



SPI Core Specifications

Author: Richard Herveille
rherveille@opencores.org

Document rev. 0.1
January 7, 2003

This page left intentionally blank

Revision History

Rev.	Date	Author	Description
0.1	07/01/02	Richard Herveille	First Draft

Table of contents

INTRODUCTION	1
IO PORTS	2
REGISTERS	4
OPERATION	8
ARCHITECTURE	10

1

Introduction

The OpenCores simple Serial Peripheral Interface core is an enhanced version of the Serial Peripheral Interface found on Motorola's M68HC11 family of CPUs. The Serial Peripheral Interface is a serial, synchronous communication protocol that requires a minimum of 3 wires. Enhancements to the original interface include a wider supported operating frequency range, 4 entries deep read and write FIFOs, and programmable transfer count dependent interrupt generation. The high compatibility with the M68HC11 SPI port ensures that existing software can use this core without major modifications. New software can use existing examples as a starting point. The core features an 8 bit wishbone interface.

FEATURES:

- Compatible with Motorola's SPI specifications
- Enhanced M68HC11 Serial Peripheral Interface
- 4 entries deep read FIFO
- 4 entries deep write FIFO
- Interrupt generation after 1, 2, 3, or 4 transferred bytes
- 8 bit WISHBONE RevB.3 Classic interface
- Operates from a wide range of input clock frequencies
- Static synchronous design
- Fully synthesizable

IO ports

2.1 WISHBONE Interface Connections

Port	Width	Direction	Description
clk_i	1	Input	Master clock input
rst_i	1	Input	Asynchronous active low reset
inta_o	1	Output	Interrupt request signal
cyc_i	1	Input	Valid bus cycle
stb_i	1	Input	Strobe/Core select
adr_i	2	Input	Lower address bus bits
we_i	1	Input	Write enable
dat_i	8	Input	Data input
dat_o	8	Output	Data output
ack_o	1	Output	Normal bus termination

2.1.1 clk_i

All internal WISHBONE logic is registered to the rising edge of the [clk_i] clock input.

2.1.2 rst_i

The active low asynchronous reset input [rst_i] forces the core to restart. All internal registers are preset and all state-machines are set to an initial state.

2.1.3 inta_o

The interrupt request output is asserted when the core needs service from the host system.

2.1.4 cyc_i

When asserted, the cycle input [cyc_i] indicates that a valid bus cycle is in progress. The logical AND function of [cyc_i] and [stb_i] indicates a valid transfer cycle to/from the core.

2.1.5 stb_i

The strobe input [stb_i] is asserted when the core is being addressed. The core only responds to WISHBONE cycles when [stb_i] is asserted, except for the [rst_i], which always receive a response.

2.1.6 adr_i

The address array input [adr_i] is used to pass a binary coded address to the core. The most significant bit is at the higher number of the array.

2.1.7 we_i

When asserted, the write enable input [we_i] indicates that the current bus cycle is a write cycle. When negated, it indicates that the current bus cycle is a read cycle.

2.1.8 dat_i

The data array input [dat_i] is used to pass binary data from the current WISHBONE Master to the core. All data transfers are 8 bit wide.

2.1.9 dat_o

The data array output [dat_o] is used to pass binary data from the core to the current WISHBONE Master. All data transfers are 8 bit wide.

2.1.10 ack_o

When asserted, the acknowledge output [ack_o] indicates the normal termination of a valid bus cycle.

2.2 External (SPI Port) Connections

Port	Width	Direction	Description
sck_o	1	Output	SPI clock
mosi_o	1	Output	Master Out Slave In
miso_i	1	Input	Master In Slave Out

2.2.1 sck_o

SCK [sck_o] is generated by the master device and synchronizes data movement in and out of the device through the MOSI [mosi_o] and MISO [miso_o] lines. The SPI clock is generated by dividing the WISHBONE clock [clk_i]. The division factor is software programmable.

2.2.2 mosi_o

The Master Out Slave In line is a unidirectional serial data signal. It is an output from a master device and an input to a slave device.

2.5.3 miso_i

The Master In Slave Out line is a unidirectional serial data signal. It is an output from a slave device and an input to a master device.

M68HC11 compatibility

The M68HC11 features a Slave Select signal. To reduce resource usage, this core does not. Use a GPIO core, like the OpenCores simple_gpio core, to generate the Slave Select signal(s), if this functionality is required.

Registers

3.1 Registers List

Name	adr_i[1:0]	Width	Access	Description
SPCR	0x00	8	R/W	Control Register
SPSR	0x01	8	R/W	Status Register
SPDR	0x02	8	R/W	Data Register
SPER	0x03	8	R/W	Extensions Register

3.2 Serial Peripheral Control Register [SPCR]

Bit #	Access	Description
7	R/W	SPIE
6	R/W	SPE
5	R/W	<i>Reserved</i>
4	R/W	MSTR
3	R/W	CPOL
2	R/W	CPHA
1:0	R/W	SPR

Reset Value: 0x10

3.2.1 SPIE – Serial Peripheral Interrupt Enable

When the Serial Peripheral Interrupt Enable is set ('1') and the Serial Peripheral Interrupt Flag in the status register is set, the host is interrupted. Setting this bit while the SPIF flag is set generates an interrupt.

