



Imagination大学计划

# RVfpga实验10

## 串行总线

## 1. 简介

在本实验中，我们首先介绍串行总线的工作方式，以及目前使用的其中一种最典型的串行总线“SPI总线”的主要特性（第2部分）。然后，我们重点介绍Nexys A7开发板上的SPI加速计：首先分析该外设的高级规范并辅以基本练习（第3和第4部分），然后分析该外设的底层实现并辅以一些高级练习（第5和第6部分）。

## 2. 串行总线 - SPI总线

并行总线一次发送多位，串行总线一次发送一位。我们首先比较这两种通信方案，然后介绍目前最常用的其中一种串行总线，即SPI（串行外设接口）协议。关于这一重要的通信协议，网上有许多相关信息，可以帮助您加深了解。

根据先前实验中的演示可知，嵌入式电子产品的主要用途在于连接处理器和电路以实现所需的功能。为了使处理器和电路共享信息，二者必须采用相同的通信协议。针对这类数据交换而定义的通信协议多达数百个，但大体上主要分为以下两种：并行接口或串行接口。

并行接口一次可以同时传输多位，因此需要多条数据总线（线路）。举例来说，该协议可以同时传输8位、16位甚至更多（参见图1）。此外，这类接口需要依靠时钟来确定何时准备好传输下一组 $N$ 个数据位。

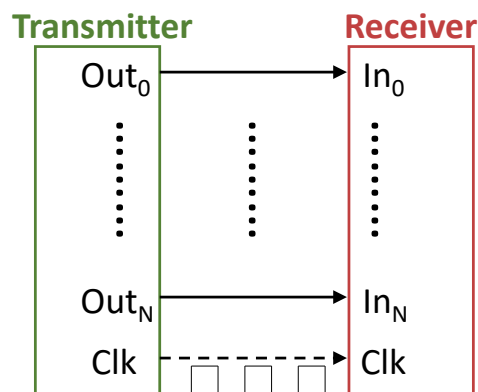


图1. 8位并行数据总线示例

与并行通信不同的是，串行接口一次只能传输一个数据位。串行接口使用一条线路即可工作，通常不会超过四条。图2给出了一个串行接口示例，其中包含一条数据线路和一条时钟线路。每个时钟边沿传输一个数据位。

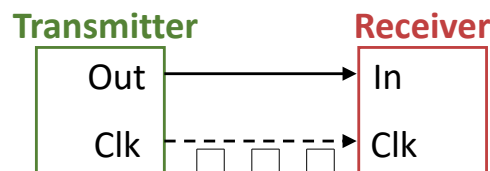


图2. 1位串行数据总线示例

并行通信的优势在于快速、直接且相对易于实现，但是需要的输入/输出（I/O）线路也更多。因此，嵌入式系统往往会由于引脚数量有限而选择串行通信，放弃更高的传输速度。

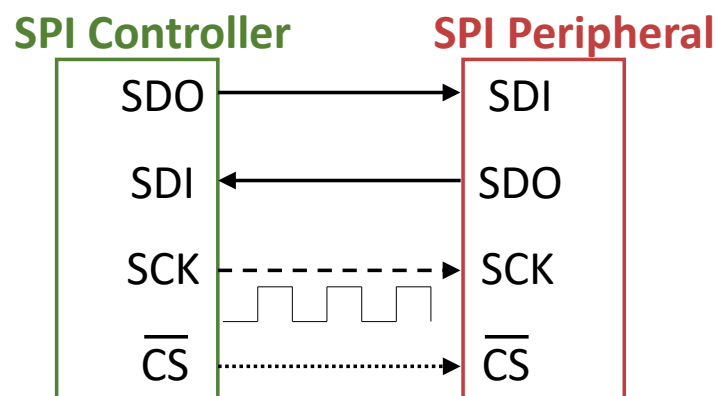
### SPI总线：

串行外设接口（**Serial Peripheral Interface, SPI**）协议是单片机与外设IC（如传感器、ADC、DAC、移位寄存器、SRAM等）之间使用最广泛的通信接口之一。SPI是一种基于控制器-外设（以前称为主-从）通信的同步全双工接口。

