XAPP386 (v1.0) December 24, 2002

CoolRunner-II Serial Peripheral Interface Master

Summary

This document details the VHDL implementation of a Serial Peripheral Interface (SPI) master in a Xilinx CoolRunner™-II CPLD. CoolRunner-II CPLDs are the lowest power CPLDs available, making this the perfect target device for an SPI Master. To obtain the VHDL code described in this document, go to section **VHDL Code Download and Disclaimer**, page 19 for instructions. This design fits XC2C256 CoolRunner-II or XCR3256XL CoolRunner XPLA3 CPLDs. For the CoolRunner-II CPLD version, please refer to **XAPP348, CoolRunner Serial Peripheral Interface Master**.

Introduction

The Serial Peripheral Interface (SPI) is a full-duplex, synchronous, serial data link that is standard across many microprocessors, microcontrollers, and peripherals. It enables communication between microprocessors and peripherals and/or inter-processor communication. The SPI system is flexible enough to interface directly with numerous commercially available peripherals.

A SPI Master design has been implemented in a CoolRunner-II CPLD. The CoolRunner-II SPI Master design can be used to provide a SPI controller to those microcontrollers or microprocessors that do not contain a SPI interface. A high-level block diagram is shown in Figure 1. The microcontroller (µC) interface chosen in this SPI Master implementation is based on the popular 8051 microcontroller bus cycles, but can easily be modified to other microcontroller interfaces.

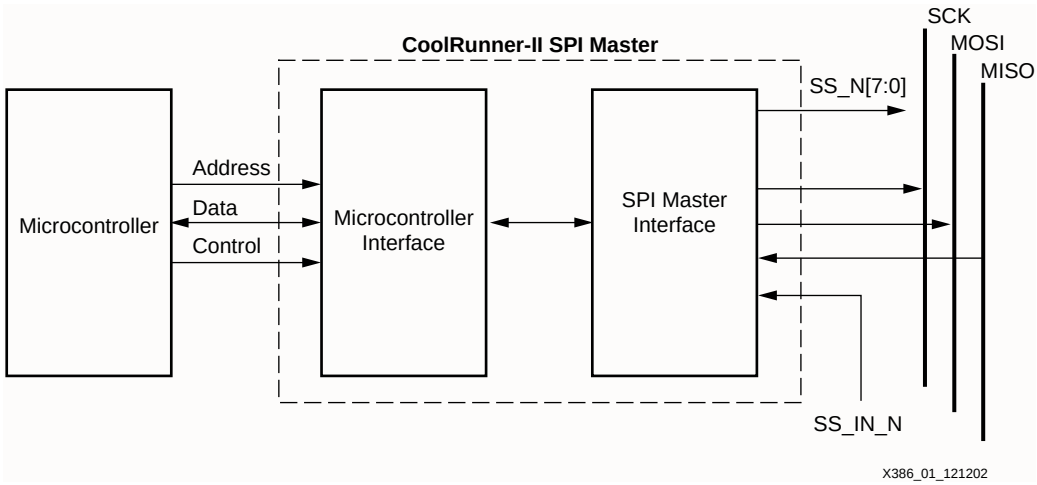

Figure 1: **CoolRunner-II SPI Master**

SPI Background

This section will describe the main protocol of the SPI bus. For more details and timing diagrams, please refer to the description of the SPI bus in the Motorola 68HC11 Reference Manual.

The SPI bus consists of four wires, Serial Clock (SCK), Master Out Slave In (MOSI), Master In Slave Out (MISO), and Slave Select (SS_N), which carry information between the devices connected to the bus.

The SCK control line is driven by the SPI Master and regulates the flow of data bits. The master may transmit data at a variety of baud rates; the SCK line transitions once for each bit in the transmitted data. The SPI specification allows a selection of clock polarity and a choice of two fundamentally different clocking protocols on an 8-bit oriented data transfer. The master selects one of four different bit rates for SCK. Data is shifted on one edge of SCK and is sampled on the opposite edge when the data is stable.

The MOSI data line carries the output data from the master which is shifted as the input data to the selected slave. The MISO data line carries the output data from the selected slave to the input of the master. There may be no more than one slave transmitting data during a particular transfer.

All SCK, MOSI, and MISO pins are tied together. A single SPI device is configured as a master; all other SPI devices on the SPI bus are configured as slaves. The single master drives data out its SCK and MOSI pins to the SCK and MOSI pins of the slaves. One selected slave device can drive data out its MISO pin to the MISO master pin.

The SS_N control line selects a particular slave via hardware control. This control line allows individual selection of a slave SPI device. Slave devices that are not selected do not interfere with SPI bus activities. Slave devices ignore the SCK signal and keeps the MISO output pin in a high impedance state unless the slave select pin is active low.

The SS_IN_N control line can be used as an input to the SPI Master indicating a multiple-master bus contention (SS_IN_N). If the SS_IN_N signal to the master is asserted, it indicates that some other device on the bus is attempting to be a master and address this device as a slave. Assertion of SS_IN_N automatically disables SPI output drivers in the master device if more than one device attempts to become master.

The clock phase and polarity can be modified for SPI data transfers. The clock polarity (CPOL) selects an active high or active low clock and has no significant effect on the transfer format. If CPOL = "0", then the idle state of SCK is low. If CPOL = "1", then the idle state of SCK is high. The clock phase (CPHA) can be modified to select one of two fundamentally different transfer formats. If CPHA = "0", data is valid on the first SCK edge (rising or falling) after SS_N has asserted. If CPHA = "1", data is valid on the second SCK edge (rising or falling) after SS_N has asserted. The clock phase and polarity should be identical for the master SPI device and the communicating slave device.

Figure 2 shows the timing diagram for a SPI data transfer when the clock phase, CPHA, is set to "0". The waveform is shown for both positive and negative clock polarities of SCK. The SCK signal remains inactive for the first half of the first SCK cycle (SCK Cycle 1). In this transfer format, the falling edge of SS_N indicates the beginning of a data transfer. Therefore, the SS_N

line must be negated and reasserted between each successive serial byte. If the slave writes data to the SPI data register while SS_N is active Low, a write-collision error results.

