Key Design Features

- Synthesizable, technology independent VHDL Core
- SPI™ serial-bus compliant
- Simple programming makes use of a single control register and a single address register
- Architecture allows sustained 8-bit read/write operations
- Configurable number of 8-bit read-write configuration registers and 8-bit read-only status registers
- Configurable clock polarity setting (CPOL)
- Configurable clock phase setting (CPHA)
- SPI signals are treated asynchronously allowing the implementation of single clock domain designs
- Data rates of up to 40 Mbps[1]

Applications

- SPI slave communication
- Inter-chip board-level communications
- Robust communication at higher data rates than other serial protocols such as I2C, UART and USB 1.X

Generic Parameters

Generic name	*Description*	*Type*	*Valid range*
num_config	Number of Configuration registers	integer	2 ≤ regs ≤ 256 (power of 2)
num_config_log2	Log2 number of Configuration registers	integer	Log2 (num_config)
num_status	Number of Status registers	integer	2 ≤ regs ≤ 256 (power of 2)
num_status_log2	Log2 number of Status registers	integer	Log2 (num_status)
cpol	Clock polarity	integer	0, 1
cpha	Clock phase	integer	0, 1

Block Diagram

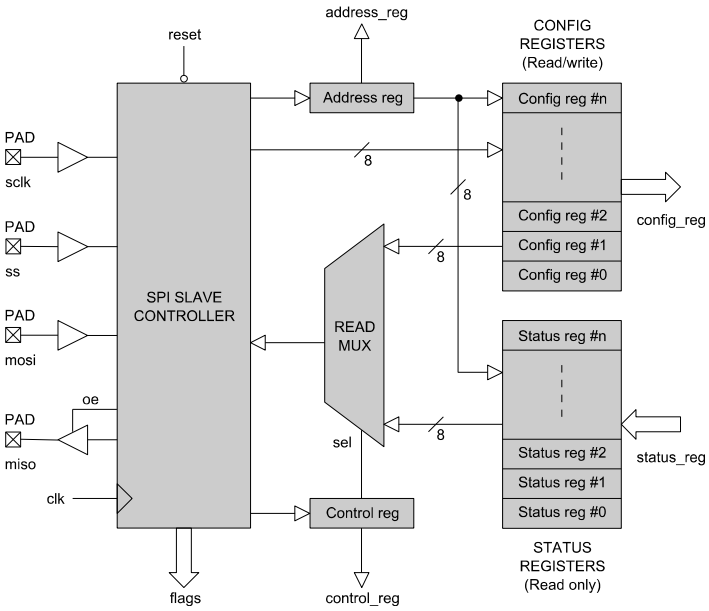

Figure 1: SPI Slave Interface Controller architecture

Pin-out Description

Pin name	*I/O*	*Description*	*Active state*
clk	in	Synchronous clock	rising edge
reset	in	Asynchronous reset	low
sclk	in	SPI™ Serial clock	rising or falling edge[2]
ss	in	SPI™ Serial select	low
mosi	in	SPI™ Master out / Slave in	data
miso	tristate out	SPI™ Master in / Slave out	data
co_flag	out	Control register write	pulse high
ad_flag	out	Address register write	pulse high
wr_flag	out	Config register write	pulse high
rd_flag	out	Config register read	pulse high
ro_flag	out	Status register read	pulse high
control_reg [7:0]	out	Internal control register	data
address_reg [7:0]	out	Internal address register	data
config_reg [num_config*8-1:0]	out	Configuration register output bits	data
status_reg [num_status*8-1:0]	in	External status register input bits	data

1 Maximum attainable data rate will be determined by the choice of system clock frequency and the physical characteristics of the bus

2 Note that the serial clock characteristics are dependent on the CPOL and CPHA settings. See the SPI™ specification for more details

General Description

SPI_SLAVE is an SPI™ compliant slave interface controller. The controller decodes the bus signals and de-serializes them into a series of 8-bit bytes. Communication with the slave controller is by means of a simple data structure - a single control register and a single address register. The control register defines whether the transfer is a read or write and also the type of register to be accessed (config or status). The address register provides an index into the chosen register bank.

