THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 10
Serial Buses

1. INTRODUCTION

In this lab, we first describe how serial buses work and the main features of one of the most typical serial buses currently used, the SPI bus (Section 2). We then focus on the SPI accelerometer available on the Nexys A7 board: we analyse the high-level specification for this peripheral and propose fundamental exercises (Sections 3 and 4), and then we analyse its low-level implementation and propose some advanced exercises (Sections 5 and 6).

2. SERIAL BUSES – THE SPI BUS

Parallel buses send several bits at once, whereas serial buses send one bit at a time. We first compare these two communication schemes and then describe the SPI (serial peripheral interface) protocol, which is one of the most common serial buses currently used. You can find lots of information on the internet for extending your knowledge about this important communication protocol.

As already demonstrated in previous labs, the main purpose of embedded electronics is to connect processors and circuits to create desired functions. In order for processors and circuits to share information, they must share a common communication protocol. Hundreds of communication protocols have been defined to achieve this data exchange, and, in general, they can be separated into two main categories: parallel or serial interfaces.

Parallel interfaces transfer multiple bits in parallel, i.e., at the same time. They require buses (multiple wires) of data. For example, the protocol may transmit eight, sixteen, or more bits at the same time (see Figure 1). They also require a clock to time when new groups of *N* data bits are ready to transfer.

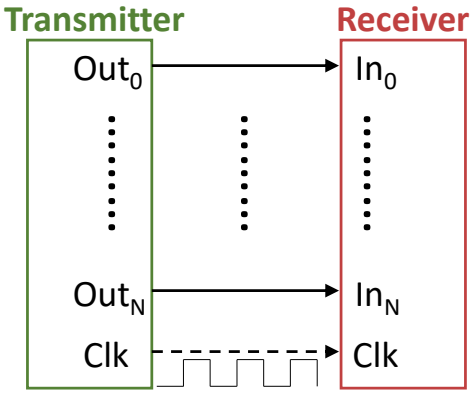


Figure 1. Example of a parallel 8-bit data bus.

In contrast to parallel communication, serial interfaces stream their data one bit at a time. These interfaces can operate using as few as one wire and usually never more than four. Figure 2 shows an example serial interface with one wire for data and one for a clock. At each new clock edge, a new data bit is transferred.

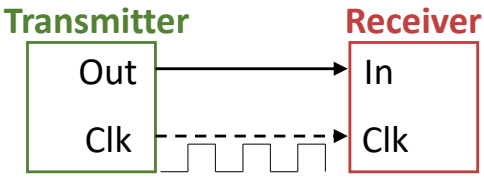


Figure 2. Example of a serial 1-bit data bus.

Parallel communication has the benefits of being fast, straightforward, and relatively easy to implement. However, it requires many more input/output (I/O) lines. So, because pins are limited, embedded systems often opt for serial communication, sacrificing potential speed for pin real estate.

SPI Bus:

The Serial Peripheral Interface (SPI) protocol is one of the most widely used interfaces between microcontroller and peripheral ICs such as sensors, ADCs, DACs, shift registers, SRAM, and others. SPI is a synchronous, full duplex interface based on controller-peripheral (formerly called master-slave) communication.

The SPI bus usually communicates via 4 ports (see Figure 3):

- **SDO** – Serial Data Out: Controller's output to peripheral device
- **SDI** – Serial Data Input: Controller's input from peripheral device
- **SCK** – Serial Clock: Sent from controller to peripheral device
- **CS** – Chip Select: Active low signal; Controller sends signal (0 when peripheral is selected) to peripheral

Note: historically, SDO has also been called MOSI (master data out, slave data in) and SDI has been called MISO (master data in, slave data out). Those terms are outdated and offensive, but they still exist in the literature and in documentation.

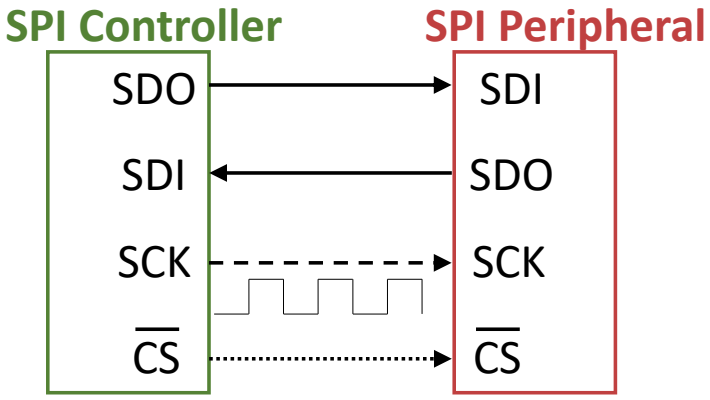


Figure 3. Example of a system with one SPI controller and one SPI peripheral.