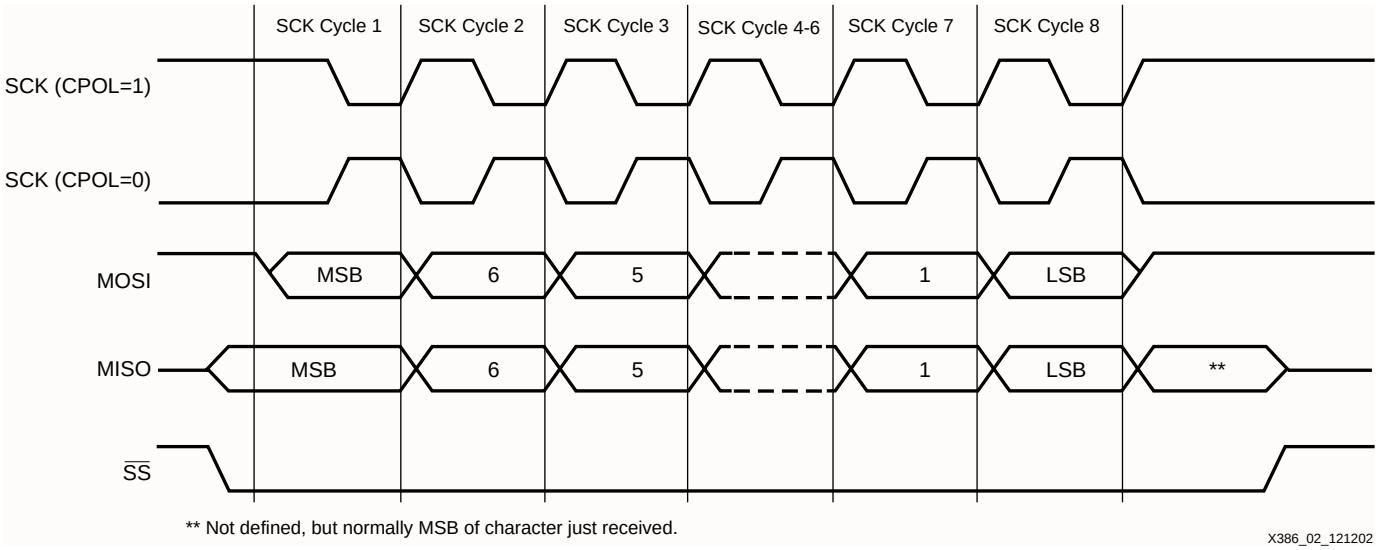

Figure 2: **Data Transfer on the SPI Bus with CPHA=0**

Figure 3 shows the timing diagram for a SPI transfer when the clock phase, CPHA, is set to "1". The waveform is shown for both positive and negative clock polarities of SCK. The first SCK cycle begins with an edge on the SCK line from its inactive to its active level. The first edge of SCK indicates the start of the data transfer in this format. The SS_N line may remain active low between successive transfers. This format is useful in systems with a single master and single slave.

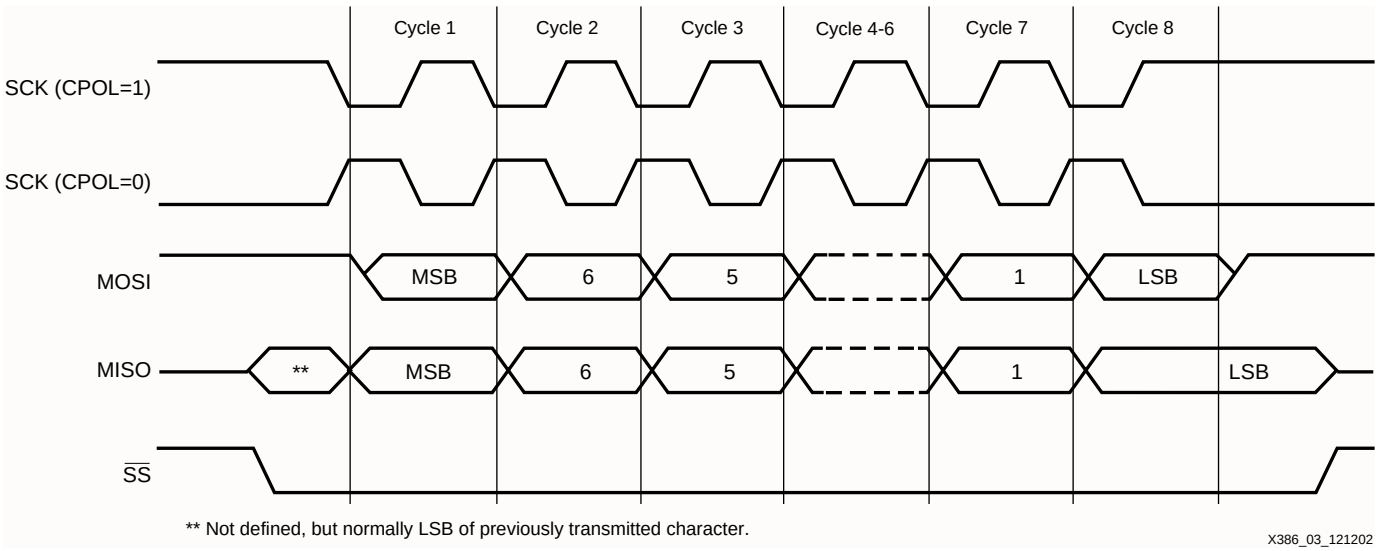

Figure 3: **Data Transfer on SPI Bus with CPHA=1**

When an SPI transfer occurs, an 8-bit data word is shifted out one interface pin while a different 8-bit data word is being shifted in on another interface pin. This can be viewed as an 8-bit shift register in the SPI Master device and another 8-bit shift register in a SPI slave device that are connected as a circular 16-bit shift register. When a transfer occurs, this 16-bit shift register is shifted 8 positions, thus exchanging the 8-bit data between the master and slave devices.

The SPI specification details the timing and wave forms for byte transfers on the SPI bus, but does not dictate the data protocol used in these transfers, i.e., the interface specification does

not specify that the first byte contain an address or a read/write line. When communicating to devices over the SPI bus, it is important to review the specifications of these devices to determine the required communication protocol so that commands, addresses and the data direction (read/write) can be set correctly. The CoolRunner-II CPLD SPI Master implementation will not enforce a particular data protocol. It is expected that the µC will specify the data bytes to be transferred on the SPI bus in the correct order. All data received on the SPI bus will be stored in a receive register for the user logic to interpret. All data written to the transmit register will be transmitted on the SPI bus.

CoolRunner-II SPI Master Implementation

The CoolRunner-II CPLD implementation of an SPI Master supports the following features:

- Microcontroller interface
- Multi-master bus contention detection and interrupt
- Eight external slave selects
- Four transfer protocols available with selectable clock polarity and clock phase
- SPI transfer complete microcontroller interrupt
- Four different bit rates available for SCK

Signal Descriptions

The 36 I/O signals of the CoolRunner-II CPLD SPI Master are described in Table 1. Pin numbers have not been assigned to this design, this can be done to meet the system requirements of the designer.

Table 1: **CoolRunner-II SPI Master Signal Description**

Name	Direction	Description
MOSI	Output	**SPI Serial Data Output.** Serial data output from the SPI Master to a SPI slave.
MISO	Input	**SPI Serial Data Input.** Serial data input from a SPI slave to the SPI Master.
SS_IN_N	Input	**SPI Slave Select Input.** Active Low slave select input to the SPI Master. If this line is asserted, it indicates that another master on the bus was trying to select this SPI Master as a SPI slave and thus bus contention can occur. Assertion of this line causes the CoolRunner-II CPLD SPI Master to reset all communication and interrupt the µC.
SS_N[7:0]	Output	**SPI Slave Selects.** Active Low slave select signals to eight SPI slaves in the system.
SCK	Output	**SPI Serial Clock.** Clock output selectable as 1/2, 1/4, 1/8, or 1/16 of the system clock.
ADDR[15:8]	Input	**µC Address Bus.** High byte address bus.
ADDR_DATA[7:0]	Bidirectional	**µC Multiplexed Address/Data Bus**.
ALE_N	Input	**Address Latch Enable**. Active Low µC control signal indicating that the data present on the multiplexed address/data bus is a valid address.
PSEN_N	Input	**Program Store Enable.** Active Low µC control signal indicating that the current bus cycle is an access to the external program memory.
RD_N	Input	**Read Strobe.** Active Low µC control signal indicating that the current bus cycle is a read cycle.