Both the config registers and the status registers are directly connected to the external ports of the controller. The config registers provide general purpose read/write bits for the control of an external device. The status registers are read only and allow the state of external pins to be monitored via the SPI interface.

All inputs to the slave interface controller are driven by the bus Master with the exception of *miso* which is a tristate output. The signal *miso* is normally in the high-impedance state unless a read operation is in active progress.

The SPI slave controller is comprised of three main blocks as described by Figure 1. These blocks are the SPI Slave Controller core, the Configuration register bank and the Status register bank.

SPI Slave Controller Core

The slave controller core is a state-machine that continually monitors the state of the SPI signals. An SPI transfer begins with the high-to-low transition of the slave select signal *ss.* Once *ss* is driven low, the controller will sample the next 16-bits from the master at the *mosi* input. Bits are sampled on either the rising or falling edge of *sclk* depending on the clock configuration settings *cpol* and *cpha*.

The first 8-bits in the transfer are written to the internal control register and the next 8-bits are written to the internal address register. Figure 2 shows the programming of the control and address registers in more detail.

CONTROL REGISTER

7	6	5	4	3	2	1	0
U4	U3	U2	U1	U0	INC	C/S	R/W
MSB							LSB

Bit 0 - Read/Write flag 0 = Write, 1 = Read
Bit 1 - Config/Status register select 0 = Config, 1 = Status
Bit 2 - Address auto-increment 0 = Auto, 1 = No auto
Bit 7:3 - User defined control flags

ADDRESS REGISTER

7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0
MSB							LSB

Bit 7:0 - Address of register to access

Figure 2: Control and Address register definitions

Every SPI transfer must begin with a write to the control register and the address register. The *R/W* flag in the control register determines whether the operation is a read or a write. The *C/S* flag determines whether a configuration register or a status register is to be accessed. The *INC* flag when set turns off the address pointer auto-increment function. Bits *U4 to U0* are user defined flags that may be programmed as required. The address register contains the address of the first register to be accessed in the chosen register bank. Once the control and address registers have been written, the next 8 serial clocks are used to synchronize a write to a configuration register or a read from a configuration/status register.

Normally after each 8-bit read or write, the internal address register is incremented by 1 and the master may write or read a further 8-bits. This means that successive back-to-back reads or writes will be performed on the next register in the chosen register bank. Once the maximum address has been reached, the address pointer will wrap around back to 0. Note that the address auto-increment function may be disabled by setting the *INC* flag in the control register to '1'.

Any number of sequential register read or write operations may be performed (to the same register bank). The SPI bus transfer will terminate immediately as soon as *ss* is driven high. If the user wishes to read and write different register banks, the current SPI transfer must be terminated before the next bank is accessed.

The controller state machine generates a series of output flags whenever an 8-bit read or write to one of the internal registers is performed. These flags take the form of a single pulse that lasts for one system clock cycle. The strobes may be used as interrupt or valid flags to indicate that the contents of one of the registers has changed.

Clock Polarity and Phase settings

The generic settings *cpol* and *cpha* determine how the serial data is sampled and changed with respect to the serial clock. These settings are defined in the standard SPI™ specification. The table below shows a brief summary of these settings.

CPOL	CPHA	Description
0	0	Serial clock default state logic '0' Data sampled on rising-edge of serial clock Data changed on falling-edge of serial clock
0	1	Serial clock default state logic '0' Data sampled on falling-edge of serial clock Data changed on rising-edge of serial clock
1	0	Serial clock default state logic '1' Data sampled on falling-edge of serial clock Data changed on rising-edge of serial clock
1	1	Serial clock default state logic '1' Data sampled on rising-edge of serial clock Data changed on falling-edge of serial clock

Configuration Register bank

The configuration registers are organized as a bank of 8-bit general purpose read/write registers that may be accessed via the SPI slave interface. The config registers are designed to be used for the general configuration of devices external to the controller.

The contents of these registers are made available at the output port *config_reg.* This port contains the contents of all the config register bits concatenated together. This means that bits 7..0 represent the contents of config reg #0, bits 15..8 the contents of config reg #1 etc.

The total number of config registers is defined by the generic parameter *num_config*. The total number of configuration registers must be a power of 2 for the register addressing to work correctly.