The serial data is synchronized to the rising or falling clock edge. SPI is a full-duplex interface; the controller and the peripheral can send data at the same time via the SDO and SDI lines, respectively. SPI interfaces only have one controller, but they may have multiple peripherals. When more than one peripheral is connected, multiple low-asserted chip select signals (CSbar) from the controller are used to select which peripheral is being accessed. SDO and SDI are the serial data lines: SDO (serial data out) is the output data from the controller to the peripheral and SDI (serial data in) is the input data from the peripheral to the controller.

To initiate SPI communication, the controller must select the peripheral device (by asserting the CSbar signal, i.e., CSbar = 0) and then sending the clock signal to the peripheral. During SPI communication, the data is simultaneously transmitted from and to the controller through the SDO and SDI signals, respectively. The serial clock (SCK) edge synchronizes the sampling of the data.

The SPI interface also provides additional signals, CPOL and CPHA, for selecting the idle state of the clock and the phase for sampling the signal. The clock polarity (CPOL) signal is 0 when the clock (SCK) idles at 0 and 1 when it idles at 1. The clock phase (CPHA) signal selects the phase of the clock to send and sample data. When CPHA = 0, data (on SDI or SDO) is sampled on the leading edge (i.e., the first edge after SCK stops idling - and on every cycle thereafter); so data (SDI and SDO) must change on the trailing edge, as shown in the top two timing diagrams of Figure 4. CPHA = 1 does the opposite: data is sampled on the trailing edge and data changes on the leading edge, as shown in the bottom two figures of Figure 4. The edge on which new data is transmitted is also called the *shifting edge*, because this serial communication is typically implemented using a shift register.

The SPI interface we use in this lab is CPHA = 0 and CPOL = 0, so SCK idles low and the controller and peripheral sample data on the rising edge and shift new data onto the line (SDO or SDI) just after each falling edge, as shown in the top timing diagram of Figure 4. Note that when SCK is idle, and just before it rises, SDO and SDI must carry the most significant bit of the next data byte.

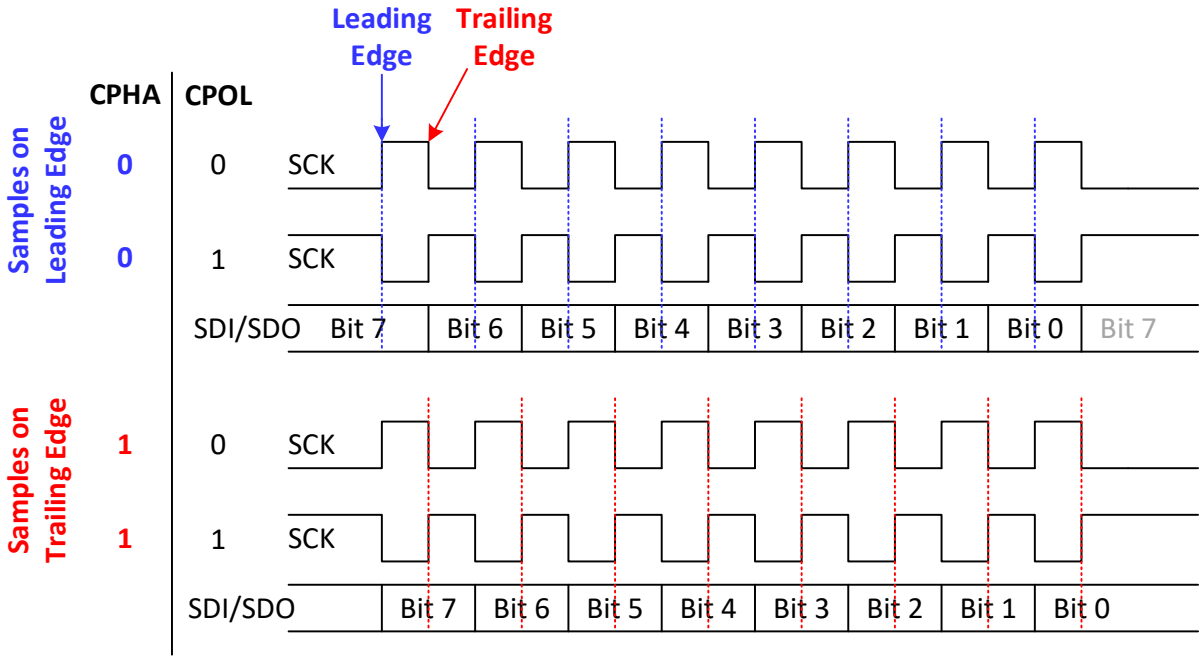


Figure 4. Relationship of CPHA/CPOL with sampling/sending data

3. SPI ACCELEROMETER: HIGH-LEVEL SPECIFICATION

Many peripherals include an SPI interface. For example, the accelerometer on the Nexys A7 board has an SPI interface. In this section we describe the high-level specification of the RVfpga System's SPI controller and introduce the ADXL362 accelerometer included on the Nexys A7 board. We also introduce an exercise that uses the accelerometer.