SPI总线通常采用4个端口进行通信（参见图3）：

- **SDO** - 串行数据输出：控制器的输出（发送至外设）
- **SDI** - 串行数据输入：控制器的输入（从外设接收）
- **SCK** - 串行时钟：从控制器发送至外设
- **CS** - 片选：低电平有效信号；控制器向外设发送信号（选择外设时为0）

**注：**SDO过去被称为MOSI（主数据输出，从数据输入），SDI过去被称为MISO（主数据输入，从数据输出）。虽然这些旧术语已经过时且遭到禁用，但某些文献和文档中依旧在使用。



**图3. 具有一个SPI控制器和一个SPI外设的系统示例**

串行数据与时钟的上升沿或下降沿同步。SPI属于全双工接口，控制器和外设可以分别通过SDO和SDI线同时发送数据。SPI接口只有一个控制器，但可连接多个外设。当连接多个外设时，控制器可通过发送多个低电平有效的片选信号（CS<sub>bar</sub>）来选择要访问的外设。SDO和SDI是串行数据线：SDO（串行数据输出）是指从控制器向外设输出数据，SDI（串行数据输入）是指从外设向控制器输入数据。

要启动SPI通信，控制器必须选择外设（通过将CS<sub>bar</sub>信号置为有效，即CS<sub>bar</sub> = 0），然后将时钟信号发送到外设。在SPI通信期间，控制器与外设分别通过SDO和SDI信号同时相互传输数据。串行时钟（SCK）边沿用于同步数据采样。

SPI接口还可提供CPOL和CPHA信号，分别用于选择时钟的空闲状态和采样信号的相位。当时钟（SCK）的空闲状态为0时，时钟极性（CPOL）信号为0；当时钟（SCK）的空闲状态为1时，时钟极性（CPOL）信号为1。时钟相位（CPHA）信号用于选择在哪个时钟相位发送和采样数据。当CPHA = 0时，数据在上升沿（即，SCK停止空闲之后的第一个上升沿以及随后每个周期的上升沿）采样（SDI或SDO上的）；因此数据（SDI和SDO）必须在下沿变化，如图4中上面的两个时序图所示。当CPHA = 1时，情况相反：数据在下沿采样，在上升沿变化，如图4中下面的两个时序图所示。传输新数据的边沿也称为移位边沿，因为这种串行通信通常使用移位寄存器来实现。

在本实验中，我们使用的SPI接口配置为CPHA = 0和CPOL = 0，因此SCK在低电平空闲，控制器和外设在上升沿采样数据并在每个下降沿之后将新数据立即移到SDO或SDI线上，如图4中上面的时序图所示。请注意，当SCK空闲且即将上升时，SDO和SDI必须承载下一个数据字节的最高有效位。