Status Register bank

The status registers follow exactly the same structure as the configuration registers. The difference is that these registers are read only. Any attempt to write these registers will have no effect other than to perform a dummy transfer on the SPI bus. The status registers are designed to be used for snooping the state of control signals in an external device.

The port *status_reg* contains the contents of all the status register bits concatenated together. This means that bits 7..0 represent the contents of status reg #0, bits 15..8 the contents of status reg #1 etc.

The total number of status registers is defined by the generic parameter *num_status*. The total number of status registers must be a power of 2 for the register addressing to work correctly.

Functional Timing

The following timing diagrams demonstrate the SPI protocol for reading and writing registers in the various register banks. All the examples show SPI mode 0,0 operation – meaning that data is sampled on the rising edge of the serial clock and data changes on the falling edge. The default state of the clock is logic '0'.

Figure 3 shows a write to two consecutive config registers at addresses 0x02 and 0x03. In this particular example the user-defined control flags have also been set in the control register. Auto-increment is set to '0' so that successive writes increment the value in the address pointer.

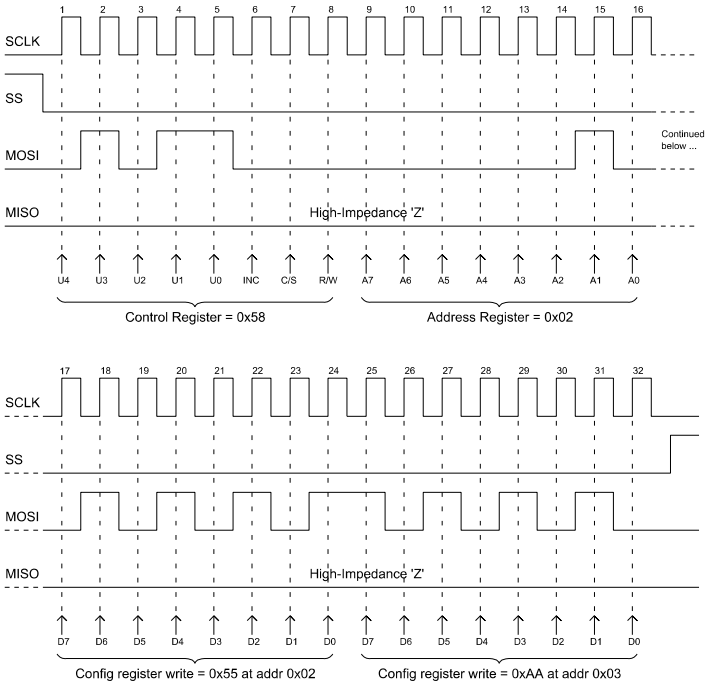

Figure 3: Config register write example

Figure 4 shows the corresponding config register read operation after the previous write example.

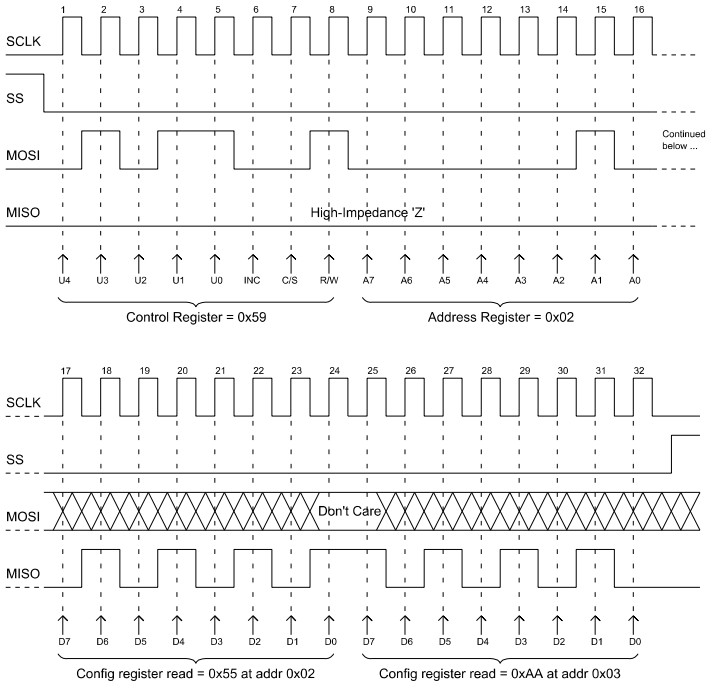

Figure 4: Config register read example

Figure 5 demonstrates a sequential read from registers 0x01 and 0x02 in the status register bank.