Table 1: **CoolRunner-II SPI Master Signal Description *(Continued)***

WR_N	Input	**Write Strobe.** Active Low µC control signal indicating that the current bus cycle is a write cycle.
INT_N	Output	**Interrupt Request.** Active Low signal to generate an interrupt to the µC. This signal is asserted when interrupts are enabled and there is SPI bus contention as indicated by SS_IN_N or when the transmit register is empty (SPITR) during a transaction, or when the receive register is full and the transaction is complete.
XMIT_EMPTY	Output	**Transmit Register Empty.** Active High signal indicating that the transmit register (SPITR) is empty. This bit is used to signal the loading of data from the SPITR to the SPI transmit shift register indicating that the µC can load another byte of data into the SPITR. This signal could be connected to an µC interrupt or to an I/O port. This signal causes INT_N to assert during data transfers but does not cause an interrupt after the transfer is complete (i.e. START = 0). This signal is brought out of the CPLD as a separate I/O pin for systems that do not want to use interrupts.
RCV_FULL	Output	**Receive Register Full**. Active High signal indicating that the receive register (SPIRR) is full. This bit is used to signal the loading of data from the SPI receive shift register to the SPIRR. This signal could be connected to an µC interrupt or to an I/O port. This signal causes INT_N to assert only for the last word received from the transfer (i.e., START = 0). This signal is brought out of the CPLD as a separate I/O pin for systems that do not want to use interrupts.
CLK	Input	**Clock.** This clock is input from the system and is used to generate the SCK signal.
RESET	Input	**Reset.** Active High reset from the system. When asserted, all logic in the CoolRunner-II CPLD is reset.

Block Diagram

The block diagram of the CoolRunner-II CPLD SPI Master, shown in Figure 4 was broken into two major blocks, the µC interface and the SPI interface.

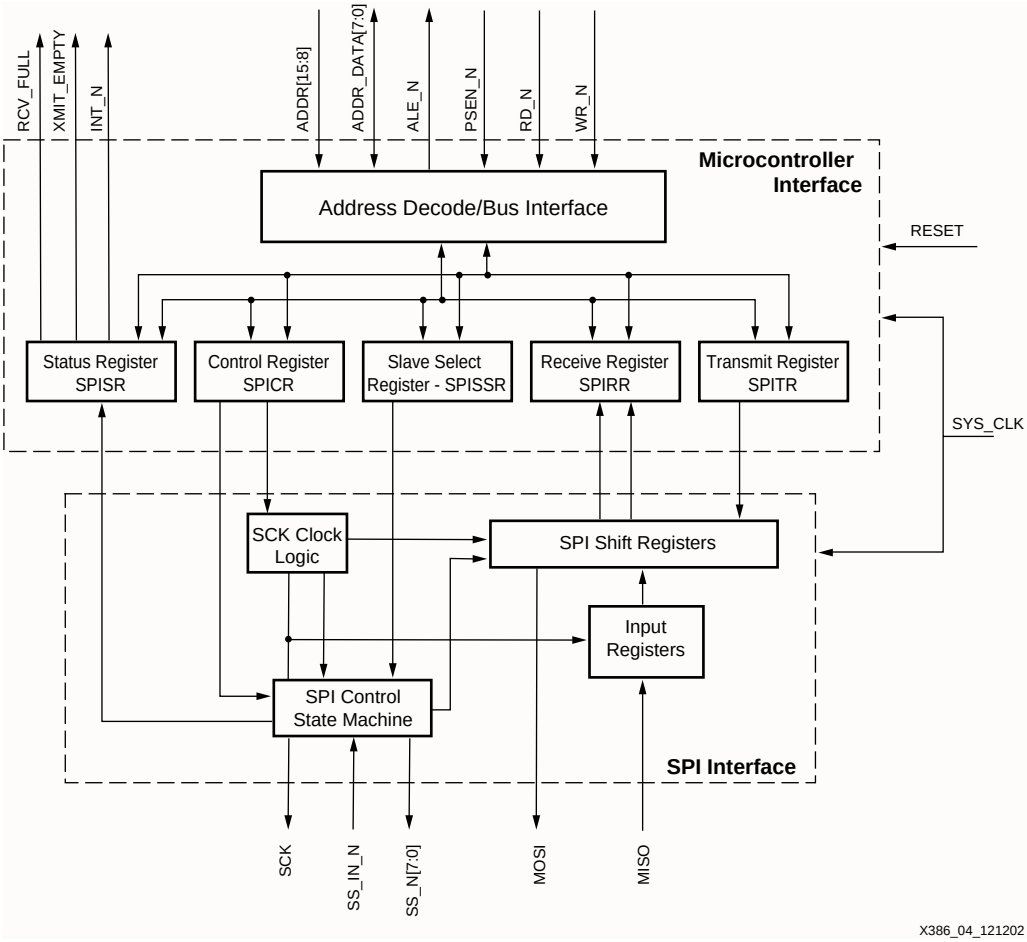

Figure 4: **SPI Master Block Diagram**

µC Interface Logic

The µC interface for the SPI Master design supports a byte-wide, multiplexed address/data bus protocol found in the popular 8051 µC. This protocol is the method in which the µC reads and writes the registers in the CoolRunner-II CPLD and is shown in Figure 5.

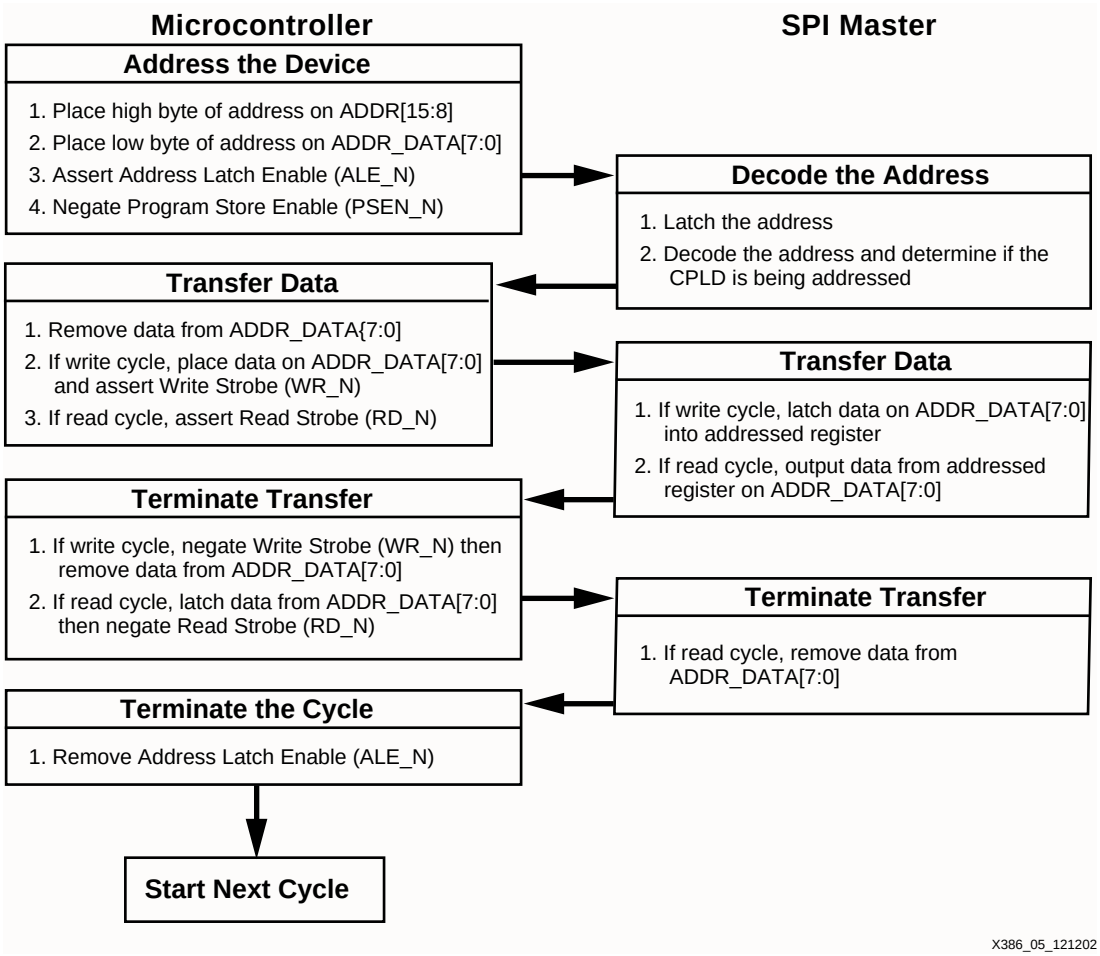

Figure 5: **µC Read/Write Protocol**

Note that the code to interface to the 8051 µC is a separate VHDL module which interfaces to the SPI interface logic via a set of registers. Therefore, this code can easily be replaced with the µC interface of your choice.