A. SPI controller specification

The RVfpga System's SPI module is from OpenCores (https://opencores.org/projects/simple_spi). If you download the package, a document is provided that describes the high-level specification of the module. This document is also provided here:

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

We summarize the main operation and features of the SPI module; however, refer to the above document for additional information.

This module has the following main features:

- It is compatible with Motorola's SPI specifications
- It uses the 8-bit WISHBONE RevB.3 classic interface
- It contains a 4-entry read FIFO buffer and a 4-entry write FIFO buffer
- It allows interrupt generation after 1, 2, 3, or 4 transferred bytes
- It can operate with a wide range of input clock frequencies
- It is fully synthesizable

Section 3 of the SPI core specification describes the control and status registers available inside the SPI module, each of which is assigned to a different address (see Table 1). The base address of the SPI controller is **0x80001100**. These registers are described in detail below.

Table 1. SPI Registers

Name	Address	Width	Access	Description
SPCR	0x80001100	8	R/W	Control register
SPSR	0x80001108	8	R/W	Status register
SPDR	0x80001110	8	R/W	Data register
SPER	0x80001118	8	R/W	Extensions register
SPCS	0x80001120	8	R/W	CS register

The SPI Control Register (SPCR) controls the SPI module; Table 2 shows the function of each of its bits.

Table 2. SPCR bits

Bit	Access	Name & Description
0:1	R/W	**SPR** SPI clock Rate: These bits select the SPI clock rate.
2	R/W	**CPHA** Clock Phase: Determines the phase of sampling and sending data. When CPHA = 1, new data is shifted onto the wire at the leading edge and data is sampled on the trailing edge. When CPHA = 0, new data is shifted onto the wire at the trailing edge and sampled on the leading edge.
3	R/W	**CPOL** Clock Polarity: Determines idle state of SPI clock (SCK). When CPOL = 0, SCK idles at 0, when CPOL = 1, SCK idles at 1.
4	R/W	**MSTR** Mode Select: When MSTR = 1, the SPI core is a controller device. This is the only supported mode for this controller.
6	R/W	**SPE** SPI Enable: When SPE = 1, the SPI core is enabled. When it is cleared (SPE = 0), the SPI core is disabled.
7	R/W	**SPIE** SPI Interrupt Enable: When SPIE = 1, when the SPI Interrupt Flag in the status register is set, the host is interrupted.

The SPI Status Register (SPSR) provides the status of the SPI module; Table 3 shows the function of each of its bits.

Table 3. SPSR bits

Bit	Access	Description
0	R/W	**RFEMPTY** Read FIFO Empty: If RFEMPTY = 1, the read FIFO is empty.
1	R/W	**RFFULL** Read FIFO Full: If RFFULL = 1, the read FIFO is full.
2	R/W	**WFEMPTY** Write FIFO Empty: IF WFEMPTY = 1, the write FIFO is empty.
3	R/W	**WFFULL** Write FIFO Full: IF WFFULL = 1, the write FIFO is full.
6	R/W	**WCOL** Write Collision flag: When WCOL = 1, the SPDATA register was written to while the Write FIFO was full. Writing a 1 to WCOL clears this bit.
7	R/W	**SPIF** SPI Interrupt Flag: SPIF = 1 upon completion of a transfer block. If SPIF is asserted ('1') and SPIE is set, an interrupt is generated. Writing a 1 to SPIF clears it.

The SPI Data Register (SPDR) provides the data to read or write. The SPI controller includes 4 x 8-bit Write Buffer and a 4x 8-bit Read Buffer.

The SPI Extended Register (SPER) provides some additional functionality; Table 4 describes the different fields that it contains.

Table 4. SPER bits

Bit	Access	Description
0:1	R/W	**ESPR** Extended SPI Clock Rate Select: Add two bits to the SPR (SPI Clock Rate Select).
6:7	R/W	**ICNT** Interrupt Count: Determine the transfer block size. The SPIF bit is set after ICNT transfers. Thus, it is possible to reduce kernel overhead due to reduced interrupt service calls.

Finally, the SPI Chip Select (SPCS) register selects which peripheral to use. The width of this signal is configurable through parameter SS_WIDTH (SPI Select Width). In the RVfpga System, only one peripheral exists for each SPI interface, so SS_WIDTH = 1.

TASK: Locate the declaration of registers SPCR, SPSR, SPDR, SPER and SPCS in the SPI module, as well as the definition of their addresses. The SPI module is available inside folder *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/spi*.