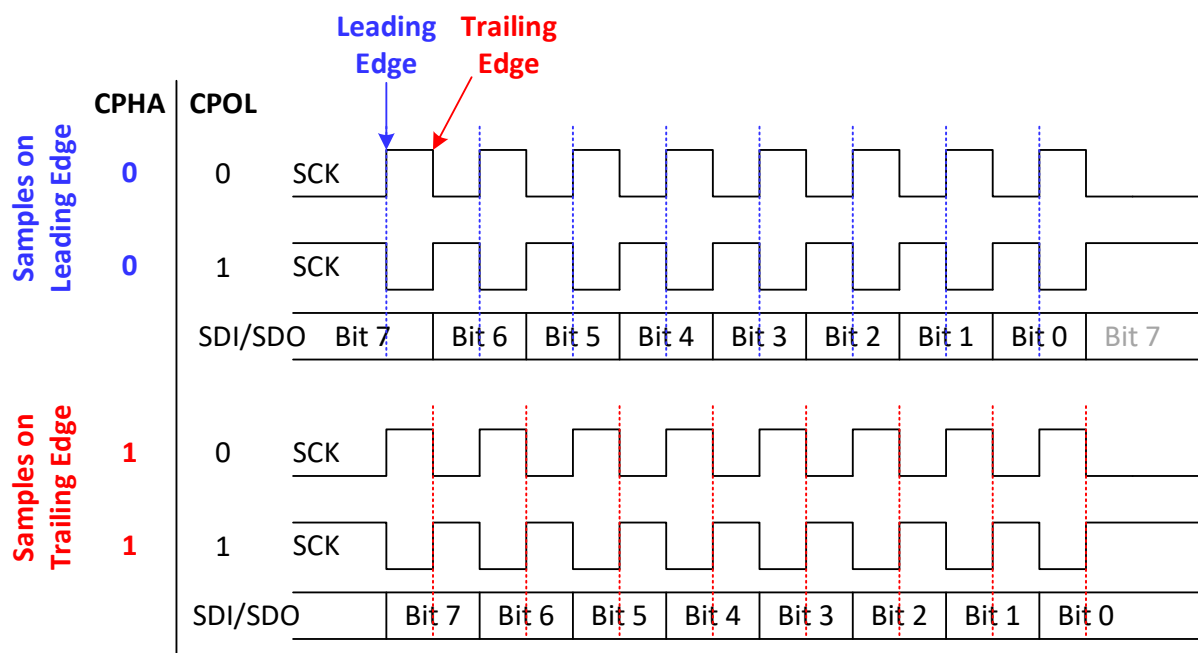


图4. CPHA/CPOL与采样/发送数据的关系

### 3. SPI加速计：高级规范

许多外设都包含SPI接口。例如，Nexys A7板上的加速计包含SPI接口。在本节中，我们将介绍RVfpga系统的SPI控制器的高级规范以及Nexys A7开发板上的ADXL362加速计，并辅以加速计的使用练习。

#### A. SPI控制器规范

RVfpga系统的SPI模块可从OpenCores（[https://opencores.org/projects/simple\\_spi](https://opencores.org/projects/simple_spi)）获取。如果下载教学包，其中会随附一个文档，用于描述该模块的高级规范。该文档也可从以下位置获取：

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

我们总结了SPI模块的主要操作和特性；如需了解详细信息，请参见上述文档。

该模块的主要特性如下：

- 与摩托罗拉的SPI规范兼容
- 使用8位WISHBONE RevB.3经典接口
- 包含一个4项读FIFO缓冲区和一个4项写FIFO缓冲区
- 允许在传输1、2、3或4个字节后产生中断
- 可在较宽范围的输入时钟频率下工作
- 完全可综合

SPI内核规范的第3部分介绍了SPI模块内部可用的控制和状态寄存器，每个寄存器都分配到不同的地址（参见表1）。SPI控制器的基址为**0x80001100**。下文详细介绍了上述寄存器。

**表1. SPI寄存器**

名称	地址	宽度	访问	说明
SPCR	0x80001100	8	R/W	控制寄存器
SPSR	0x80001108	8	R/W	状态寄存器
SPDR	0x80001110	8	R/W	数据寄存器
SPER	0x80001118	8	R/W	扩展寄存器
SPCS	0x80001120	8	R/W	CS寄存器

