# spi_simple
# Data Sheet

# Content

## Table of Contents



## Figures

# 1 Introduction

This component implements a simple SPI master interface.

The reason for implementing our own SPI master instead of using the Xilinx IP-Core is that the Xilinx IP-Core is very limited in configurability. Especially the limitation to only 8, 16 or 32 bit transfers is not acceptable because many SPI devices required 24 bit transfers.

## 1.1 Purpose

The document here describes this very generic firmware developed for all kinds of projects.

## 1.2 Scope

This document provides a detailed overview of the firmware interface and specifies the user interface.

## 1.3 Definitions, acronyms, and abbreviations

This document is based on the “IEEE Recommended Practice for Software Requirements Specifications” [1].

| | |
|---|---|
| SPI | Serial Peripheral Interface |
| FPGA | Field Programmable Gate Array. Programmable logic device. |

## 1.4 References

[1] IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications.

# 2 IP Description

## 2.1 Vivado Component

The *spi_simple* feature is loaded into a System On Chip (SOC) as a AXI slave component. Different parameters can be configured in a simple GUI.

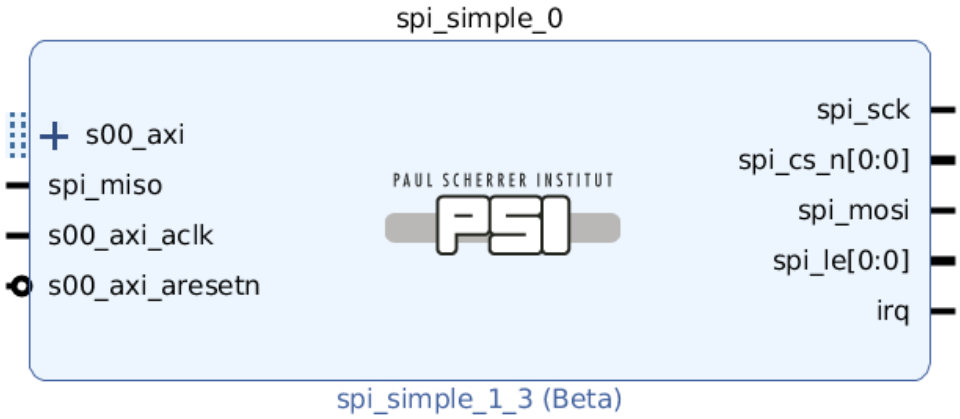


**Figure 1: Component Overview**

All parameters except *CPOL* and *CPHA* are self-explaining. For these parameters, please refer to the description of the SPI interface below.

For the FIFO depth, is recommended to use powers of two for resource efficiency reasons.