B. ADXL362 accelerometer specification

The Nexys A7 board includes an Analog Devices ADXL362 accelerometer. You can find the complete information for the device in its data sheet, located here:
https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf

The ADXL362 is a 3-axis MEMS accelerometer that consumes less than 2µA at a 100Hz output data rate and 270 nA when in motion triggered wake-up mode. It provides 12-bit output resolution, although 8-bit formatted data is also provided for more efficient single-byte

transfers when a lower resolution is sufficient. Measurement ranges of ±2 g, ±4 g, and ±8 g are available with a resolution of 1 mg/LSB on the ±2 g range. While the ADXL362 is in Measurement Mode, it continuously measures and stores acceleration data in the X-data, Y-data, and Z-data registers.

The ADXL362 accelerometer includes several registers (Table 5) that allow the user to configure it and to read the acceleration data. The device is configured by writing to the control registers, and the accelerometer data is found by reading the device registers. All communication with the device must specify a register address and a flag that indicates whether the communication is a read or a write. Data transfer occurs after the register address and communication flag are sent to the device.

This accelerometer acts as a peripheral device using an SPI communication scheme. The interface between the FPGA and accelerometer is shown in Figure 5.

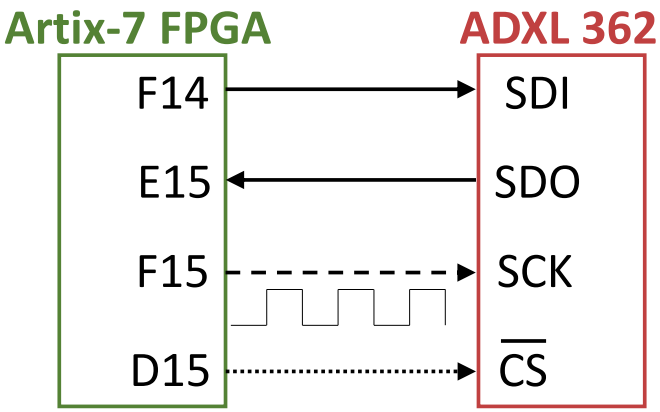


Figure 5. ADXL362 Accelerometer interface with the Nexys A7 board

The recommended SPI clock frequency ranges from 1-5 MHz. The SPI operates in SPI mode 0 (CPOL = 0 and CPHA = 0). The SPI port uses a multibyte structure wherein the first byte indicates if the communication performs a register read (0x0B) or a register write (0x0A):

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

Figure 6 and Figure 7 illustrate two examples of the communication between the SPI controller (controller) and the accelerometer (peripheral): Figure 6 shows the reading of a register and Figure 7 shows the writing of a register.

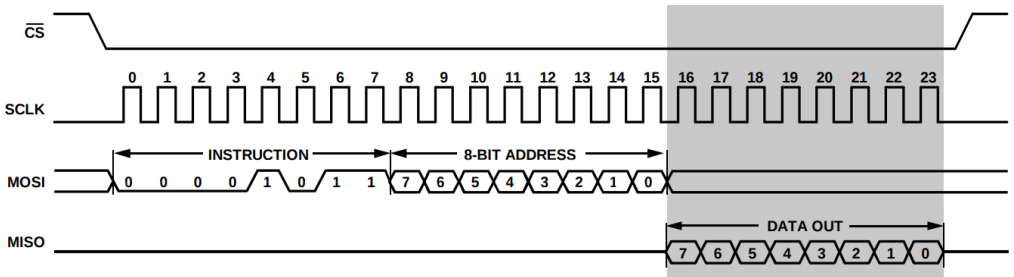


Figure 6. Register read

(Figure from https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf)

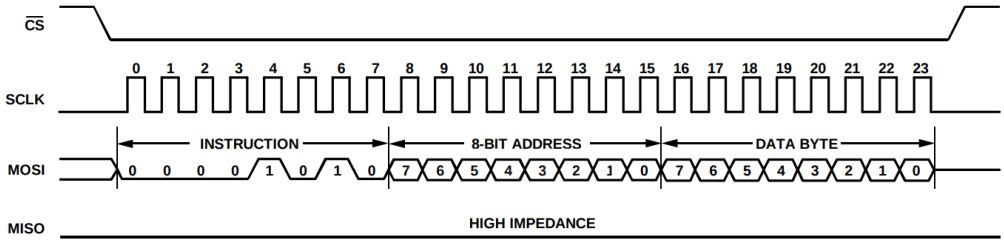


Figure 7. Register write

(Figure from https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf)

Table 5 shows the registers available in the ADXL362 accelerometer. For the complete registers description, refer to the ADXL362 data sheet: https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf.

