Logic / Home

Serial Peripheral Interface (SPI) Master (VHDL)

Created by Scott Larson, last modified on Oct 30, 2013

Code Download

Version 1.1: spi_master.vhd

Corrected ModelSim simulation error (explicitly reset clk_toggles signal)

Version 1.0: spi_master_v1_0.vhd

Initial Public Release

Features

- VHDL source code of a Serial Peripheral Interface (SPI) master component
- Configurable number of slaves
- Configurable data width
- Selectable polarity and phase
- Selectable speed
- Continuous mode for lengthy data streams

Introduction

This details an SPI master component for use in CPLDs and FPGAs, written in VHDL. The component was designed using Quartus II, version 9.0. Resource requirements depend on the implementation (i.e. the desired number of slaves and data width). Figure 1 illustrates a typical example of the SPI master integrated into a system. An SPI slave component is available here. An SPI 3-wire master component is available here.

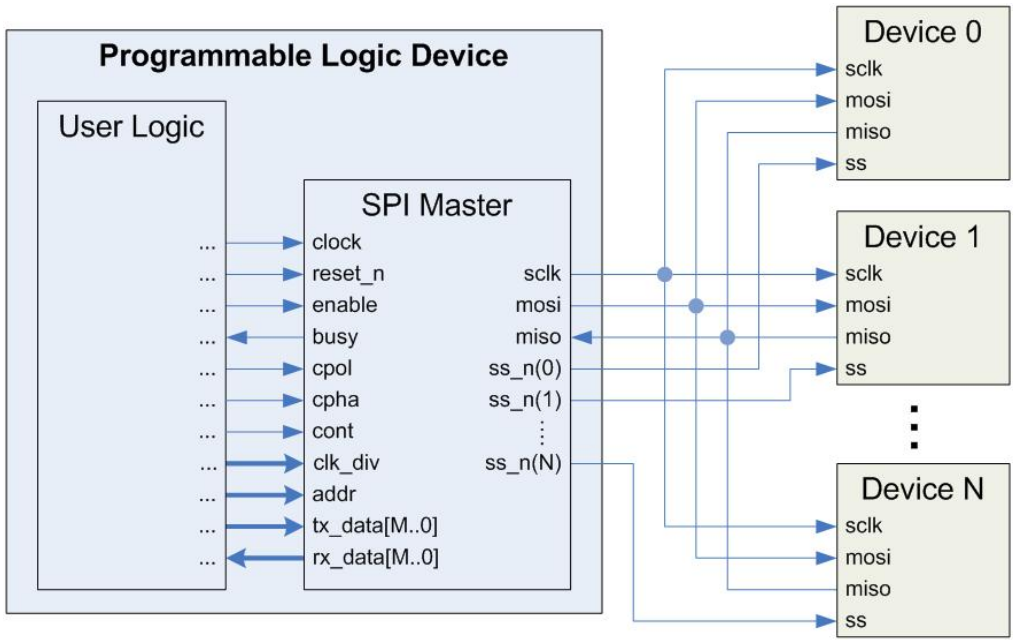

Figure 1. Example Application

Background

An SPI communication scheme is a full-duplex data link, using four wires. The master initiates the transaction by pulling the Slave Select (SS) wire low. A Serial Clock (SCLK) line, driven by the master, provides a synchronous clock source. The master transmits data via the Master Out, Slave In (MOSI) line and receives data via the Master In, Slave Out (MISO) line.

A master can communicates with multiple slaves via a variety of techniques. In the most common configuration, each slave has an independent SS line but shares the SCLK, MISO, and MOSI lines with the other slaves. Each slave ignores the shared lines when its SS line is not pulled low. This topology is shown in Figure 1 above.