SPI控制寄存器（SPCR）控制SPI模块；表2列出了该寄存器各个位的功能。

**表2. SPCR位**

位	访问	名称和说明
0:1	R/W	<b>SPR</b> SPI时钟速率：这些位选择SPI时钟速率。
2	R/W	<b>CPHA</b> 时钟相位：确定采样和发送数据的相位。当CPHA = 1时，新数据在上升沿移动到线路上，随后在下降沿进行采样。当CPHA = 0时，新数据在下降沿移动到线路上，随后在上升沿进行采样。
3	R/W	<b>CPOL</b> 时钟极性：确定SPI时钟（SCK）的空闲状态。当CPOL = 0时，SCK的空闲状态为0；当CPOL = 1时，SCK的空闲状态为1。
4	R/W	<b>MSTR</b> 模式选择：当MSTR = 1时，SPI内核充当控制器。这是该控制器支持的唯一模式。
6	R/W	<b>SPE</b> SPI使能：当SPE = 1时，使能SPI内核。清零（SPE = 0）时，禁止SPI内核。
7	R/W	<b>SPIE</b> SPI中断允许：如果SPIE = 1，当状态寄存器中的SPI中断标志置1时，主机将被中断。

SPI状态寄存器（SPCR）提供SPI模块的状态；表3列出了该寄存器各个位的功能。

**表3. SPSR位**

位	访问	说明
0	R/W	<b>RFEMPTY</b> 读FIFO为空：如果RFEMPTY = 1，则读FIFO为空。
1	R/W	<b>RFFULL</b> 读FIFO已满：如果RFFULL = 1，则读FIFO已满。
2	R/W	<b>WFEMPTY</b> 写FIFO为空：如果WFEMPTY = 1，则写FIFO为空。
3	R/W	<b>WFFULL</b> 写FIFO已满：如果WFFULL = 1，则写FIFO已满。
6	R/W	<b>WCOL</b> 写冲突标志：如果WCOL = 1，则表示写FIFO已满时仍在写入SPDATA寄存器。向WCOL写入1可将其清零。
7	R/W	<b>SPIF</b> SPI中断标志：当块完成传输后，SPIF = 1。如果SPIF置为有效（“1”）且SPIE置1，则产生中断。向SPIF写入1可将其清零。

SPI数据寄存器（SPDR）提供要读取或写入的数据。SPI控制器包括4个8位写缓冲区和4个8位读缓冲区。

SPI扩展寄存器（SPER）提供了一些附加功能。表4列出了该寄存器包含的各个位域。

**表4. SPER位**

位	访问	说明
0:1	R/W	<b>ESPR</b> 扩展SPI时钟速率选择：为SPR（SPI时钟速率选择）增加两位。
6:7	R/W	<b>ICNT</b> 中断计数：确定传输块的大小。完成ICNT次传输后，SPIF位将置1。这样可以减少中断服务调用，从而降低内核开销。

SPI片选（SPCS）寄存器选择要使用的外设。该信号的宽度可通过参数SS\_WIDTH（SPI选择宽度）进行配置。在RVfpga系统中，每个SPI接口只有一个对应的外设，因此SS\_WIDTH = 1。

**任务：**在SPI模块中找到寄存器SPCR、SPSR、SPDR、SPER和SPCS的声明及其地址的定义。SPI模块位于文件夹[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/spi内。

## B. ADXL362加速计规范

Nexys A7开发板包括一个模拟器件ADXL362加速计。有关该器件的完整信息，请访问以下位置的数据手册：

<https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>

ADXL362是一款3轴MEMS加速计，在100 Hz输出数据速率下的功耗不到2  $\mu$ A，在运动触发唤醒模式下的功耗为270 nA。该加速计拥有12位输出分辨率，但同时也提供8位格式化数据，以便在低分辨率也可满足要求实现更高效的单字节传输。测量范围为 $\pm 2$  g、 $\pm 4$  g和 $\pm 8$  g，其中 $\pm 2$  g范围内的分辨率为1 mg/LSB。当ADXL362处于测量模式时，将连续测量加速度数据并将其存储在X数据、Y数据和Z数据寄存器中。

ADXL362加速计包括多个寄存器（表5），用户可以使用这些寄存器对其进行配置以及读取加速度数据。写入控制寄存器可以配置加速计，读取设备寄存器可以获取加速计数据。与该器件通信时，必须指定一个寄存器地址以及一个用于指示通信是读操作还是写操作的标志。当寄存器地址与通信标志发送到设备后，即可开始传输数据。