Address Decode/Bus Interface Logic

The µC bus protocol is implemented in the CoolRunner-II SPI Master in the state machine shown in Figure 6.

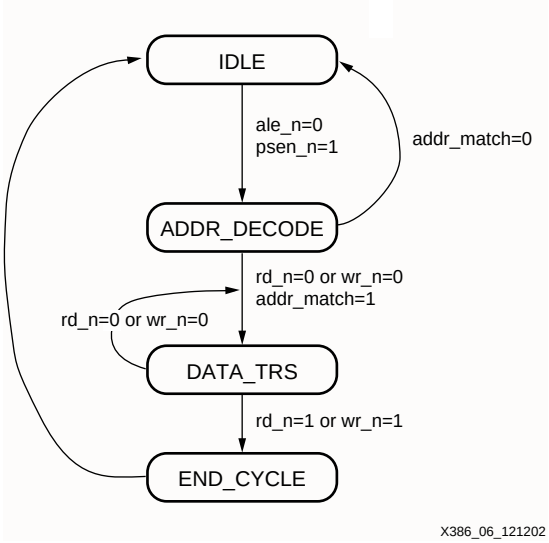

Figure 6: **µC Bus Interface State Machine**

In the first cycle, the µC places the address on the address bus and asserts address latch enable (ALE_N). ALE_N indicates that the data on the multiplexed address/data bus is a valid address and that the address on ADDR[15:0] is also valid.

Upon the assertion of ALE_N, the CoolRunner-II SPI Master transitions to the ADDR_DECODE state to decode the address and determine if it is the device being addressed. The enables for the internal registers are set in this state. ALE_N is also used to register the lower address bits from the multiplexed ADDR_DATA bus.

If this is a write cycle, the µC removes the address from the multiplexed address/data bus and places the data to be written onto these signals. The write strobe (WR_N) is then asserted. If this a read cycle, the µC 3-states the multiplexed address/data bus and asserts the read strobe (RD_N) indicating that the CoolRunner-II SPI Master can place data from the addressed register on the data bus.

If the CoolRunner-II SPI Master is being addressed and either RD_N or WR_N are asserted, the CoolRunner-II SPI Master progresses to the DATA_TRS state. If this is a read cycle, the requested data is placed on the bus and if this is a write cycle, the data from the data bus is latched in the addressed register.

The µC latches the data present on the bus if this is a read cycle and then negates the read strobe (RD_N). If this is a write cycle, the µC removes data from the bus and then negates the write strobe (WR_N). The negation of either RD_N or WR_N causes the CoolRunner-II SPI Master to progress to the END_CYCLE state. The CoolRunner-II CPLD will 3-state the multiplexed address/data bus in this state, removing the data if it is a read cycle.

At this point, the µC ends the cycle by negating address latch enable (ALE_N), which causes the CoolRunner-II CPLD to return to the IDLE state.

CoolRunner-II SPI Master Registers

The base address used for address decoding is set in the VHDL code via the constant BASE_ADDRESS. The lower four address bits determine which register inside the CPLD is being accessed. Each register address is set by a constant in the µC interface VHDL code that can easily be modified to meet the addressing scheme of the system.

The registers supported in the CoolRunner-II SPI Master are described in the Table 2. The µC interface logic of the CoolRunner-II SPI Master handles the reading and writing of these registers by the µC and supplies and/or retrieves these bits to/from the SPI interface logic.

Table 2: **SPI Master Registers**

Address	Register	VHDL Constant	Description
BASE + $80\h	SPISR	SPISR_ADDR	SPI Status Register
BASE + $84\h	SPICR	SPICR_ADDR	SPI Control Register
BASE + $88\h	SPISSR	SPISSR_ADDR	SPI Slave Select Register
BASE + $8A\h	SPITR	SPITR_ADDR	SPI Transmit Data Register
BASE + $8E\h	SPIRR	SPIRR_ADDR	SPI Receive Data Register

SPI Status Register (SPISR)

This register contains the status of the SPI controller. This status register is read-only with the exception of certain bits which are software clearable as described in Table 3.

Table 3: **Status Register Bits**

Bit Location	Name	µC Access	Description
7	Done	Read	**Done Bit.** While one byte of data is being transferred, this bit is cleared. It is set during the eighth SCK clock period of a byte transfer. • "1" transfer is complete • "0" transfer in progress
6	SPIERR	Read Software Clearable	**SPI Error Bit.** When the SS_IN_N input pin is asserted, this bit is set indicating an SPI error condition. The SPI interface resets and 3-states all signals on the SPI bus. An interrupt will be asserted to the µC when this bit is set if interrupts are enabled. This bit must be cleared by the µC writing a "0" to this bit in the interrupt service routine. See Figure 11 for details on how to handle an SPI error.
5	BB	Read	**Bus Busy Bit.** This bit indicates the status of the SPI bus. This bit is set when a slave select line (SS_N) is asserted and cleared when a slave select line (SS_N) is negated. • "1" indicates the bus is busy • "0" indicates the bus is idle The µC should examine this bit to insure that the bus is not busy before configuring the SPI interface for an SPI transaction.
4	INT_N	Read Software Clearable	**Interrupt Bit.** This bit is asserted (active low) when an interrupt is pending which causes a processor interrupt request if INTEN is set. An interrupt is asserted if any of the following conditions occur: • The transmit register (SPITR) is empty and there are more data words to transmit (START = 1) • The receive register (SPIRR) is full and there are no more data words to transmit (START = 0) • An SPI error has occurred This bit is negated whenever the µC writes data to the SPITR, reads data from the SPIRR, or resets the SPIERR bit.

Table 3: **Status Register Bits *(Continued)***

Bit Location	Name	µC Access	Description
3	XMIT_EMPTY	Read	**Transmit Register Empty.** This bit is set when the transmit register (SPITR) is empty. It is cleared when the µC writes data into this register. An interrupt will be asserted to the µC when this bit is set if interrupts are enabled and there are more data words to transmit (START = 1). Note that this bit is also an output pin.
2	RCV_FULL	Read	Receive Register Full. This bit is set whenever the receive register (SPIRR) is full. It is cleared when the µC reads from this register. An interrupt will be asserted to the µC when this bit is set if interrupts are enabled and there are no more data words to transmit (START =0). Note that this bit is also an output pin.
1-0	Unused		**Unused Bits.** These bits will read as "0" when the status register is read.