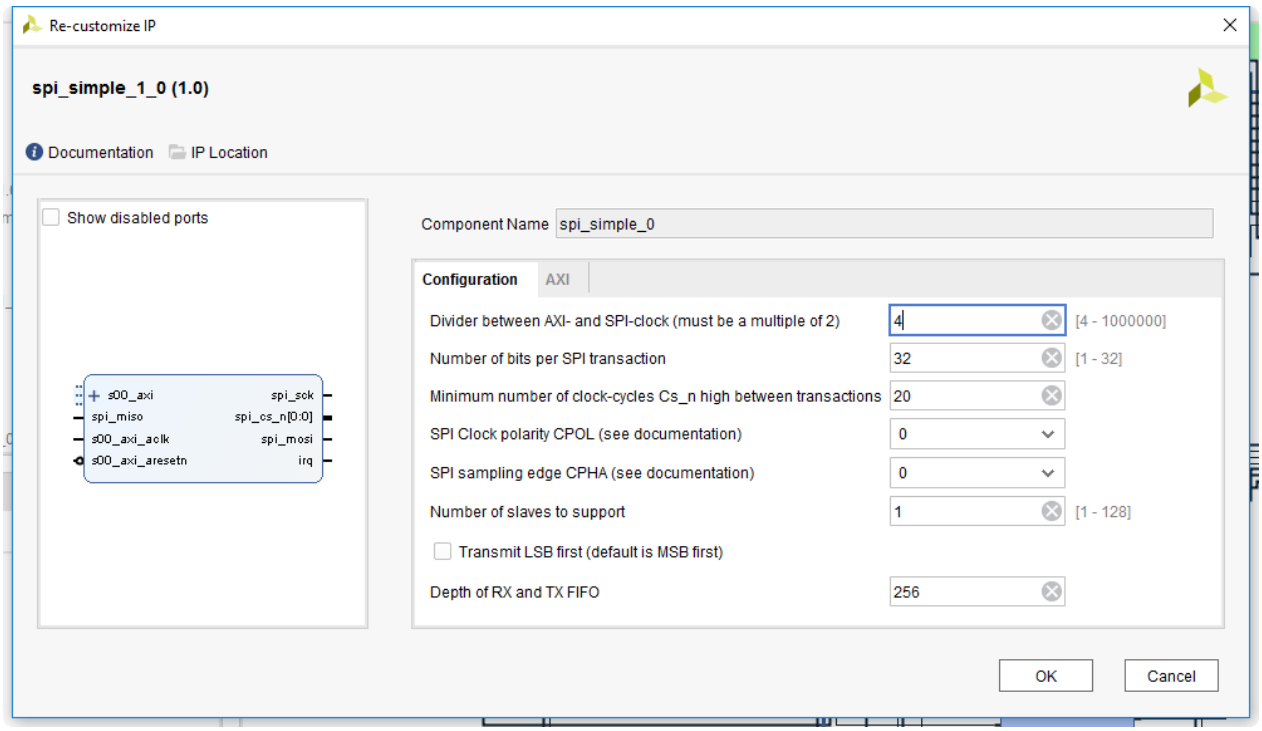


**Figure 2: Component Configuration GUI**

## 2.2 Architecture

The SPI interface is realized using the SPI master implementation from the *psi_common* library. All other logic such as data buffering is implemented in the IP-Core.