该加速计充当采用SPI通信方案的外设。图图5显示了FPGA与加速计之间的连接。

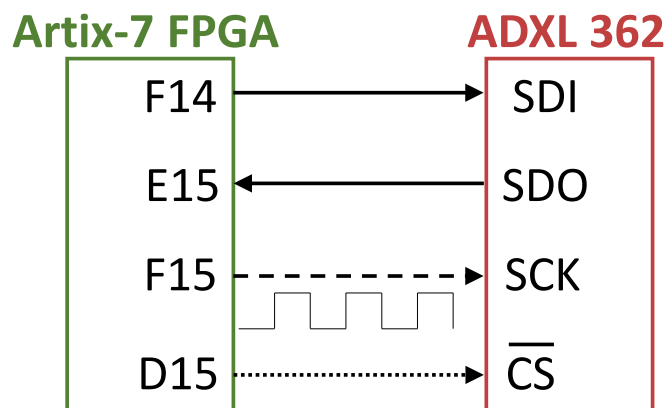


图5. ADXL362加速计与Nexys A7开发板的连接

推荐的SPI时钟频率范围为1-5 MHz。SPI以SPI模式0（CPOL = 0且CPHA = 0）工作。SPI端口采用多字节结构，其中第一个字节指示通信是执行寄存器读操作（0x0B）还是寄存器写操作（0x0A）：

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

图6和图7举例说明了SPI控制器（控制器）与加速计（外设）之间的通信：图6给出了寄存器读操作，图7给出了寄存器写操作。

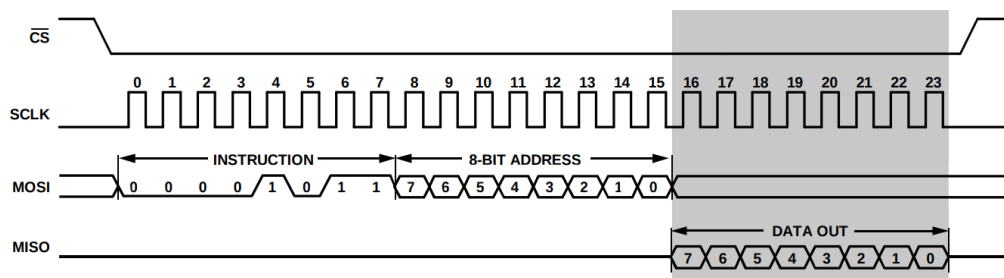


图6. 寄存器读操作

（图片来源：<https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>）



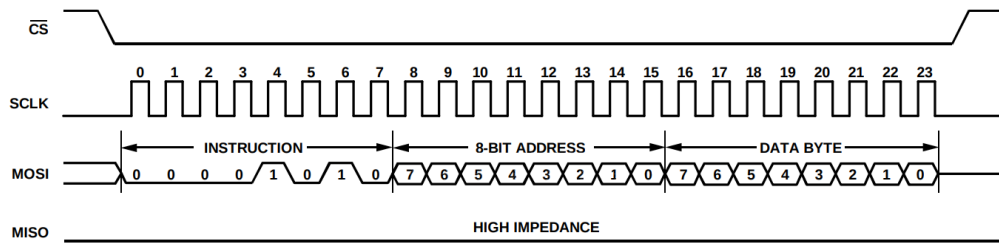


图7. 寄存器写操作

(图片来源: <https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>)

表5列出了ADXL362加速计中提供的寄存器。有关完整的寄存器说明, 请参见ADXL362数据手册: <https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>.