SPI Control Register (SPICR)

This register contains the bits to configure the SPI Master. (Table 4)

Table 4: **Control Register Bits**

Bit Location	Name	µC Access	Description
7	SPIEN	Read/Write	**SPI Master Enable.** This bit must be set before any other SPICR bits have any effect • "1" enables the SPI Master • "0" resets and disables the SPI Master
6	INTEN	Read/Write	**Interrupt Enable.** • "1" enables interrupts. An interrupt occurs if the INT_N bit in the status register is also set • "0" disables interrupts but does not clear the cause of any currently pending interrupts
5	START	Read/Write	**SPI Transfer Start.** When the µC changes this bit from "0" to "1", the SPI Master begins to transfer the data in the SPI Transfer data register (SPITR) on the SPI bus if XMIT_EMPTY is negated. All data received from the SPI bus is captured in the SPI Receive data register (SPIRR). As long as this bit stays asserted, SPI transfers will occur. After the µC has written the last data word to be transferred in SPITR and XMIT_EMPTY asserts, the µC must negate this bit to indicate that this is the last word in the SPI transfer.
4-3	CLKDIV	Read/Write	**Clock Divider Bits.** These bits determine the frequency of SCK by selecting a divide by 4, 8,16, or 32 of the system clock. • "00" - SCK is 1/4 of the system clock • "01" - SCK is 1/8 of the system clock • "10" - SCK is 1/16 of the system clock • "11" - SCK is 1/32 of the system clock

Table 4: **Control Register Bits *(Continued)***

Bit Location	Name	µC Access	Description
2	CPHA(1)	Read/Write	Clock Phase Bit. This bit determines the clock phase of SCK in relationship to the serial data. • "0" - data is valid on first SCK edge (rising or falling) after slave select has asserted • "1" - data is valid on second SCK edge (rising or falling) after slave select has asserted
1	CPOL(1)	Read/Write	Clock Polarity Bit. This bit determines the polarity of SCK. • "0" - SCK is low when idle • "1" - SCK is high when idle
0	RCV_CPOL(1)	Read/Write	**Receive Clock Polarity.** This bit determines whether the MISO data is sampled on the rising or falling edge of the outgoing SCK. • "0" - MISO data is sampled on the falling edge of SCK • "1" - MISO data is sampled on the rising edge of SCK Note that in most cases, RCV_CPOL = "1" when CPHA is the same value as CPOL and is "0" otherwise. However, this bit should be set according to the slave that is being accessed.

Notes:
1. CPHA, CPOL, RCV_CPOL should be set to match the protocol expected by the SPI slave that is the target of the SPI bus cycle.

SPI Slave Select Register (SPISSR)

This register contains bits which indicate which slave select line should be asserted. A "1" in a bit position indicates that the corresponding bit in the slave select output bus is asserted according to the SPI timing specifications. A "0" in a bit position indicates that the corresponding bit in the slave select output bus stays negated. This allows the µC to specify which slave select line is asserted during a SPI data transfer. Note that only one slave select line can be asserted during a SPI data transfer. This must be adhered to by the µC, the hardware implementation of this specification does not enforce this requirement, in other words, if the µC sets multiple bits in this register, multiple slave select lines will be asserted for the SPI transfer. (Table 5)

Table 5: **SPI Slave Select Register**

Bit Location	Name	µC Access	Description
7 - 0	SS_N7 - SS_N0	Read/Write	SPI Slave Selects

SPI Transfer Data Register (SPITR)

This register contains data to be transmitted on the SPI bus on the MOSI pin (Table 6). Data written into this register is output on the SPI bus once the START bit in the control register (SPICR) has been asserted. As long as the START bit in the SPICR is asserted, additional bytes of data in this register will continue to be transmitted on the SPI bus.

Once this data has been loaded into the SPI transmit shift register, XMIT_EMPTY asserts and the µC can place the next data byte for transmission on the SPI bus into this register. Writing data into this register resets the XMIT_EMPTY flag. Note that XMIT_EMPTY must be negated

and START must be asserted before the SPI state machine will begin transmission of data on the SPI bus.

Table 6: **SPI Transmit Data Register**

Bit Location	Name	µC Access	Description
7 - 0	D7 - D0	Read/Write	SPI Transmit Data

SPI Receive Data Register (SPIRR)

This register contains the data received from the SPI bus on the MISO pin. When a byte of data has been received from the SPI bus and transferred to the SPIRR, the RCV_FULL flag asserts. The µC then reads data from the SPIRR which resets the RCV_FULL flag. Because data is loaded from the SPI receive shift register to the SPIRR, the µC has an entire 8-bit SPI transfer to read the data from the SPIRR. (Table 7)

Table 7: **SPI Receive Data Register**

Bit Location	Name	µC Access	Description
7 - 0	D7 - D0	Read/Write	SPI Receive Data

SPI Interface Logic

The SPI bus interface logic consists of several different processes as seen in Figure 4. Control bits from the µC interface registers determine the behavior of these processes.

SPI Control State Machine

This process generates the slave select signals and controls the shift and load operations of the SPI transmit shift register. It also monitors the SPI bus and determines when a byte transfer is complete. This process also generates clock mask signals that control when the clock is output to the SPI bus. If the START signal stays asserted after a byte has been transferred, the state machine will continue to output the next byte and the SCK signal continues to transition. Note that if CPHA = 0, the slave select signal will negate and then re-assert between consecutive byte transfers as stated in the SPI specification. If the START signal is negated after a byte has been transferred, then the state machine first insures that the current transfer has been completed and the SCK output is placed in its inactive state as determined by CPOL.

The slave select is then negated after the required hold time. The SPI control state machine is shown in Figure 7.

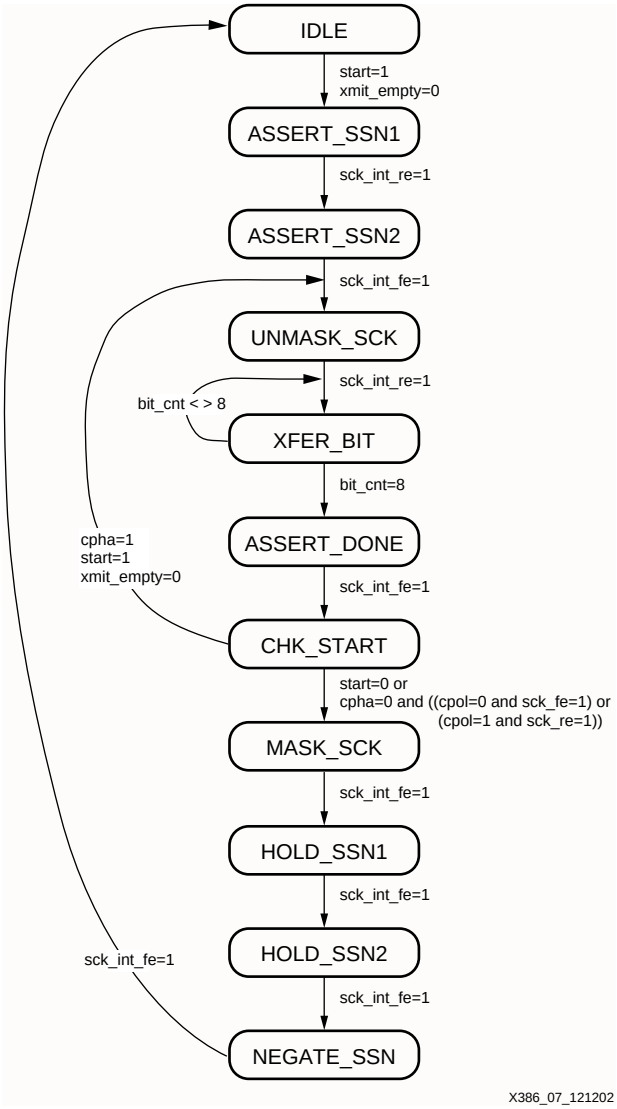

Figure 7: **SPI Control State Machine**

The SPI Control state machine remains in the IDLE state until the START bit in the SPI Control register is asserted and the XMIT_EMPTY signal is negated. At this point, the state machine moves to the ASSERT_SSN1 state to assert the internal SS_N signal. This signal is then masked with the SPI Slave Select Register (SPISSR) to generate the slave select signals output to the system. After a rising edge of the internal SCK (SCK_INT_RE=1), the state machine transitions to the ASSERT_SSN2 state, keeping SS_N asserted until the falling edge of the internal SCK. This state machine will insure that SS_N will assert ~1 SCK period before the first SCK edge, meeting the SS_N setup time requirement of most SPI slave devices. This timing parameter should be verified by the designer for the target system as the SS_N setup time requirement may vary between different SPI slaves.