SPI has four modes of operation, based on two parameters: clock polarity (CPOL) and clock phase (CPHA). Master and slave must use the same mode to communicate articulately. If CPOL is zero, then SCLK is normally low, and the first clock edge is a rising edge. If CPOL is one, SCLK is normally high, and the first clock edge is a falling edge. CPHA defines the data alignment. If CPHA is zero, then the first data bit is written on the SS falling edge and read on the first SCLK edge. If CPHA is one, data is written on the first SCLK edge and read on the second SCLK edge. The timing diagram in Figure 2 depicts the four SPI modes.

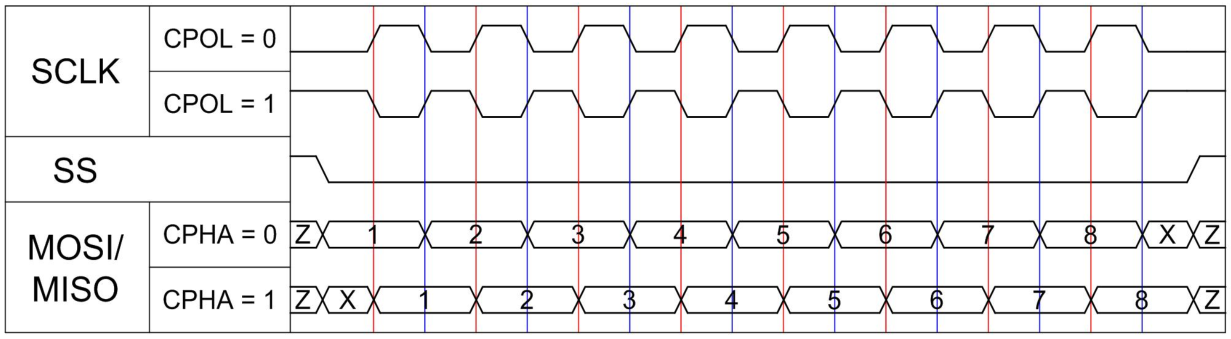

Figure 2. SPI Timing Diagram

Port Descriptions

Table 1 describes the SPI master’s ports. The number of slaves is declared in the ENTITY by the GENERIC parameter *slaves*, and the transmit and receive data bus widths are declared by the GENERIC parameter d_width.

Table 1. Port Descriptions

Port	Width	Mode	Data Type	Interface	Description
clock	1	in	standard logic	user logic	System clock.
reset_n	1	in	standard logic	user logic	Asynchronous active low reset.
enable	1	in	standard logic	user logic	H: latches in settings, address, and data to initiate a transaction, L: no transaction is initiated.
cpol	1	in	standard logic	user logic	SPI clock polarity setting.
cpha	1	in	standard logic	user logic	SPI clock phase setting.
cont	1	in	standard logic	user logic	Continuous mode flag.
clk_div	32	in	integer	user logic	Speed setting. The integer input is the number of system clocks per 1/2 period of sclk.
addr	32	in	integer	user logic	Address of target slave. The slaves are assigned addresses starting with 0.
tx_data	M*	in	standard logic vector	user logic	Data to transmit.
miso	1	in	standard logic	slave devices	Master in, slave out data line.
sclk	1	buffer	standard logic	slave devices	SPI clock.
ss_n	N^	buffer	standard logic vector	slave devices	Slave select signals.
mosi	1	out	standard logic	slave devices	Master out, slave in data line.
busy	1	out	standard logic	user logic	Busy / data ready signal.
rx_data	M*	out	standard logic vector	user logic	Data received from target slave.

Notes
* M is the specified data width, set by the d_width generic
^ N is the specified number of slaves, set by the *slaves* generic

Clocking

The *clock* and *clk_div* inputs define the frequency of SCLK (i.e. the SPI data rate). *clock* is the system clock used to operate the synchronous logic inside the component. The *clk_div* integer input allows the user to set the relative speed at which the current transaction occurs. *clk_div* is the number of *clock* periods between SCLK transitions, as described by Equation 1.

(1)

$$f_{SCLK} = \frac{f_{clock}}{2 \cdot clk\_div}$$

When the *clk_div* port is set to 1, the SCLK frequency is half the *clock* frequency and is set at the maximum achievable data rate. The *enable* pin latches the value of *clk_div* into the component to begin each transaction, so it is possible to adjust the data rate for individual slaves.