表5. ADXL362加速计寄存器

(表格来源: <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. 基本练习

**练习1.** 创建一个RISC-V汇编程序，该程序读取X轴、Y轴和Z轴加速度数据的高8位，然后在8位7段显示屏上显示这些值。有关配置和寄存器信息，请参阅B部分。使用以下子程序访问SPI模块。在使用这些子程序之前，请尝试根据A部分中提供的SPI模块相关信息来加以理解。以下是每个子程序的简要介绍：

- 函数spiInit: 初始化SPI模块。
- 函数spiCS: 将CS状态发送到SPCS寄存器。
- 函数spiCSUp: 通过调用子程序spiCS将CS线拉为高电平。
- 函数spiCSDown: 通过调用子程序spiCS将CS线拉为低电平。
- 函数spiSendGetData: 通过SPI发送字节并获取外设数据。

```
# 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: 无
```

```
# outputs: 无
```

```
# 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: 无
```

```
# 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. 底层实现

### A. SPI加速计底层实现

在本实验的第一部分中，我们展示了如何使用RVfpga系统的SPI模块，而在本实验的最后一部分，我们将介绍如何在RVfpga中实现SPI模块。本实验的结构编排与之前的实验类似，我们将分三个阶段来分析SPI控制器：

1. SoC和加速计之间的物理连接（图8中的左侧阴影区域）
2. SPI控制器的集成，该控制器包含在SweRVolfX系统控制器中（图8中的中间阴影区域）
3. SPI控制器与SweRV EH1内核之间的连接（图8中的右侧阴影区域）

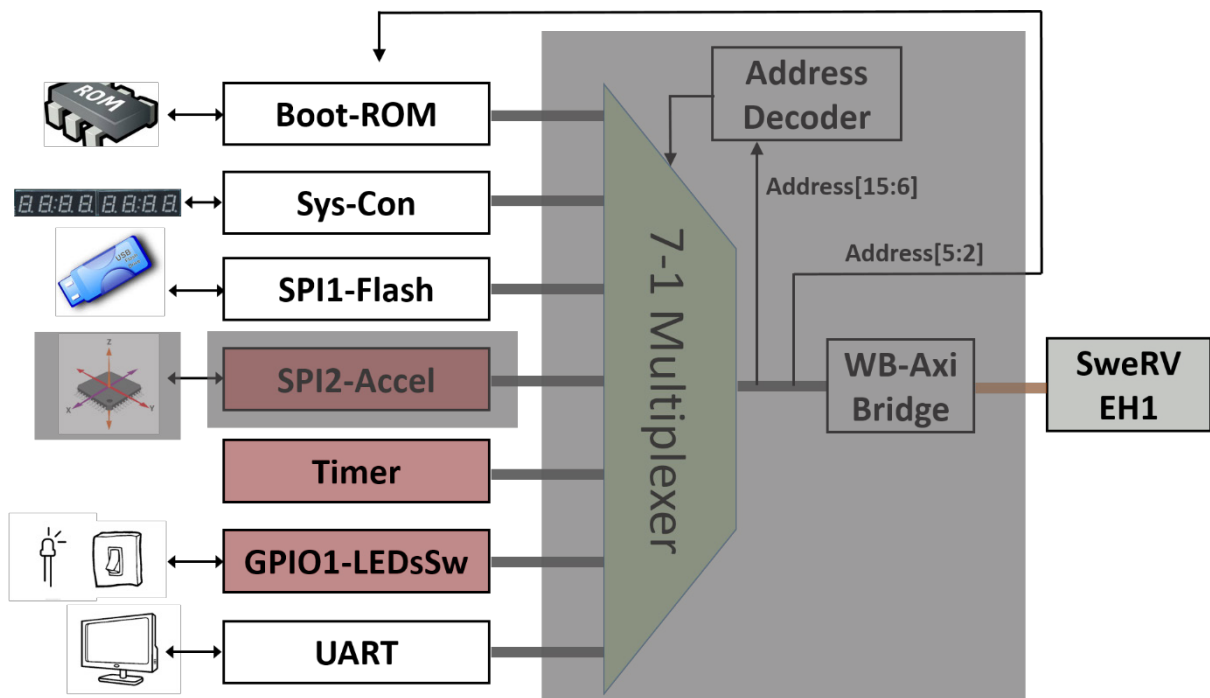


图8. SPI控制器集成到RVfpga系统中

## 1. 加速计与SoC的物理连接

与其他外设一样，RVfpgaNexys约束文件必须包含与加速计的物理连接。项目的约束文件（`[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc`）定义了输入/输出SoC信号与开发板器件之间的连接。用于将加速计的四个引脚与SoC相连的信号分别称为：`o_accel_cs_n`、`o_accel_mosi`（相当于信号SDO）、`i_accel_miso`（相当于信号SDI）和`accel_sclk`。请注意，这些信号的名称已经过时，继续使用的原因是为了与RVfpga系统中的OpenCores SPI模块所使用的名称保持一致（您可以在图11中查看该模块的实例化）。图9给出了定义这4个连接的Verilog代码片段。