After SS_N has been asserted for both the ASSERT_SSN1 and ASSERT_SSN2 states, the state machine transitions to the UNMASK_SCK state. The first edge of the phase 1 (CPHA=1) version of SCK occurs before the first data is output, therefore, this state unmasks the phase1 version of SCK. The SPI transmit shift register is loaded with data from the SPITR in this state. The SPI transmit shift register is clocked by the rising edge of the internal SCK signal. The state

machine waits for a rising edge of the internal SCK signal (SCK_INT_RE) to leave this state to insure that the SPI transmit shift register has been loaded.

The state machine then moves to the XFER_BIT state. The first edge of the phase 0 (CPHA=0) version of SCK occurs after data has been output, therefore, this state unmasks the phase 0 version of SCK. This state shifts data from the SPI transmit shift register and the state machine remains in this state until the byte transfer is complete. Once the byte transfer has completed, the state machine transitions to the ASSERT_DONE state where the DONE signal in the SPI Status Register (SPISR) is asserted. The state machine will not transition to the CHK_START state until the next falling edge of the internal SCK to synchronize this state machine to the internal SCK.

The SPI specification requires that SS_N be negated and re-asserted between consecutive byte transfers when CPHA=0. If CPHA=1, SS_N can remain asserted during consecutive byte transfers. Therefore, if START is still asserted in the CHK_START state and CPHA=1 and XMIT_EMPTY is negated, the state machine transitions back to the UNMASK_SCK state and continues SPI transfers. If START is negated or CPHA=0, the state machine transitions to the MASK_SCK state which masks both the phase 0 and the phase1 versions of SCK. Note, however, that the state machine waits for either the rising edge (if CPOL=1) or the falling edge (if CPOL=0) of the external SCK before transitioning to this state to insure that the transmission has been completed before masking the external clock.

At the next falling edge of the internal SCK (SCK_INT_FE), the state machine transitions to the HOLD_SSN1 state. SS_N must stay asserted for some time period after the last SCK edge.To insure that the SS_N hold time is at least two SCK periods, the state machine transitions to HOLD_SSN2 after the next falling edge of the internal SCK (SCK_INT_FE) and remains in this state until another falling edge of the internal SCK has occurred. This will meet the SS_N hold time requirement of most SPI slave devices. This timing parameter should be verified by the designer for the target system as the SS_N hold time requirement may vary between different SPI slaves.

At this point, the state machine transitions to the NEGATE_SSN state and remains in this state until the next falling edge of the internal SCK. This insures that the pulse width of SS_N between SPI transfers is at least one SCK period. This will meet the SS_N pulse width requirement of most SPI slave devices. This timing parameter should be verified by the designer for the target system as the SS_N pulse width requirement may vary between different SPI slaves.

The state machine then transitions to the IDLE state. If START is asserted and XMIT_EMPTY is negated, the SPI transfer and state machine operation will repeat.

Note that if no further SPI transfers are required, the µC must negate the START signal after writing the last data word of the transmission to the SPITR as shown in Figure 11.

Also note that at any time, the assertion of SS_IN_N will cause the SPI Control state machine to return to the IDLE state and the MOSI, SS_N, and SCK outputs will be 3-stated. The SPI Master will remain in this state until the SS_IN_N signal is negated and the START signal is asserted. When a SPI error occurs, the system must be examined to determine the cause of the error. The µC can reset the bit in the SPI status register (SPISR) and then continue to read the SPI status register (SPISR) to determine if the system error has been corrected. Once the error has been fixed at the system level, the SPI interface should be reset to guarantee correct operation as shown in Figure 11. The µC can reset the SPI Master by negating the SPIEN bit in the SPI control register (SPICR).

Transmit Empty and Receive Full Flags

The transmit empty flag (XMIT_EMPTY) is set whenever data is loaded from the SPITR to the SPI transmit shift register. This signal is clocked from the internal SCK and is reset whenever the µC writes data into the SPITR or when the system reset signal is asserted.

The receive full flag (RCV_FULL) is set whenever data is loaded from the SPI receive shift register to the SPIRR. This signal is clocked from the system clock and is reset whenever the µC reads data from the SPIRR.

SCK Clock Logic

This process generates the SCK output based on the CLKDIV, CPHA, and CPOL settings in the SPI control register. The clock frequency of the SCK signal is determined by dividing down the input clock based on the entries in the control register. The signal, SCK_INT is the internal SCK used to clock serial data out of the device and is continually generated. The SPI Control state machine is synchronized to this internal signal. The signal SCK_1 represents SCK when CPHA = 1 and the signal SCK_0 represents SCK when CPHA = 0. The SPI control state machine generates the masks for these clocks (CLK0_MASK, CLK1_MASK) so that the output SCK has the correct phase relationship with the data and is held in its inactive state when there is no data to be transferred. A representation of the logic required to generate the SCK signal output to the SPI bus is shown in Figure 8.

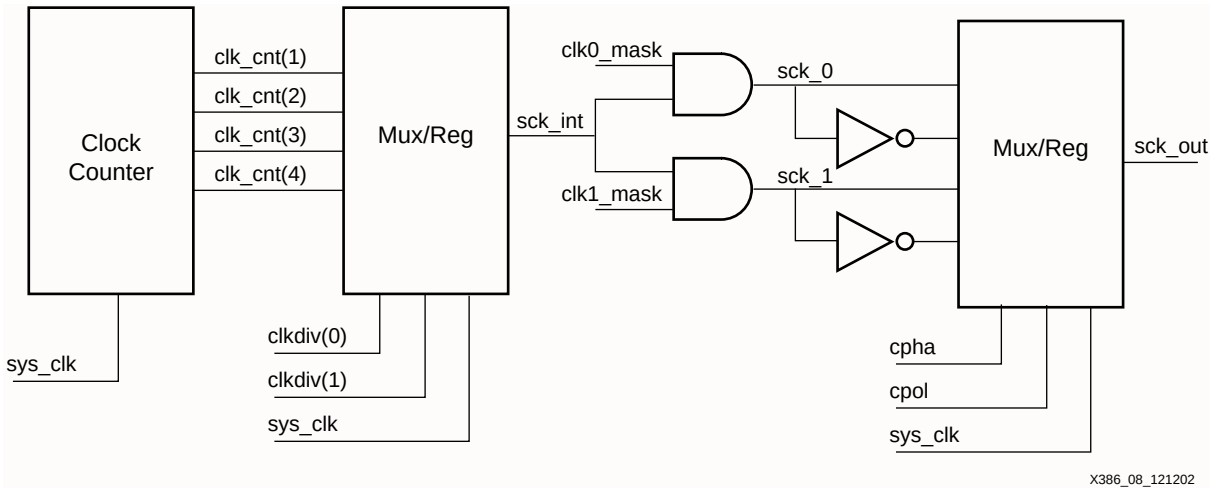

Figure 8: **SCK Clock Generation Logic**

SPI Shift Registers

SPI Transmit Shift Register

The SPI transmit shift register is an 8-bit loadable shift register containing SPI data. This shift register is loaded from the SPI Transmit Register (SPITR) via a load signal generated by the SPI Control state machine and is clocked by the rising edge of SCK_INT. The data shifting out is the MOSI data. Note that in Figure 8, SCK_OUT is one SYS_CLK delay from SCK_INT. Therefore, it is necessary to delay the data being shifted out from the SPI transmit shift register by one SYS_CLK as well so that the relationship between MOSI and SCK_OUT is maintained.