Table 5. ADXL362 accelerometer registers

(Table from 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

4. FUNDAMENTAL EXERCISES

Exercise 1. Create a RISC-V assembly program that reads the eight most significant bits of the X-axis, Y-axis, and Z-axis acceleration data and then displays those values on the 8-digit 7-Segment Displays. Refer to Section B for configuration and register information. Use the following subroutines to access the SPI module. Before using the subroutines, try to understand them based on the information provided in Section A about the SPI module. Here is a brief summary of each subroutine:

- Function `spiInit`: Initializes the SPI module.
- Function `spiCS`: Send CS status to SPCS register.
- Function `spiCSUp`: Pull CS Line to high, by invoking subroutine `spiCS`.
- Function `spiCSDown`: Pull CS Line to low, by invoking subroutine `spiCS`.
- Function `spiSendGetData`: Send byte through SPI and get the peripheral data back.

```
# 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. LOW-LEVEL IMPLEMENTATION

A. SPI Accelerometer low-level implementation

In the first part of this lab, we showed how to use the RVfpga System's SPI modules, and in this last part of the lab we describe how the SPI module is implemented in RVfpga. Similar to the format from previous labs, we divide the analysis of the SPI controller into three phases:

1. Physical connection between the SoC and the accelerometer (left shadowed region in Figure 8)
2. Integration of the SPI controller, which is included inside the SweRVolfX System Controller (middle shadowed region in Figure 8)
3. Connection between the SPI controller and the SweRV EH1 Core (right shadowed region in Figure 8)

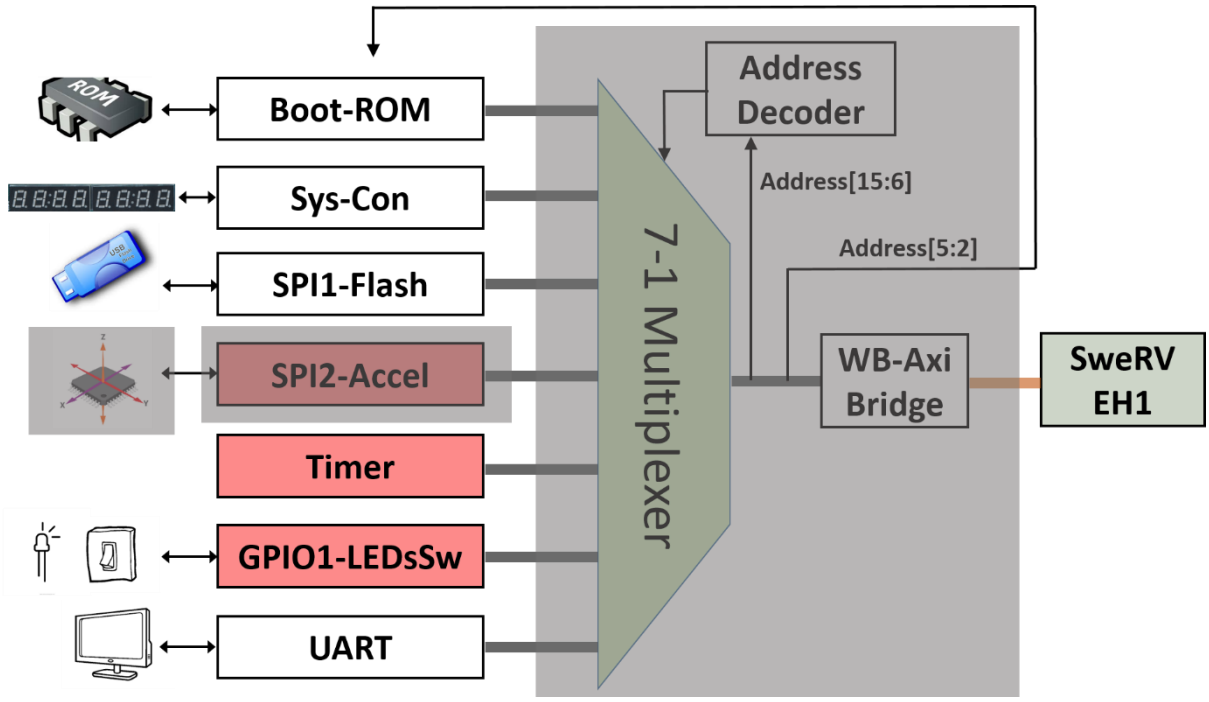


Figure 8. SPI controller integrated into the RVfpga System

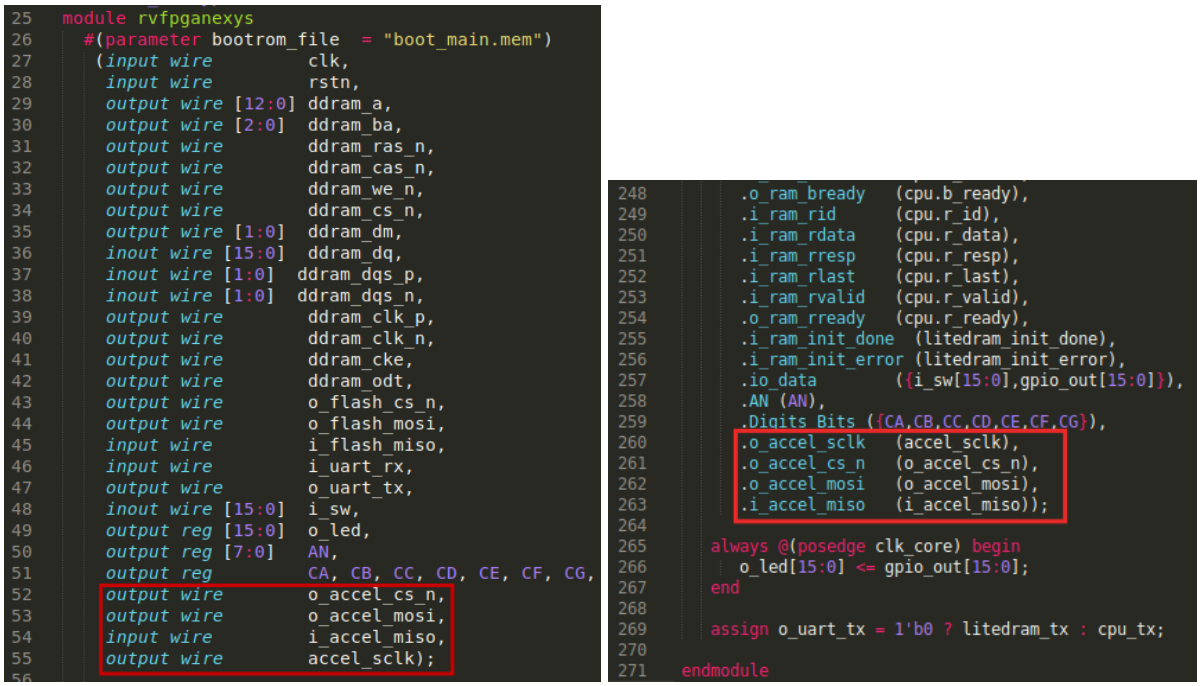
1. Physical connection of the accelerometer and the SoC

As with other peripherals, the RVfpgaNexys constraints file must include the physical connections to the accelerometer. The constraints file of the project (*[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc*) defines the connection between the input/output SoC signals and the board devices. The signals that connect the four pins of the accelerometer with the SoC are called: *o_accel_cs_n*, *o_accel_mosi* (equivalent to signal SDO), *i_accel_miso* (equivalent to signal SDI) and *accel_sclk*. Note that these signals refer to outdated names, but we maintain them in order to be coherent with the names used by the OpenCores's SPI module that we use in the RVfpga System (you can see the instantiation of this module at Figure 11). Figure 9 shows the piece of Verilog code where these 4 connections are defined.

