archivo de restricciones del proyecto (*[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc*) define la conexión entre las señales de entrada/salida del SoC y los dispositivos de la placa. Las señales que conectan los cuatro pines del acelerómetro con el SoC se denominan: *o_accel_cs_n*, *o_accel_mosi* (equivalente

a la señal SDO), *i_accel_miso* (equivalente a la señal SDI) y *accel_sclk*. Obsérvese que estas señales se refieren a nombres obsoletos, pero se mantienen para ser coherentes con los nombres usados por el módulo SPI de OpenCores que se usa en el Sistema RVfpga (se puede ver la instanciación de este módulo en la Figura 11). Figura 9 muestra el segmento de código Verilog donde se definen estas 4 conexiones.

```
78  ##Accelerometer
79  set_property -dict { PACKAGE_PIN E15   IOSTANDARD LVCMOS33 } [get_ports { i_accel_miso }]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
80  set_property -dict { PACKAGE_PIN F14   IOSTANDARD LVCMOS33 } [get_ports { o_accel_mosi }]; #IO_L5N_T0_AD9N_15 Sch=acl_mosi
81  set_property -dict { PACKAGE_PIN F15   IOSTANDARD LVCMOS33 } [get_ports { accel_sclk }]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
82  set_property -dict { PACKAGE_PIN D15   IOSTANDARD LVCMOS33 } [get_ports { o_accel_cs_n }];
```

Figura 9. Conexión del SoC y el acelerómetro (archivo *rvfpganexys.xdc*).

En las líneas 52-55 del módulo superior de RVfpgaNexys (es decir, el módulo **rvfpganexys**) se pueden ver estas cuatro señales conectadas al SoC (parte izquierda de la Figura 10), y el final de ese módulo son sus conexiones con el módulo **swervolf_core** (parte derecha de la Figura 10).

```
25  module rvfpganexys
26    #(parameter bootrom_file  = "boot_main.mem")
27     (input wire          clk,
28      input wire          rstn,
29      output wire [12:0] ddram_a,
30      output wire [2:0]  ddram_ba,
31      output wire        ddram_ras_n,
32      output wire        ddram_cas_n,
33      output wire        ddram_we_n,
34      output wire        ddram_cs_n,
35      output wire [1:0]  ddram_dm,
36      inout wire [15:0]  ddram_dq,
37      inout wire [1:0]   ddram_dqs_p,
38      inout wire [1:0]   ddram_dqs_n,
39      output wire        ddram_clk_p,
40      output wire        ddram_clk_n,
41      output wire        ddram_cke,
42      output wire        ddram_odt,
43      output wire        o_flash_cs_n,
44      output wire        o_flash_mosi,
45      input wire         i_flash_miso,
46      input wire         i_uart_rx,
47      output wire        o_uart_tx,
48      inout wire [15:0]  i_sw,
49      output reg [15:0]  o_led,
50      output reg [7:0]   AN,
51      output reg         CA, CB, CC, CD, CE, CF, CG,
52      output wire        o_accel_cs_n,
53      output wire        o_accel_mosi,
54      input wire         i_accel_miso,
55      output wire        accel_sclk);
```

```
248         .o_ram_bready   (cpu.b_ready),
249         .i_ram_rid      (cpu.r_id),
250         .i_ram_rdata    (cpu.r_data),
251         .i_ram_rresp    (cpu.r_resp),
252         .i_ram_rlast    (cpu.r_last),
253         .i_ram_rvalid   (cpu.r_valid),
254         .o_ram_rready   (cpu.r_ready),
255         .i_ram_init_done  (litedram_init_done),
256         .i_ram_init_error (litedram_init_error),
257         .io_data        ({i_sw[15:0],gpio_out[15:0]}),
258         .AN (AN),
259         .Digits_Bits ({CA,CB,CC,CD,CE,CF,CG}),
260         .o_accel_sclk   (accel_sclk),
261         .o_accel_cs_n   (o_accel_cs_n),
262         .o_accel_mosi   (o_accel_mosi),
263         .i_accel_miso   (i_accel_miso));
264
265       always @(posedge clk_core) begin
266         o_led[15:0] <= gpio_out[15:0];
267       end
268
269       assign o_uart_tx = 1'b0 ? litedram_tx : cpu_tx;
270
271   endmodule
```

Figura 10. Conexión del acelerómetro con el módulo superior (archivo *rvfpganexys.sv*).