This is accomplished by a single register clocked from SYS_CLK which simply clocks the data output from the shift register as shown in Figure 9..

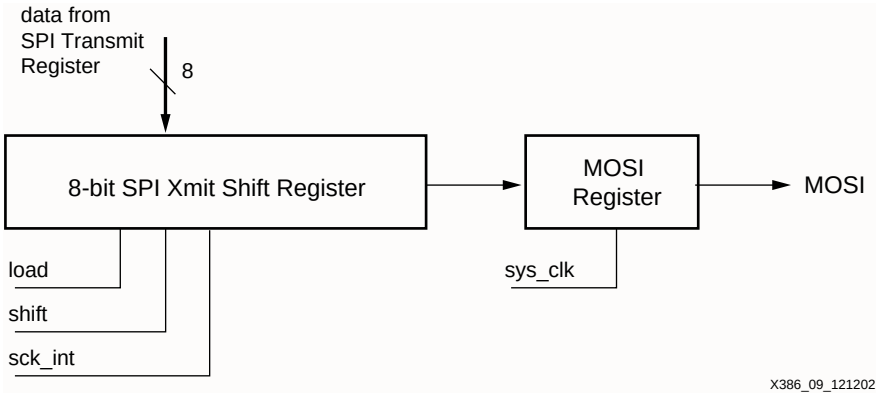

Figure 9: **SPI Transmit Shift Register**

SPI Receive Shift Register

A separate shift register is used to receive the MISO data since the clock phase and polarity of the SCK output can vary based on each transaction. The SPI receive shift register is clocked on the rising edge of the external SCK. The RCV_CPOL bit set in the control register allows the µC to specify which edge of the external SCK incoming MISO data is sampled on. This allows for flexibility in dealing with all types of different SPI slave devices as some SPI slaves will clock data out on the rising edge of SCK while others will clock data out on the falling edge of SCK. If a slave clocks data out on the falling edge of SCK, then RCV_CPOL should be set to "1" so that the CoolRunner-II SPI Master will clock data in on the rising edge of SCK. If a slave clocks data out on the rising edge of SCK, then RCV_CPOL should be set to "0" so that the CoolRunner-II SPI Master clocks data in on the falling edge of SCK. This eliminates any setup and hold timing issues. If slaves behave according to the SPI specification for CPHA and CPOL, RCV_CPOL will equal "1" whenever CPHA and CPOL are equal and "0" otherwise.

In the actual implementation, two input registers are used to sample MISO, one clocked on the rising edge of SCK, the other clocked on the falling edge of SCK. The outputs of these two registers are then multiplexed with the RCV_CPOL bit used as the select line. The output of this multiplexor then becomes the input to the SPI receive shift register which is clocked on the rising edge of the external SCK. Using this implementation method eliminates multiplexing clocks inside the CPLD. Multiplexing clocks within the CPLD places a delay between SCK and the actual data input to the shift register. This delay on SCK could impose an additional data hold time requirement on the slave device which is undesirable. Therefore, it was decided to multiplex the data instead, thus eliminating the need to specify a holdtime requirement on the slave device.

A counter, clocked off the rising edge of the external SCK, is used to count the bits shifted into the SPI receive shift register and to indicate when a byte transfer is complete. When the byte transfer is complete, the data in the SPI receive shift register is loaded into the SPI Receive

Register for use by the µC. The SPI receive shift register, MISO input registers, and receive bit counter are shown in Figure 10.

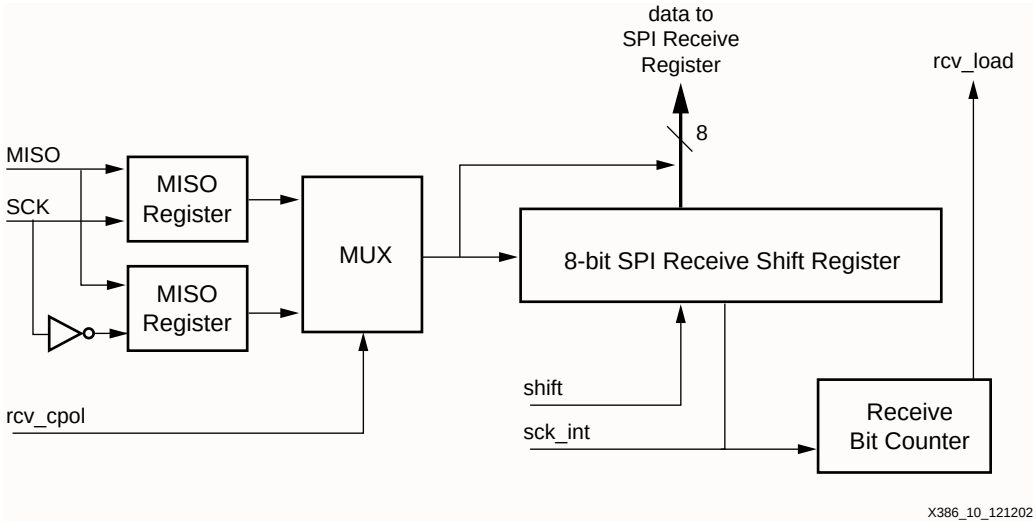

Figure 10: **SPI Receive Shift Register and MISO Input Data Registers**

Operational Flow Diagrams

The flow of the interface between the µC and the CoolRunner-II SPI Master is detailed in the following flow charts. These flow charts are meant to be a guide for utilizing the CoolRunner-II SPI Master in a µC system.

SPI Transaction Flow Chart

The flow chart for configuring and performing a SPI transaction for the CoolRunner-II SPI Master is shown in Figure 11. Since SPI is a full duplex communication protocol, data is received while data is transmitted.

Note that if an SPI error occurs indicating that another master has taken control over the bus, the system operation should be verified. The flow chart shows continually polling SPIERR in the status register to determine when the SPI error has been corrected. Since an SPI error also asserts an interrupt (if interrupts have been enabled), an alternative to polling the SPI status register (SPISR) is to service the interrupt to determine if the SPI error has been corrected. Note that either of these flows may not be the operational flow required by a system when an SPI error has occurred; the designer should determine the correct operations to execute when an SPI error occurs based on the system requirements