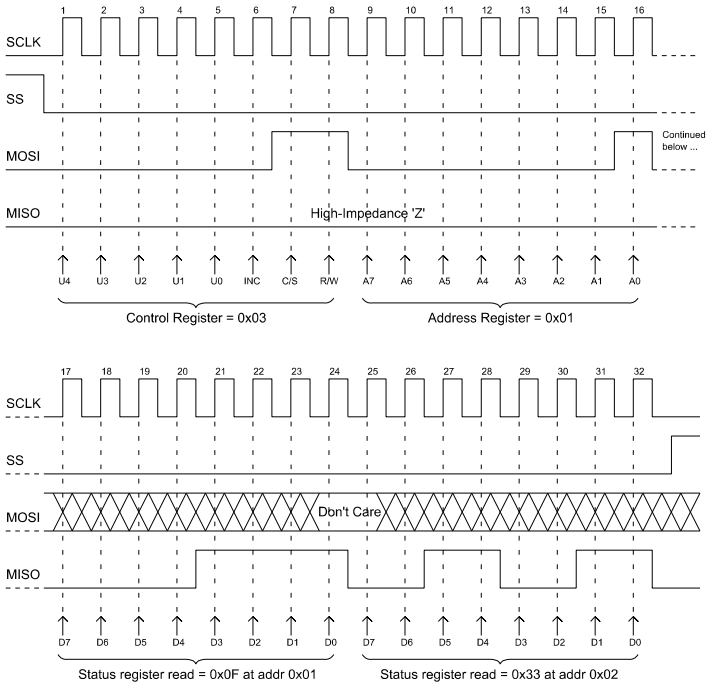

Figure 5: Status register read example

Source File Description

All source files are provided as text files coded in VHDL. The following table gives a brief description of each file.

Source file	***Description***
spi_slave_stim.txt	Input stimulus text file
spi_obuf.vhd	Tristate output buffer
spi_config_reg.vhd	Configuration register bank
spi_status_reg.vhd	Status register bank
spi_slave_cont.vhd	Main SPI slave controller
spi_slave_file_reader.vhd	Reads the SPI bus signals from a text file
spi_slave.vhd	Top-level block
spi_slave_bench.vhd	Top-level test bench

Functional Testing

An example VHDL test bench is provided for use in a suitable VHDL simulator. The compilation order of the source code is as follows:

1. spi_obuf.vhd
2. spi_config_reg.vhd
3. spi_status_reg.vhd
4. spi_slave_cont.vhd
5. spi_slave.vhd
6. spi_slave_file_reader.vhd
7. spi_slave_bench.vhd

The VHDL test bench instantiates the spi_slave component together with a file-reader module that reads the SPI bus signals from a text file. The SPI serial clock characteristics may be modified by changing the generic parameters *cpol* and *cpha*. In addition, the number of configuration and status registers may be changed with the parameters *num_config* and *num_status*. The testbench provided sets up the slave controller with 4 config regs and 4 status regs. By default, the SPI mode is set to 0,0 (i.e. cpol = 0, cpha = 0).

The input stimulus text file is called *spi_slave_stim.txt* and should be put in the current top-level VHDL simulation directory. This text file contains SPI commands that emulates the action of the SPI master on the bus. As an example, in order to send the byte 0x55 to the slave controller (in SPI mode 0,0) the text file would read:

```
0 0 0   # SS = 0, SCLK = 0, MOSI = 0
0 1 0   # SS = 0, SCLK = 1, MOSI = 0
0 0 1   # SS = 0, SCLK = 0, MOSI = 1
0 1 1   # SS = 0, SCLK = 1, MOSI = 1
0 0 0   # SS = 0, SCLK = 0, MOSI = 0
0 1 0   # SS = 0, SCLK = 1, MOSI = 0
0 0 1   # SS = 0, SCLK = 0, MOSI = 1
0 1 1   # SS = 0, SCLK = 1, MOSI = 1
0 0 0   # SS = 0, SCLK = 0, MOSI = 0
0 1 0   # SS = 0, SCLK = 1, MOSI = 0
0 0 1   # SS = 0, SCLK = 0, MOSI = 1
0 1 1   # SS = 0, SCLK = 1, MOSI = 1
0 0 0   # SS = 0, SCLK = 0, MOSI = 0
0 1 0   # SS = 0, SCLK = 1, MOSI = 0
0 0 1   # SS = 0, SCLK = 0, MOSI = 1
0 1 1   # SS = 0, SCLK = 1, MOSI = 1
```