```
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 }];
```

图9. SoC与加速计的连接（文件rvfpganexys.xdc）

在RVfpgaNexys顶层模块（即rvfpganexys模块）的第52-55行，您可以看到这四个与SoC连接的信号（图10的左半部分），该模块的末尾是这些信号与swervolf\_core模块的连接（图10的右半部分）。

```

25 module rvfpganexys
26     #(parameter bootrom_file = "boot_main.mem")
27     (input wire      clk,
28      input wire      rstn,
29      output wire [12:0] ddr_a,
30      output wire [2:0] ddr_ba,
31      output wire      ddr_ras_n,
32      output wire      ddr_cas_n,
33      output wire      ddr_we_n,
34      output wire      ddr_cs_n,
35      output wire [1:0] ddr_dm,
36      inout wire [15:0] ddr_dq,
37      inout wire [1:0] ddr_dqs_p,
38      inout wire [1:0] ddr_dqs_n,
39      output wire      ddr_clk_p,
40      output wire      ddr_clk_n,
41      output wire      ddr_cke,
42      output wire      ddr_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);
56
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 = 'b0 ? litedram_tx : cpu_tx;
270
271 endmodule

```

图10. 加速计与顶层模块的连接（文件rvfpganexys.sv）

**任务：** 按照约束文件将这四个信号（`o_accel_cs_n`、`o_accel_mosi`、`i_accel_miso`和`accel_sclk`）与SweRVolfX SoC模块相连。您将需要检查以下文件：

[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc

[RVfpgaPath]/RVfpga/src/rvfpganexys.sv

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf\_core.v

## 2. SPI2加速计模块到SoC的集成

在`swervolf_core`模块（[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf\_core.v）的第387-403行，实例化加速计的SPI模块（参见图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));
404

```

图11. SPI2加速计模块的集成（文件swervolf\_core.v）

与外设一样，模块的接口可以分为两个模块：Wishbone信号（表6）和外部I/O信号（表7）。Wishbone信号允许SweRV EH1内核使用SPI协议与ADC通信。

**表6. Wishbone信号**

端口	宽度	方向	说明
cyc_i	1	输入	指示有效的总线周期（内核选择）
adr_i	15	输入	地址输入
dat_i	32	输入	数据输入
dat_o	32	输出	数据输出
sel_i	4	输入	指示数据总线上的有效字节（在有效周期内，必须为0xf）
ack_o	1	输出	应答输出（指示正常事务终止）
err_o	1	输出	错误应答输出（指示异常事务终止）
rty_o	1	输出	未使用
we_i	1	输入	置为高电平时写事务
stb_i	1	输入	指示有效的数据传输周期
inta_o	1	输出	中断输出

**表7. 外部I/O信号**

端口	宽度	方向	说明
miso_i	1	输入	控制器数据输入 - 外设数据输出
mosi_o	1	输出	控制器数据输出 - 外设数据输入
ss_o	1	输出	片选
sck_o	1	输出	系统时钟

如图11所示，Wishbone总线信号中由内核提供的地址的位[5:2]