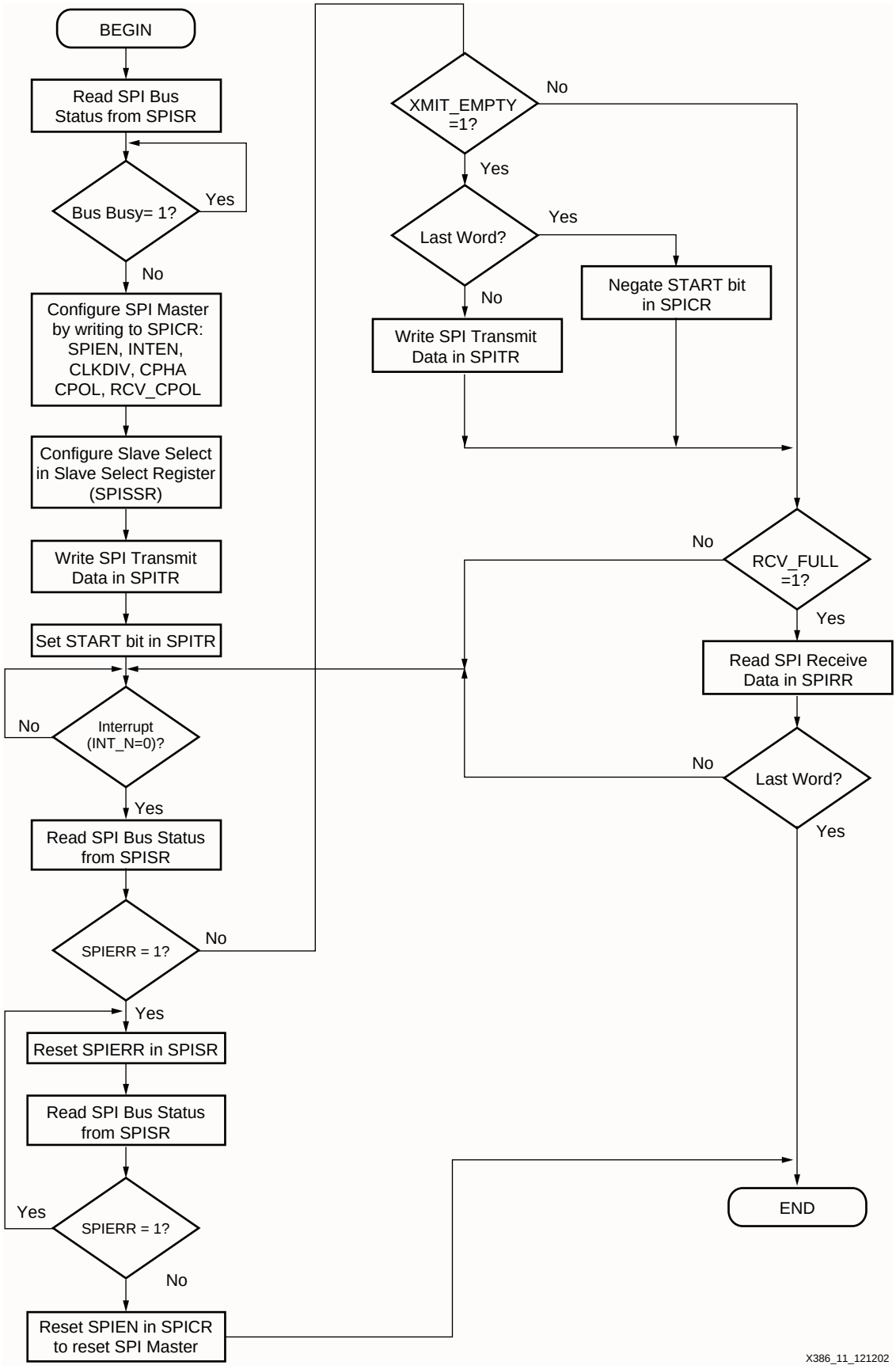

Figure 11: **CoolRunner-II SPI Master Transaction Flow Chart**

VHDL Testbench and Functional Simulation

A VHDL testbench has been developed that verifies the CoolRunner-II SPI Master implementation through all the possible combinations of CLKDIV, CPHA, CPOL, and RCV_CPOL. This testbench contains a process that emulates the bus cycles of the 8051 µC. Constants are provided at the top of the testbench file to set up the base address of the SPI Master device and all of the registers contained within the device. These constants should be modified to match the addressing scheme of the designer's system.

The VHDL testbench also provides a model of a simple SPI slave. The slave first transmits the hex value "CE" and then simply transmits the value that was received. The clock phase and polarity of the slave are dynamically set in the test bench based on the CPHA and CPOL values currently being tested.

To test the SPI Master reaction to an SPI error, the test bench contains the constant SS_IN_ASSERT_TIME which specifies the delay from the beginning of the simulation until SS_IN_N is asserted and also specifies the amount of time that SS_IN_N is asserted. Setting this constant to "0" disables assertion of SS_IN_N.

The test bench contains a signal, ERROR, which indicates that the data received did not match the expected data. The simulation has run correctly if ERROR stays "0" throughout all SPI transfers. Note, however, that ERROR may assert some time during the simulation if the testbench is set up to test SS_IN_N assertion. This is due to the fact that the data received may be out of alignment with the expected data value.

The ModelSim command file, *func_sim.do,* can be used to open the correct waveform window and run the simulation.

CoolRunner-II CPLD Implementation

The CoolRunner-II SPI Master design has been targeted to a CoolRunner-II 256 macrocell device. The speed grade chosen is dependent on the system clock frequencies and should be analyzed by the designer to determine which speed grade is required.

The CoolRunner-II SPI Master utilizes 128 of the 256 macrocells available in the device, leaving 50% of the device resources for user logic.

The top level file for the SPI Master has been entered as a heirarchical VHDL file showing the connection between the *uc_interface* block and the *spi_interface* block. The *spi_interface* block is also a hierarchical VHDL fileshowing the inter connectivity between the *spi_control_sm* block, the *sck_logic* block, the *spi_rcv_shift_reg* block and the *spi_xmit_shift_reg* block. The structural VHDL models are found in the files *spi_master.vhd* and *spi_interface.vhd.*

Post-fit Timing Simulation

The Xilinx Project Navigator software package outputs a timing VHDL model of the fitted design. This post-fit VHDL was simulated with the original VHDL test benches to insure design functionality using ModelTech Xilinx Edition (MXE). Please note that all verification of this design has been done through simulations.

The user of this design is strongly encouraged to thoroughly inspect the timing report for this design to insure that the design meets the timing specification of the system. The user is also strongly encouraged to perform post-fit timing simulations as well. The ModelSim command file, *post_sim.do*, can be used to open the correct waveform window and run the simulation.

VHDL Code Download and Disclaimer

All VHDL source code, VHDL testbenches, and software files associated with this design are available. THE DESIGN IS PROVIDED TO YOU "AS IS". XILINX MAKES AND YOU RECEIVE NO WARRANTIES OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, AND XILINX SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. This design should be used only as an example design, not as a fully functional core. XILINX does not warrant the performance, functionality, or operation of this Design will meet your requirements, or that the operation of the Design will be uninterrupted or error free, or that defects in the Design will be corrected. Furthermore, XILINX does not warrant or make any

representations regarding use or the results of the use of the Design in terms of correctness, accuracy, reliability or otherwise.

THIRD PARTIES INCLUDING MOTOROLA MAY HAVE PATENTS ON THE SERIAL PERIPHERAL INTERFACE (SPI) BUS. BY PROVIDING THIS HDL CODE AS ONE POSSIBLE IMPLEMENTATION OF THIS STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THE PROVIDED IMPLEMENTATION OF THE SPI BUS IS FREE FROM ANY CLAIMS OF INFRINGEMENT BY ANY THIRD PARTY. XILINX EXPRESSLY DISCLAIMS ANY WARRANTY OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, AND XILINX SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE, THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OR REPRESENTATION THAT THE IMPLEMENTATION IS FREE FROM CLAIMS OF ANY THIRD PARTY. FURTHERMORE, XILINX IS PROVIDING THIS REFERENCE DESIGNS "AS IS" AS A COURTESY TO YOU.

XAPP386 - http://www.xilinx.com/products/xaw/coolvhdlq.htm

Conclusion

This document has detailed the design of a Serial Peripheral Interface Master design for a CoolRunner-II CPLD. Though the design has been extensively verified in simulations, Xilinx assumes no responsibility for the accuracy or the functionality of this design.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/24/02	1.0	Initial Xilinx release.