In the text file, the SPI bus signalling is split into 2 phases on 2 consecutive lines. Each line is comprised of three bits in the format 'A B C' where 'A' specifies the state of the SS line, 'B' is the state of the SCLK line and 'C' is the state of the MOSI line. The values 'A', 'B' and 'C' can either be specified as '0' or '1'.

In the default set up, the simulation must be run for around 20 ms during which time the file-reader module will drive the SPI bus with the input stimulus. In this particular example, the test bench performs a sequential write and read of the 4 config and status registers.

The simulation generates the text file *spi_slave_out.txt* which contains a snapshot of the 8-bit read/write data captured at the SPI interface during the course of the test. The contents of this file may be examined to verify the operation of the SPI slave controller.

Development Board Testing

The SPI Slave Serial Interface Controller was implemented on a Xilinx® 2V3000 FPGA running at a system clock frequency of 130MHz. The SPI Master device was also implemented on the FPGA and was initially set up to run at a serial clock frequency of 3.3MHz. After testing was performed at 3.3MHz, further testing was performed to verify correct operation at the higher frequency of 10MHz. The slave controller was implemented with a bank of 4 config registers and 4 status registers.

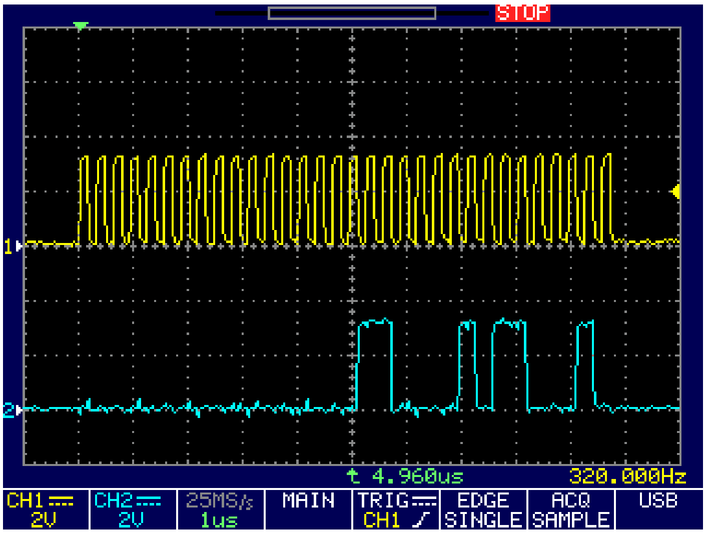

Figure 6: SPI write to config regs #0 and #1 values 0x61 and 0x62

Figure 6 above demonstrates a 32-bit sequential SPI write operation of the values 0x61 and 0x62 to the first two configuration registers. This was achieved by writing 0x00 to the control register, 0x00 to the address register then 0x61 and 0x62 to the first two config registers. The top trace is the serial clock running at 3.3MHz (SCLK). The bottom trace is the serial data in (MOSI).

The next figure shows the same two config registers being read in succession. The SPI write commands were respectively 0x01, 0x00, 0x00 and 0x00. The top trace is the serial clock (SCLK) and the bottom trace is the serial data out (MISO). Notice that the slave output is is high-impedance (and left floating) until the output enable is asserted for the last 16-bits.