TAREAS: Siga estas cuatro señales (*o_accel_cs_n*, *o_accel_mosi*, *i_accel_miso* y *accel_sclk*) desde el archivo de restricciones hasta el módulo SoC de SweRVolfX. Necesitará examinar los siguientes archivos:
[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc
[RVfpgaPath]/RVfpga/src/rvfpganexys.sv
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v

2. Integración del módulo SPI2-Accelerometer en el SoC

En las líneas 387-403 del módulo **swervolf_core** (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v*) se instancia el módulo SPI para el acelerómetro (ver Figura 11).

```
382     // SPI for the Accelerometer
383     wire [7:0]           spi2_rdt;
384     assign wb_s2m_spi_accel_dat = {24'd0,spi2_rdt};
385     wire        spi2_irq;
386
387     simple_spi spi2
388       (// Wishbone slave interface
389        .clk_i  (clk),
390        .rst_i  (wb_rst),
391        .adr_i  (wb_m2s_spi_accel_adr[2] ? 3'd0 : wb_m2s_spi_accel_adr[5:3]),
392        .dat_i  (wb_m2s_spi_accel_dat[7:0]),
393        .we_i   (wb_m2s_spi_accel_we),
394        .cyc_i  (wb_m2s_spi_accel_cyc),
395        .stb_i  (wb_m2s_spi_accel_stb),
396        .dat_o  (spi2_rdt),
397        .ack_o  (wb_s2m_spi_accel_ack),
398        .inta_o (spi2_irq),
399        // SPI interface
400        .sck_o  (o_accel_sclk),
401        .ss_o   (o_accel_cs_n),
402        .mosi_o (o_accel_mosi),
403        .miso_i (i_accel_miso));
```

Figura 11. Integración del módulo SPI2-Accelerometer (archivo *swervolf_core.v*).

Como es habitual en los periféricos, la interfaz del módulo puede dividirse en dos bloques: Señales Wishbone (Tabla 6) y señales externas de Entrada/Salida (Tabla 7). Las señales Wishbone permiten al core SweRV EH1 comunicarse con el ADC mediante el protocolo SPI.

Tabla 6. Señales Wishbone

Puerto	Ancho	Dirección	Descripción
cyc_i	1	Entrada	Indica un ciclo de bus válido (selección de core)
adr_i	15	Entradas	Entradas de direcciones
dat_i	32	Entradas	Entradas de datos
dat_o	32	Salidas	Salidas de datos
sel_i	4	Entradas	Indica los bytes válidos en el bus de datos (durante el ciclo válido debe ser 0xf)
ack_o	1	Salida	Salida de *acknowledgment* (indica la terminación normal de la transacción)
err_o	1	Salida	Salida *acknowledgment* de error (indica una terminación anormal de la transacción)
rty_o	1	Salida	No se usa
we_i	1	Entrada	Transacción de escritura cuando su valor es 1
stb_i	1	Entrada	Indica un ciclo de transferencia de datos válido
inta_o	1	Salida	Salida de interrupción

Tabla 7. Señales de Entrada/Salida externas

Puerto	Ancho	Dirección	Descripción
miso_i	1	Entrada	Entrada de datos del controlador - Salida de datos del periférico
mosi_o	1	Salida	Salida de datos del controlador - Entrada de datos del periférico
ss_o	1	Salida	Selección de Chip
sck_o	1	Salida	Reloj del sistema

Como se muestra en la Figura 11, los bits [5:2] de la dirección proporcionada por el core en la señal del bus Wishbone (*wb_m2s_spi_accel_adr[5:2]*) se utilizan para seleccionar uno de los 5 registros SPI disponibles (Tabla 1).

3. Conexión entre el Controlador SPI y el Core SweRV EH1

Como se ha explicado en prácticas anteriores, los controladores de dispositivo se conectan al core SweRV EH1 a través de un multiplexor y un *bridge* (Figura 8). El multiplexor 7:1 (Figura 12) está implementado en el archivo *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v,* que se instancia en las líneas 104-205 del archivo *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh* . Este último archivo se incluye en la línea 168 del módulo **swervolf_core** que se encuentra en: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v*.