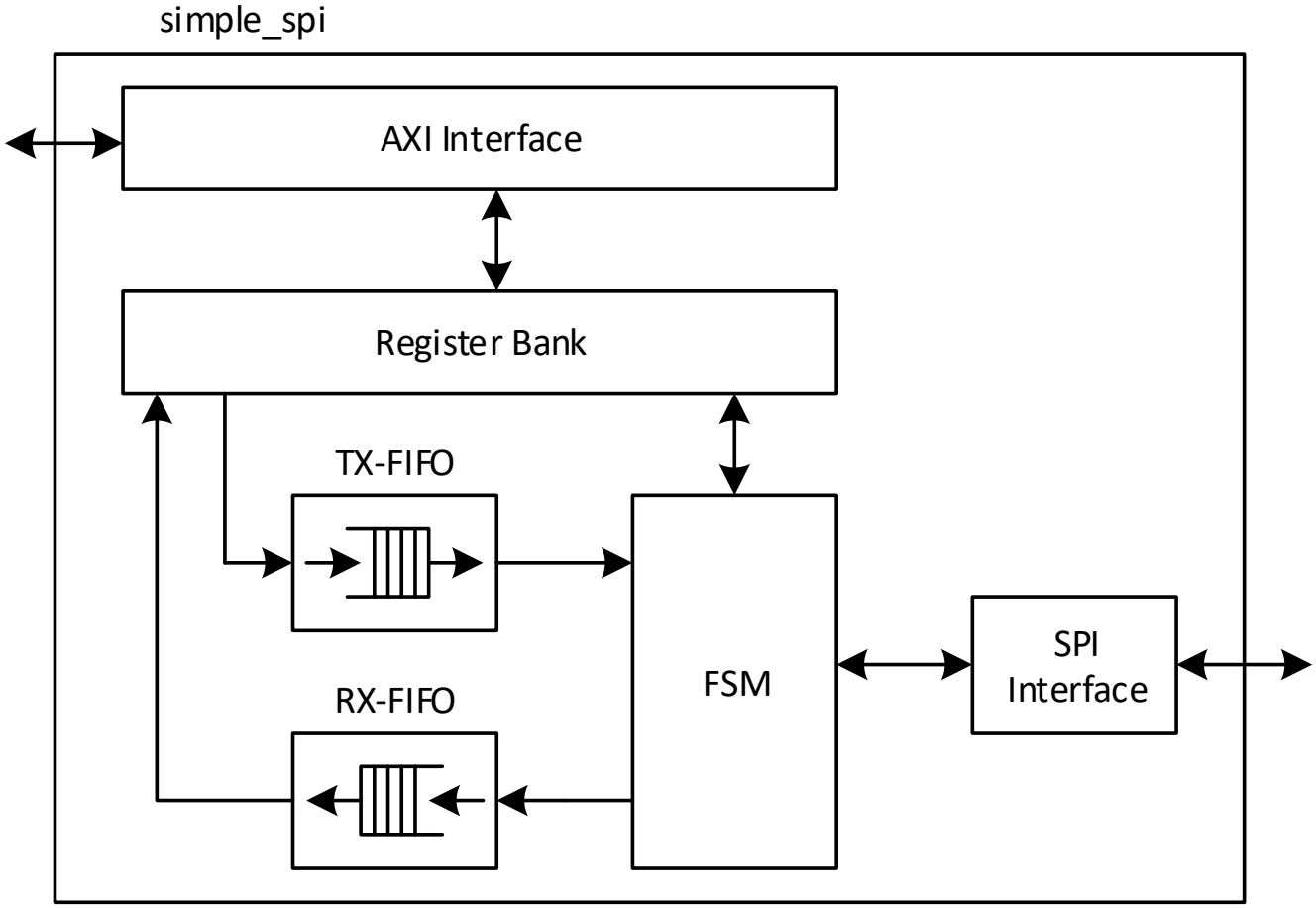


**Figure 3: Architecture Overview**

A transfer is initiated by writing the data to transmit into the TX FIFO. At the time the data is written to the TX FIFO, the information about which slave to access and whether the RX data is kept or not is sampled. All the information is then stored together in the FIFO.

Wheneer the SPI core is ready to execute a transfer and there is one stored in the TX FIFO, the SPI transaction is started. Depending on the information stored in the TX FIFO, the data received is either dropped (for write-only transfers) or stored in the RX FIFO (for read/write transfers).

Additionally to the data-buffering and the SPI logic, the IP-Core allows generating interrupts for several conditions and reading the status of the IP-Core.

## 2.3 Interfaces

### 2.3.1 SPI Interface

The SPI interface is well known and therefore not described in detail here. The only point covered in more detail are the CPOL and CPHA parameters.

The clock and data phase is configurable according to the SPI standard terminology described in the picture below:

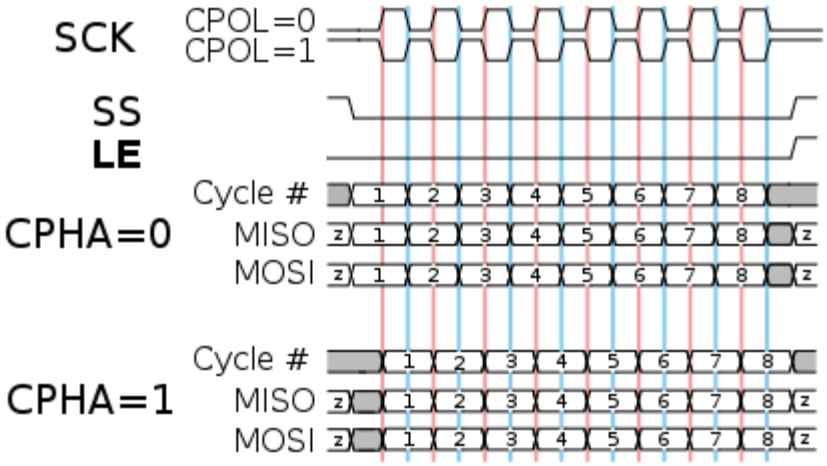


**Figure 4: CPOL and CPHA meaning**

For CPHA = 1, the sampling happens on the second edge (blue) and data is applied on the first edge (red). For CPHA = 0 it is the opposite way.