Setting the *clk_div* port to a constant value permanently sets the data rate. If *clk_div* is set to 0, the component assumes a value of 1. Therefore, tying the *clk_div* port low configures the component to always operate at maximum speed.

Polarity and Phase

The *enable* pin latches in the standard logic values of *cpol* and *cpha* at the start of each transaction. This allows communication with individual slaves using independent SPI modes. If all slaves require the same mode, *cpol* and *cpha* can simply be tied to the corresponding logic levels.

Transactions

A low logic level on the *busy* output port indicates that the component is ready to accept a command. The component latches the settings, address, and data for a transaction on the first rising edge of *clock* where the *enable* input is asserted. On the following *clock*, the component asserts the *busy* signal and begins performing the transaction. Once complete, the component outputs the received data on the *rx_data* port. This data remains on the port until the component receives new data from a subsequent transaction. The component sets *busy* low to notify the user when the data is available, and the component is immediately ready for another instruction.

Figure 3 shows the timing diagram for a typical transaction. This SPI master is instantiated with four slaves and a four bit data width. It transmits the data “1001” to slave 2, which operates in SPI mode 3 (CPOL = 1, CPHA = 1). The master receives the “1010” data.

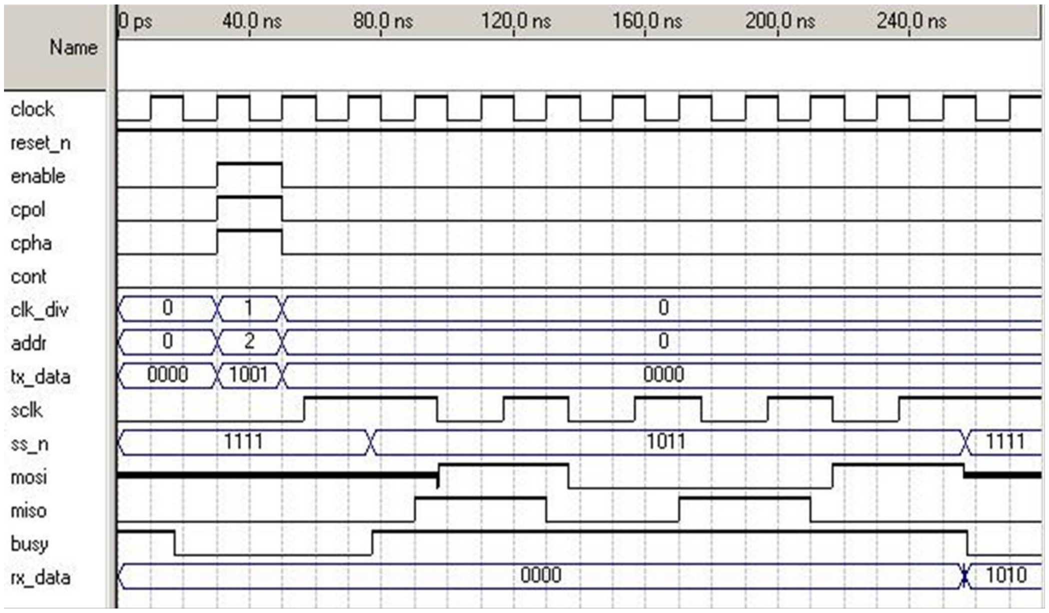

Figure 3. Typical Transaction Timing Diagram

Continuous Mode

During the final receive bit, the SPI master reads the *cont* flag. If low, the component completes the communication as described above. If high, the component latches in data on the *tx_data* port and seamlessly continues the transmission with the new data once the current data is exhausted. When the last bit of the current data finishes (i.e. when the first bit of new data appears on the mosi port), the component outputs the first block of received data on the *rx_data* port and deasserts the *busy* signal for one *clock* cycle to indicate this receive data is available. The receive data remains on the *rx_data* port until the new data communication completes and the component presents the successive block of receive data.