（`wb_m2s_spi_accel_adr[5:2]`）用于从5个可用的SPI寄存器中选择1个（表1）。

### 3. SPI控制器与SweRV EH1内核的连接

如先前的实验所述，设备控制器通过多路开关和桥与SweRV EH1内核连接（图8）。7:1多路开关（图12）在文件

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v`中实现，该文件在

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh`文件的第104-205行实例化。后一个文件包含在`swervolf_core`模块的第168行，该模块位于：`[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'hffffff00, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffff00}))
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_o (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

```

CPU/Controller Signals

Peripheral Signals

图12. 7-1多路开关选择与CPU连接的外设 (`wb_intercon.v`)

多路开关选择要读取或写入哪个外设，根据地址（第110-111行）将CPU（`wb_io_*`信号 – 图12的第115-126行）与一个外设的Wishbone总线（图12的第127-138行）连接。例如，如果CPU生成的地址在0x80001100-0x8000113F范围内，则选择加速计模块，从而将信号`wb_io_*`与信号`wb_spi_accel_*`连接。

## 6. 高级练习

**练习2.**通用异步收发器（UART）是一种异步串行通信协议。RVfpga系统的基本设计中包含UART模块（参见图8），该模块的相关规范位于以下位置：

[\[RVfpgaPath\]/RVfpga/src/SweRVolfSoC/Peripherals/uart/docs/UART\\_spec.pdf](#)

首先分析RVfpga中该模块的底层实现，与我们在SPI加速计的A部分中所做的分析类似。

然后，创建一个RISC-V汇编程序，以通过串行端口向PlatformIO shell打印一条消息。使用以下子程序访问UART模块。在使用子程序之前，请首先尝试理解。以下是每个子程序的简要介绍：

- 函数`uartInit`：初始化UART模块。
- 函数`uartSendByte`：通过UART发送字节。
- 函数`uartSendString`：通过UART发送字符串。

```

# 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_RCVRCCLR 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: 无
# outputs: 无
# 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_RCVRCCLR | 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: 无
# 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: 无
# 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
```

### 练习3. 用C语言实现以下三个函数：

- `char uart_getchar(void)`: 该函数等待键盘通过UART向Nexys A7开发板发送一个字符，然后将该字符作为输出参数返回。请记住，字符是用ASCII代码表示的 (<https://www.ascii-code.com/>)。
- `int uart_putchar(char c)`: 该函数接收一个字符作为输入参数，并通过UART将其显示在串行控制台上。必须自行实现访问UART寄存器的函数，而不是使用WD BSP (Western Digital的开发板支持包) 提供的printfNexys函数。
- `int SevSegDispl(char c)`: 该函数接收一个字符作为输入参数，并将其显示在7段显示屏最右边的一位数字上，而剩余几位数字向左移动一位（最左边的一位数字丢失）。鉴于7段显示屏只显示字符0至9、A、B、C、D、E和F，任何其他字符均只能显示0。您可以使用实验7 - 练习3中实现的7段显示屏扩展控制器来扩展此练习以显示更多字符。

请注意，要实现前两个函数，必须使用UART模块规范文档，该文档位于：

**[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/uart/docs/UART\_spec.pdf**

基于上述三个函数，用C语言创建一个程序，该程序从键盘接收一个字符并将其显示在串行终端和7段显示屏上。

要初始化UART模块，可以使用WD BSP提供的uartInit函数。

**练习4.** 另一种常见的串行通信协议称为I2C（发音为“eye two see”或“eye squared see”）。Nexys A7开发板上的温度传感器使用此协议。本练习首先扩展RVfpga系统以包括I2C控制器，并将其与Nexys A7开发板上的ADT7420温度传感器（<https://www.analog.com/media/en/technical-documentation/data-sheets/adt7420.pdf>）相连。然后编写一个程序，与该新外设通信并在7段显示屏上显示温度。