# 3 Register Bank Description

## 3.1 Configuration

The following short notations for component parameters are used:

N SPI Transfer size in bits
M $\log_2(Depth_{FIFO})$

| Address offset | R/W | Bit | Name | Description |
|---|---|---|---|---|
| 0x00 | R/W | N-1:0 | Data | Read and write accesses to this register will read data from the RX FIFO resp. write data to the TX FIFO. |
| 0x04 | R | 0 | TxEmpty | 1 = TX FIFO is empty |
| | R | 1 | TxFull | 1 = TX FIFO is full |
| | R | 2 | TxAlmEmpty | 1 = TX FIFO is almost empty<br>The threshold to assert this flag can be configured using the *TxAlmEmptyLevel* register. |
| | R | 3 | RxEmpty | 1 = RX FIFO is empty |
| | R | 4 | RxFull | 1 = RX FIFO is full |
| | R | 5 | RxAlmFull | 1 = RX FIFO is almost full<br>The threshold to assert this flag can be configured using the *RxAlmFullLevel* register. |
| | R | 6 | Busy | 1 = SPI Transaction ongoing<br>0 = IP Core is idle<br>This flag is guaranteed to stay set continuously until all transactions stored in the TX FIFO are completed. The flag is not de-asserted between two consecutive transfers. |
| 0x08 | R | M:0 | RxLevel | Number of entries in the RX FIFO |
| 0x0C | R | M:0 | TxLevel | Number of entries in the TX FIFO |
| 0x10 | RW | 7:0 | Slave | Slave number to access. The index of the slave is stored (not a CS_n bitmask). So 0 means slave 0, 1 for slave 1, 2 for slave 2, etc.<br>This register must be set prior to writing data to the *Data* register. After data is written, this register may be changed without affecting the behavior of the transaction. |
| 0x14 | RW | 0 | StoreRx | 1 = Store received data in RX-FIFO<br>0 = Drop reseived data (write only accesses)<br>This register must be set prior to writing data to the *Data* register. After data is written, this register may be changed without affecting the behavior of the transaction. |
| 0x18 | RW | M:0 | TxAlmEmptyLevel | Threshold for the *TxAlmEmpty* condition. |

| | | | | |
|---|---|---|---|---|
| 0x1C | RW | M:0 | RxAlmFullLevel | Threshold for the *RxAlmFull* condition. |
| 0x20 | RCW1 | 6:0 | IrqVec | Interrupt vector. Whenever a certain event occurs, the corresponding bit in this register is set. Each bit can be cleared individually by writing one to it. An IRQ is generated when at least one bit in *IrqVec* is set and the corresponding bit in *IrqEna* is set too. The IRQ signal is held high as long as this condition persists (level sensitive IRQ). |
| | RCW1 | 0 | TxEmpty | The transmit FIFO is empty. This IRQ can be used to dynamically refill the FIFO whenever it is empty. |
| | RCW1 | 1 | TxAlmEmpty | The transmit FIFO is almost empty. This IRQ can be used to dynamically refill the FIFO before it gets empty completely and hence the SPI interface is idle. |
| | RCW1 | 2 | TfDone | This bit gets set whenever a transfer is done. It can be used to react on the execution of individual transactions. |
| | RCW1 | 3 | RxFull | The RX FIFO is full. This IRQ can be used to remove data before the FIFO overflows. |
| | RCW1 | 4 | RxAlmFull | The RX FIFO is almost full. This IRQ can be used to remove data before the FIFO overflows. |
| 0x24 | RW | 6:0 | IrqEna | Interrupt enable register.<br>The bits are the same as in *IrqVec*, so refer to this register for field descriptions. |

**Table 1: Registers**

Normally direct access to the registers is not required because a low-level driver for the IP-Core exists.

# 4 Developer Information

## 4.1 Tools

To work on this core, the following tools are required:

- Vivado
- Modelsim PE

## 4.2 Simulation

A selfchecking testbench including a regression test script exists for this core. To run the regression test, follow the steps below:

1. Open ModelSim
2. Navigate to the “sim” directory
3. Execute “source ./run_tcl.tcl”

The regression script automatically compiles all VHDL files, runs all testbenches and checks if any errors occurred.

For interactive work during development, execute the steps below:

1. Open ModelSim
2. Navigate to the “sim” directory
3. Execute “source ./interactive.tcl”
4. Simulations can be ran either manually or by “run_tb –all”

## 4.3 Packaging

To simplify re-packaging of the IP-Core after changes and avoid trouble with the Vivado GUI, a packaging script was written. To re-package the IP-Core, follow the steps below:

1. Open Vivado
2. In the TCL console, navigate to the “scripts” directory (using “cd <path>”)
3. Execute “source ./package.tcl”