Figure 4 depicts the timing diagram for a transaction using continuous mode. This SPI master is instantiated with four slaves and a two bit data width. Once the component latches the initial data “10,” the *cont* port remains high and the *tx_data* port presents the new data “01.” After the component latches the new data, it sets the *busy* port low, which signals that the *cont* and *tx_data* ports can be cleared and that the first receive data “01” is present on the *rx_data* port. When the transaction completes, the *busy* signal again deasserts, and the new *rx_data* “10” is available.

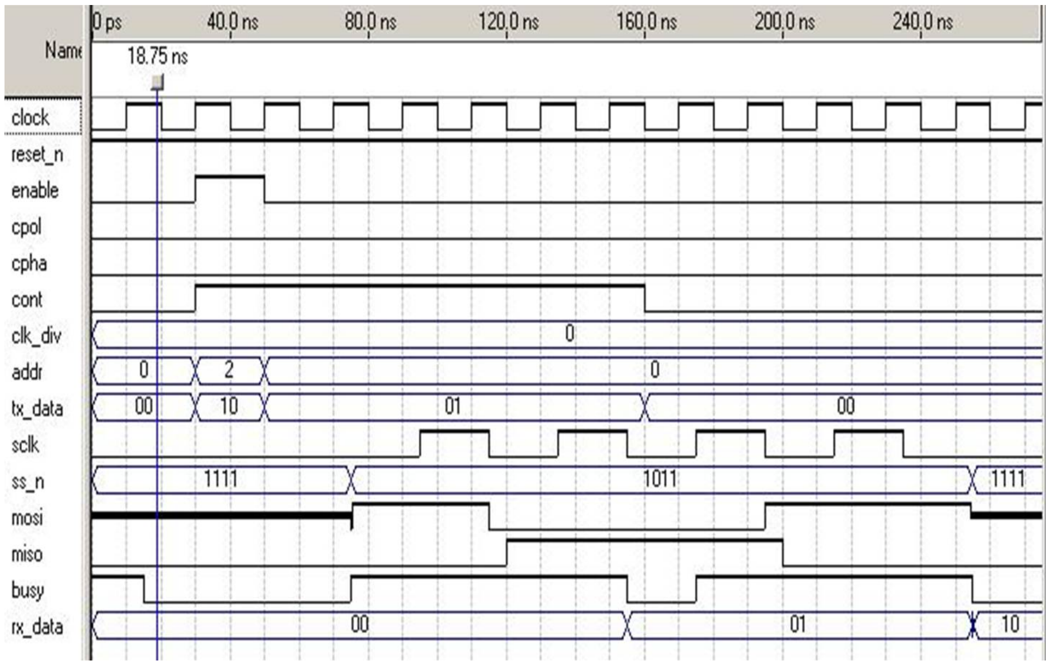

Figure 4. Transaction Timing Diagram with Continuous Mode

Continuous mode can transact very long data streams by being perpetually implemented. Tying the *cont* port low disables continuous mode.

Reset

The *reset_n* input port must have a logic high for the SPI master component to operate. A low signal on this port asynchronously resets the component. During reset, the component holds the *busy* port and all *ss_n* outputs high. The *mosi* output assumes a high impedance state, and the *rx_data* output port clears. Once released from reset, the *busy* port deasserts on the following *clock*, indicating the SPI master's readiness to communicate.

Conclusion

This SPI master is a flexible programmable logic component that accommodates communication with a variety of slaves via a single parallel interface. It allows communication with a user specified number of slaves, which may require independent SPI modes, data widths, and serial clock speeds.

Feedback for Our Sponsor

Please take a few seconds to help us justify the continued development and expansion of the eewiki.

Click on one of our Digi-Key links on your way to search for or purchase electronic components.

Is the eewiki helpful? Comments, feedback, and questions can be sent to eewiki@digikey.com.