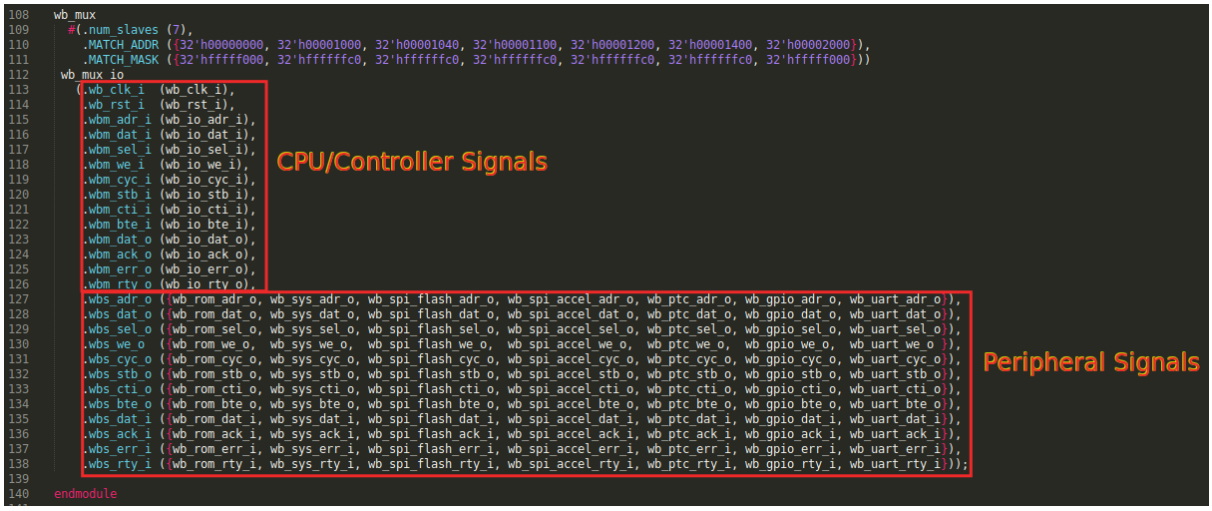

Figura 12. El multiplexor 7-1 selecciona el periférico a conectar con la CPU (*wb_intercon.v*).

El multiplexor selecciona qué periférico leer o escribir, conectando la CPU (señales *wb_io_** - líneas 115-126 de la Figura 12) con el Bus Wishbone de un periférico (líneas 127-138 de la Figura 12), dependiendo de la dirección (líneas 110-111). Por ejemplo, si la dirección generada por la CPU está en el rango 0x80001100-0x8000113F, se selecciona el módulo acelerómetro, y así las señales *wb_io_** se conectan a las señales *wb_spi_accel_**.

6. EJERCICIOS AVANZADOS

Ejercicio 2. El Receptor-Transmisor Asíncrono Universal (UART) es un protocolo de comunicación serie asíncrono. El Sistema RVfpga incluye un módulo UART en su diseño básico (ver Figura 8), para el cual puede encontrar la especificación en: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/uart/docs/UART_spec.pdf*

En primer lugar, analice la implementación de bajo nivel de este módulo en RVfpga, de manera similar a lo que se ha hecho en la Sección A para el Acelerómetro SPI.

Luego, cree un programa en lenguaje ensamblador de RISC-V que imprima un mensaje en su terminal a través del puerto serie. Utilice las siguientes subrutinas para acceder al módulo UART. Antes de usar las subrutinas, intente comprenderlas. A continuación, se

proporciona un breve resumen de cada subrutina:

- Función `uartInit`: Inicializa el módulo UART.
- Función `uartSendByte`: Envía un byte a través de la UART.
- Función `uartSendString`: Envía una cadena de caracteres a través de la UART.

```
# Register addresses for UART Peripheral
# ------------------------------------------------------------------------------

#define CONSOLE_ADDR 0x80001008
#define HALT_ADDR    0x80001009
#define UART_BASE    0x80002000

#define REG_BRDL (4*0x00) /* Baud rate divisor (LSB)        */
#define REG_IER (4*0x01)  /* Interrupt enable reg.          */
#define REG_FCR (4*0x02)  /* FIFO control reg.              */
#define REG_LCR (4*0x03)  /* Line control reg.              */
#define REG_LSR (4*0x05)  /* Line status reg.               */
#define LCR_CS8 0x03   /* 8 bits data size */
#define LCR_1_STB 0x00 /* 1 stop bit */
#define LCR_PDIS 0x00  /* parity disable */

#define LSR_THRE 0x20
#define FCR_FIFO 0x01    /* enable XMIT and RCVR FIFO */
#define FCR_RCVRCLR 0x02 /* clear RCVR FIFO */
#define FCR_XMITCLR 0x04 /* clear XMIT FIFO */
#define FCR_MODE0 0x00 /* set receiver in mode 0 */
#define FCR_MODE1 0x08 /* set receiver in mode 1 */
#define FCR_FIFO_8 0x80  /* 8 bytes in RCVR FIFO */
```

```
.section .data

welcome:
.string "\nHELLO WORLD !!!\n"
```

```
# Function: Initialize UART peripheral
# call:  by call ra, uartInit
# inputs: None
# outputs: None
# overwrites: t0, t1
# ------------------------------------------------------------------------------

uartInit:
       li     t0, UART_BASE

       /* Set DLAB bit in LCR */
       li     t1, 0x80
       sb     t1, REG_LCR(t0)

       /* Set divisor regs */
       li     t1, 27
       sb     t1, REG_BRDL(t0)

       /* 8 data bits, 1 stop bit, no parity, clear DLAB */
       li     t1, LCR_CS8 | LCR_1_STB | LCR_PDIS
       sb     t1, REG_LCR(t0)

       li     t1, FCR_FIFO | FCR_MODE0 | FCR_FIFO_8 | FCR_RCVRCLR | FCR_XMITCLR
       sb     t1, REG_FCR(t0)

       /* disable interrupts  */
       sb     zero, REG_IER(t0)
```

```
        ret
```

```
# Function: Send byte through UART
# call:  by call ra, uartSendByte
# inputs: a0, byte to be sent
# outputs: None
# destroys: t0, t1
# ------------------------------------------------------------------------------

uartSendByte:
    li t1, UART_BASE

    /* Check for space in UART FIFO */
    lb  t0, REG_LSR(t1)
    andi       t0, t0, LSR_THRE
    beqz       t0, uartSendByte
    sb  a0, 0(t1)

    ret
```

```
# Function: Send string through UART (terminated by \0)
# call:  by call ra, uartSendString
# uses: uartSendByte
# inputs: a0, address of first character of string to be sent
# outputs: None
# destroys: t0, t1, t2
# ------------------------------------------------------------------------------

uartSendString:
    li t1, UART_BASE
    add t2,zero,ra # save caller address
    add a1,zero,a0 # use a1 as index
    /* Load first byte */
    lb a0, 0(a1)

internalNextChar:
    call ra, uartSendByte
    addi a1, a1, 1
    lb a0, 0(a1)
    bne a0, zero, internalNextChar

    add ra,zero,t2 # restore caller address
    ret
```

Ejercicio 3. Implemente las siguientes funciones en lenguaje C:

- `char uart_getchar(void)`: Esta función espera a que desde el teclado se envíe un carácter a la placa Nexys A7 a través del puerto UART. A continuación, devuelve este carácter como parámetro de salida. Recuerde que los caracteres se representan en código ASCII (https://www.ascii-code.com/).

- `int uart_putchar(char c)`: Esta función recibe un carácter como argumento de entrada y lo muestra en la consola serie a través del puerto UART. Deberá implementar su propia función para acceder a los registros del UART y no utilizar la función `printfNexys` proporcionada por el BSP.

- `int SevSegDispl(char c)`: Esta función recibe un carácter como argumento de entrada y lo muestra en el dígito de más a la derecha de los displays de 7 segmentos. El resto de los dígitos se desplazan una posición a la izquierda (el de más a la izquierda simplemente se pierde). Dado que los displays de 7 segmentos

solo muestran los caracteres 0 a 9, A, B, C, D, E y F, para cualquier otro carácter de entrada se debe mostrar un 0. Puede extender esta función para que pueda mostrar cualquier carácter utilizando el controlador implementado en el Ejercicio 3 de la Práctica 7.

Observe que para implementar las dos primeras funciones debe hacer uso del documento de especificaciones del módulo UART, disponible en la siguiente ruta:

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/uart/docs/UART_spec.pdf

Basándose en estas tres funciones, cree un programa en C que reciba un carácter desde el teclado y lo muestre en el terminal serie y en los displays de 7 segmentos.

Para inicializar el módulo UART, puede utilizar la función `uartInit` proporcionada en el BSP de WD.

Ejercicio 4. Otro protocolo común de comunicación serie es el denominado I2C (se pronuncia "I dos C" o también "I cuadrado C"). El sensor de temperatura de la placa Nexys A7 utiliza este protocolo. Amplíe el Sistema RVfpga para incluir un controlador I2C, conéctelo con el sensor de temperatura ADT7420 de la placa Nexys A7 (https://www.analog.com/media/en/technical-documentation/data-sheets/adt7420.pdf). Luego escriba un programa que se comunique con este nuevo periférico y que muestre la temperatura en los displays de 7 segmentos.