```
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 }];
```

Figure 9. Connection of the SoC and the accelerometer (file *rvfpganexys.xdc*).

In lines 52-55 of the top-module of RVfpgaNexys (i.e., the **rvfpganexys** module) you can see these four signals connected to the SoC (left part of Figure 10), and the end of that module are their connection with the **swervolf_core** module (right part of Figure 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
```

Figure 10. Connection of the accelerometer with the top-module (file *rvfpganexys.sv*).

TASKS: Follow these four signals (*o_accel_cs_n*, *o_accel_mosi*, *i_accel_miso* and *accel_sclk*) from the constraints file to the SweRVolfX SoC module. You will need to inspect the following files:

[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc
[RVfpgaPath]/RVfpga/src/rvfpganexys.sv
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v

2. Integration of the SPI2-Accelerometer module in the SoC

At lines 387-403 of module **swervolf_core** (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v*) the SPI module for the accelerometer is instantiated (see Figure 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));
```

Figure 11. Integration of the SPI2-Accelerometer module (file *swervolf_core.v*).

As usual with peripherals, the interface of the module can be divided into two blocks: Wishbone signals (Table 6) and external I/O signals (Table 7). The Wishbone signals enable the SweRV EH1 Core to communicate with the ADC using the SPI protocol.

Table 6. Wishbone Signals

Port	Width	Direction	Description
cyc_i	1	Inputs	Indicates valid bus cycle (core select)
adr_i	15	Inputs	Address inputs
dat_i	32	Inputs	Data inputs
dat_o	32	Outputs	Data outputs
sel_i	4	Inputs	Indicates valid bytes on data bus (during valid cycle it must be 0xf)
ack_o	1	Output	Acknowledgment output (indicates normal transaction termination)
err_o	1	Output	Error acknowledgment output (indicates an abnormal transaction termination)
rty_o	1	Output	Not used
we_i	1	Input	Write transaction when asserted high
stb_i	1	Input	Indicates valid data transfer cycle
inta_o	1	Output	Interrupt output

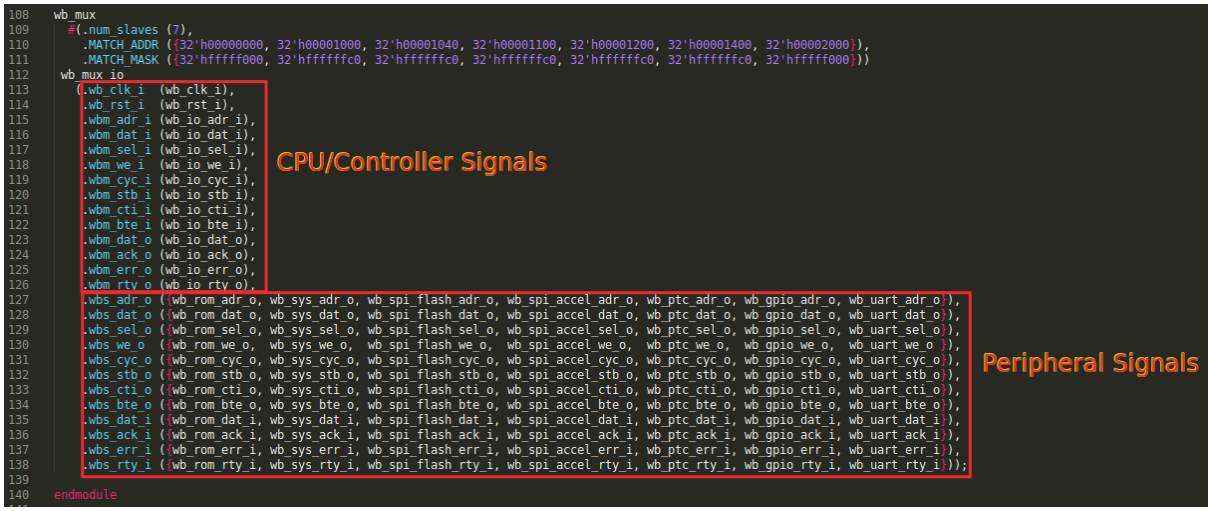
Table 7. External I/O Signals

Port	Width	Direction	Description
miso_i	1	Input	Controller data Input - Peripheral data Output
mosi_o	1	Output	Controller data Output - Peripheral data Input
ss_o	1	Output	Chip Select
sck_o	1	Output	System clock

As shown in Figure 11, bits [5:2] of the address provided by the core in the Wishbone bus signal (*wb_m2s_spi_accel_adr[5:2]*) are used for selecting one among the 5 available SPI registers (Table 1).

3. Connection between the SPI Controller and the SweRV EH1 Core

As explained in previous labs, the device controllers are connected to the SweRV EH1 Core through a multiplexer and a bridge (Figure 8). The 7:1 multiplexer (Figure 12) is implemented in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v*, which is instantiated in lines 104-205 of file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh*. This latter file is included in line 168 of the **swervolf_core** module located here: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v*.