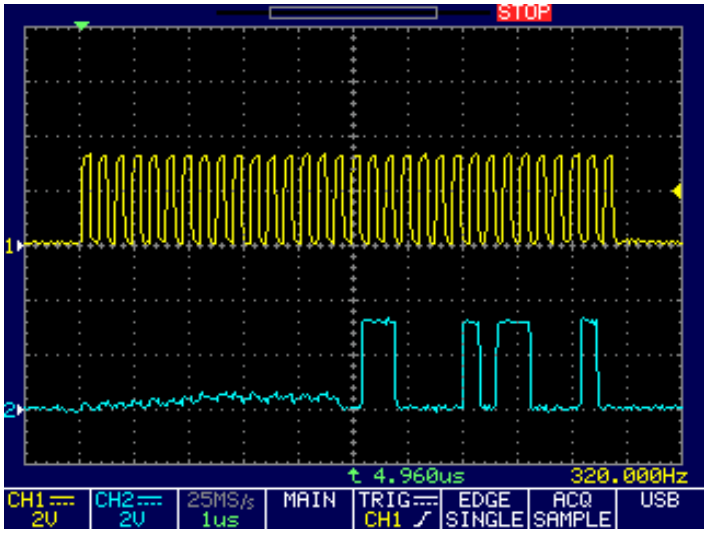

Figure 7: SPI read from config regs #0 and #1

Finally, Figure 8 shows detail of the SPI slave controller operating at 10Mbps data rate. In practice, the maximum attainable data rate is dependent on the choice of system clock frequency and the physical characteristics of the bus. This 10MHz example was achieved running a system clock of 130 MHz at LVTTL voltage levels with x12 drive strength output buffers. At higher system clock frequencies and with suitable bus characteristics then typical data rates approaching 40 Mbps can be achieved.

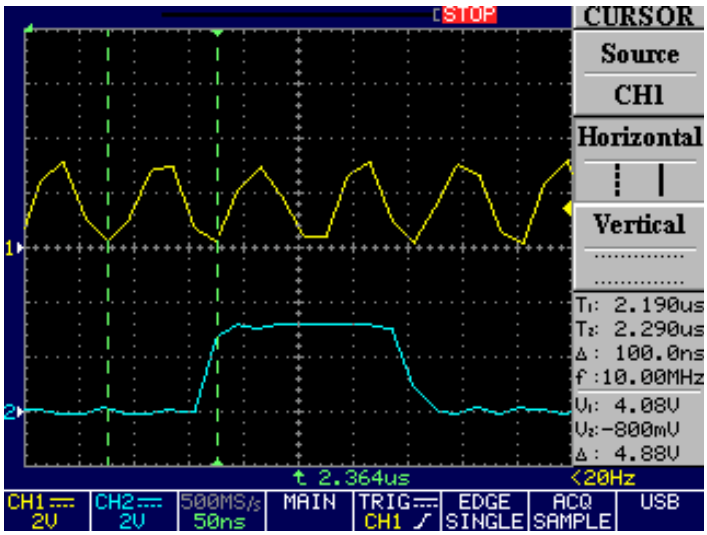

Figure 8: Example SPI read at 10Mbps data rate

Synthesis

The files required for synthesis and the design hierarchy is shown below:

- spi_slave.vhd
  - spi_slave_cont.vhd
  - spi_config_reg.vhd
  - spi_status_reg.vhd
  - spi_obuf.vhd

The VHDL core is designed to be technology independent. However, as a benchmark, synthesis results have been provided for the Xilinx® Virtex 6 and Spartan 6 FPGA devices. Synthesis results for other FPGAs and technologies can be provided on request.

Note that the number of config and status registers used in the implementation will have the greatest influence on the size and attainable clock speed of the controller core.

Trial synthesis results are shown with the generic parameters set to: num_config = 4, num_config_log2 = 2, num_status = 4, num_status_log2, cpol = 0, cpha = 0.

Resource usage is specified after Place and Route.

VIRTEX 6

Resource type	***Quantity used***
Slice register	97
Slice LUT	99
Block RAM	0
DSP48	0
Occupied slices	47
Clock frequency (approx)	300 MHz

SPARTAN 6

Resource type	***Quantity used***
Slice register	102
Slice LUT	105
Block RAM	0
DSP48	0
Occupied slices	62
Clock frequency (best case)	260 MHz

Revision History

Revision	*Change description*	*Date*
1.0	Initial revision	01/04/2009
1.1	Updated synthesis results for Xilinx® 6 series FPGAs	31/05/2012