```
108  wb_mux
109    #(.num_slaves (7),
110      .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400, 32'h00002000}),
111      .MATCH_MASK ({32'hfffff000, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hfffff000}))
112    wb_mux_io
113      (.wb_clk_i  (wb_clk_i),
114       .wb_rst_i  (wb_rst_i),
115       .wbm_adr_i (wb_io_adr_i),
116       .wbm_dat_i (wb_io_dat_i),
117       .wbm_sel_i (wb_io_sel_i),
118       .wbm_we_i  (wb_io_we_i),
119       .wbm_cyc_i (wb_io_cyc_i),
120       .wbm_stb_i (wb_io_stb_i),
121       .wbm_cti_i (wb_io_cti_i),
122       .wbm_bte_i (wb_io_bte_i),
123       .wbm_dat_o (wb_io_dat_o),
124       .wbm_ack_o (wb_io_ack_o),
125       .wbm_err_o (wb_io_err_o),
126       .wbm_rty_o (wb_io_rty_o),
127       .wbs_adr_o ({wb_rom_adr_o, wb_sys_adr_o, wb_spi_flash_adr_o, wb_spi_accel_adr_o, wb_ptc_adr_o, wb_gpio_adr_o, wb_uart_adr_o}),
128       .wbs_dat_o ({wb_rom_dat_o, wb_sys_dat_o, wb_spi_flash_dat_o, wb_spi_accel_dat_o, wb_ptc_dat_o, wb_gpio_dat_o, wb_uart_dat_o}),
129       .wbs_sel_o ({wb_rom_sel_o, wb_sys_sel_o, wb_spi_flash_sel_o, wb_spi_accel_sel_o, wb_ptc_sel_o, wb_gpio_sel_o, wb_uart_sel_o}),
130       .wbs_we_o  ({wb_rom_we_o,  wb_sys_we_o,  wb_spi_flash_we_o,  wb_spi_accel_we_o,  wb_ptc_we_o,  wb_gpio_we_o,  wb_uart_we_o }),
131       .wbs_cyc_o ({wb_rom_cyc_o, wb_sys_cyc_o, wb_spi_flash_cyc_o, wb_spi_accel_cyc_o, wb_ptc_cyc_o, wb_gpio_cyc_o, wb_uart_cyc_o}),
132       .wbs_stb_o ({wb_rom_stb_o, wb_sys_stb_o, wb_spi_flash_stb_o, wb_spi_accel_stb_o, wb_ptc_stb_o, wb_gpio_stb_o, wb_uart_stb_o}),
133       .wbs_cti_i ({wb_rom_cti_o, wb_sys_cti_o, wb_spi_flash_cti_o, wb_spi_accel_cti_o, wb_ptc_cti_o, wb_gpio_cti_o, wb_uart_cti_o}),
134       .wbs_bte_o ({wb_rom_bte_o, wb_sys_bte_o, wb_spi_flash_bte_o, wb_spi_accel_bte_o, wb_ptc_bte_o, wb_gpio_bte_o, wb_uart_bte_o}),
135       .wbs_dat_i ({wb_rom_dat_i, wb_sys_dat_i, wb_spi_flash_dat_i, wb_spi_accel_dat_i, wb_ptc_dat_i, wb_gpio_dat_i, wb_uart_dat_i}),
136       .wbs_ack_i ({wb_rom_ack_i, wb_sys_ack_i, wb_spi_flash_ack_i, wb_spi_accel_ack_i, wb_ptc_ack_i, wb_gpio_ack_i, wb_uart_ack_i}),
137       .wbs_err_i ({wb_rom_err_i, wb_sys_err_i, wb_spi_flash_err_i, wb_spi_accel_err_i, wb_ptc_err_i, wb_gpio_err_i, wb_uart_err_i}),
138       .wbs_rty_i ({wb_rom_rty_i, wb_sys_rty_i, wb_spi_flash_rty_i, wb_spi_accel_rty_i, wb_ptc_rty_i, wb_gpio_rty_i, wb_uart_rty_i}));
139
140  endmodule
```

Figure 12. 7-1 multiplexer selects the peripheral to connect to the CPU (*wb_intercon.v*).

The multiplexer selects which peripheral to read or write, connecting the CPU (*wb_io_** signals – lines 115-126 of Figure 12) with the Wishbone Bus of one peripheral (lines 127-138 of Figure 12), depending on the address (lines 110-111). For example, if the address generated by the CPU is in the range 0x80001100-0x8000113F, the accelerometer module is selected, and thus signals *wb_io_** are connected to signals *wb_spi_accel_**.

6. ADVANCED EXERCISES

Exercise 2. The Universal Asynchronous Receiver-Transmitter (UART) is an asynchronous serial communication protocol. The RVfpga System includes a UART module in its basic design (see Figure 8), for which you can find the specification at: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/uart/docs/UART_spec.pdf*

First, analyse the low-level implementation of this module in RVfpga, similarly to what we have done in Section A for the SPI Accelerometer.

Then, create a RISC-V assembly program that prints a message to the PlatformIO shell through the serial port. Use the following subroutines to access the UART module. Before using the subroutines, try to understand them. Here is a brief summary of each subroutine:

- Function `uartInit`: Initializes the UART module.
- Function `uartSendByte`: Send byte through UART.
- Function `uartSendString`: Send string through 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
```

Exercise 3. Implement the three following functions in the C language:

- `char uart_getchar(void)`: This function waits for the keyboard to send a character through the UART to the Nexys A7 board and then returns this character as an output parameter. Remember that characters are represented in ASCII code (https://www.ascii-code.com/).

- `int uart_putchar(char c)`: This function receives a character as an input argument and displays it on the serial console through the UART. You have to implement your own function that accesses the UART registers instead of using the `printfNexys` function provided by WD's BSP (Western Digital's board support package).

- `int SevSegDispl(char c)`: This function receives a character as an input argument and displays it on the right-most digit of the 7-segment displays, shifting the remaining digits one position to the left (the left-most digit is lost). Given that the 7-segment displays only display the characters 0 to 9, A, B, C, D, E and F, for any other character you can simply display a 0. You could extend this exercise to show more characters by using the 7-segment display extended controller implemented in Lab 7 – Exercise 3.

Note that to implement the first two functions you must use the UART module specification document, available at:
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/uart/docs/UART_spec.pdf

Based on the three functions above, create a program in C that receives a character from the keyboard and displays it on both the serial terminal and on the 7-Segment Displays.

For initializing the UART module, you can use the `uartInit` function provided by WD's BSP.

Exercise 4. Another common serial communication protocol is called I2C (pronounced “eye two see” or also “eye squared see”). The temperature sensor on the Nexys A7 board uses this protocol. Expand the RVfpga System to include an I2C controller, and connect it with the Nexys A7 board’s ADT7420 temperature sensor (https://www.analog.com/media/en/technical-documentation/data-sheets/adt7420.pdf). Then write a program that communicates with this new peripheral and displays the temperature on the 7-segment displays.