'0' = SPI interrupts disabled

'1' = SPI interrupts enabled

3.2.2 SPE – Serial Peripheral Enable

When the Serial Peripheral Enable bit is set ('1'), the core is enabled. When it is cleared ('0'), the core is disabled. The core only transfers data when the core is enabled.

'0' = SPI core disabled

'1' = SPI core enabled

3.2.3 MSTR – Master Mode Select

When the Master Mode Select bit is set ('1'), the core is a master device. When it is cleared ('0'), it is a slave device. Currently only master mode is supported. This bit is set, and can not be cleared.

'0' = Slave mode

'1' = Master mode

3.2.4 CPOL – Clock Polarity

The Clock Polarity bit, together with the Clock Phase bit, determines the transfer mode. Refer to the *SPI Transfers* section for more information

3.2.5 CPHA – Clock Phase

The Clock Phase bit, together with the Clock Polarity bit, determines the transfer mode. Refer to the *SPI Transfers* section for more information.

3.2.6 SPR – SPI Clock Rate Select

These bits select the SPI clock [sck_o] rate. Refer to the ESPR bits in the Extension Register for more information.

3.3 Serial Peripheral Status Register [SPSR]

Bit #	Access	Description
7	R/W	SPIF
6	R/W	WCOL
5:4	R	<i>Reserved</i>
3	R	WFFULL
2	R	WFEMPTY
1	R	RFFULL
0	R	RFEMPTY

Reset Value: 0x05

3.3.1 SPIF – Serial Peripheral Interrupt Flag

The Serial Peripheral Interrupt Flag is set upon completion of a transfer block. If SPIF is asserted ('1') and SPIE is set, an interrupt is generated. To clear the interrupt write the status register with the SPIF bit set ('1').

3.3.2 WCOL – Write Collision

The Write Collision flag is set when the Serial Peripheral Data register is written to, while the Write FIFO is full. To clear the Write Collision flag write the status register with the WCOL bit set ('1').

3.3.3 WFFULL – Write FIFO Full

The Write FIFO Full and Write FIFO empty bits show the status of the write FIFO.

3.3.4 WFEMPTY – Write FIFO Empty

The Write FIFO Full and Write FIFO empty bits show the status of the write FIFO.

3.3.5 RFFULL – Read FIFO Full

The Read FIFO Full and Read FIFO empty bits show the status of the read FIFO.

3.3.6 RFEMPTY – Read FIFO Empty

The Read FIFO Full and Read FIFO empty bits show the status of the read FIFO.

3.4 Serial Peripheral Data Register [SPDR]

Bit #	Access	Description
7:0	W	Write Buffer
7:0	R	Read Buffer

Reset Value: undefined

3.4.1 Write Buffer

The Write Buffer is a 4 entries deep FIFO. Writing to the Write Buffer adds the data to the FIFO. Writing to the Write Buffer while the FIFO is full sets the Write Collision [WCOL] bit. When the Serial Peripheral Enable [SPE] bit is cleared ('0'), the Write Buffer is reset. When the [SPE] bit is set ('1') and the write buffer is not empty, the core initiates SPI transfers. When the transfer is initiated, the data byte is removed from the FIFO.

3.4.2 Read Buffer

The Read Buffer is a 4 entries deep FIFO. When the Serial Peripheral Enable [SPE] bit is cleared ('0'), the Read Buffer is reset. When an SPI transfer is finished, the received data byte is added to the Read Buffer. There is no overrun detection; it is possible to overwrite the oldest data. This is done to maintain the highest level of compatibility with the M68HC11 type SPI port, and to minimize overhead for systems where the SPI bus is used to transfer data only (e.g. when accessing a DAC).

3.5 Serial Peripheral Extensions Register [SPER]

Bit #	Access	Description
7:6	R/W	ICNT
5:2	R/W	<i>Reserved</i>
1:0	R/W	ESPR

Reset Value: 0x00

3.5.1 ICNT – Interrupt Count

The Interrupt Count bits determine the transfer block size. The SPIF bit is set after ICNT transfers. Thus it is possible to reduce kernel overhead due to reduced interrupt service calls.

ICNT	Description
00	SPIF is set after every completed transfer
01	SPIF is set after every two completed transfers
10	SPIF is set after every three completed transfers
11	SPIF is set after every four completed transfers

3.5.2 ESPR – Extended SPI Clock Rate Select

The Extended SPI Clock Rate Select bits add two bits to the SPI Clock Rate Select bits. When ESPR = '00' the original M68HC11 coding is used.

ESPR	SPR	Divide WISHBONE clock by
00	00	2
00	01	4
00	10	16
00	11	32
01	00	8
01	01	64
01	10	128
01	11	256
10	00	512
10	01	1024
10	10	2048
10	11	4096
11	xx	<i>Reserved</i>

Errors in document are highlighted in yellow

NOTE

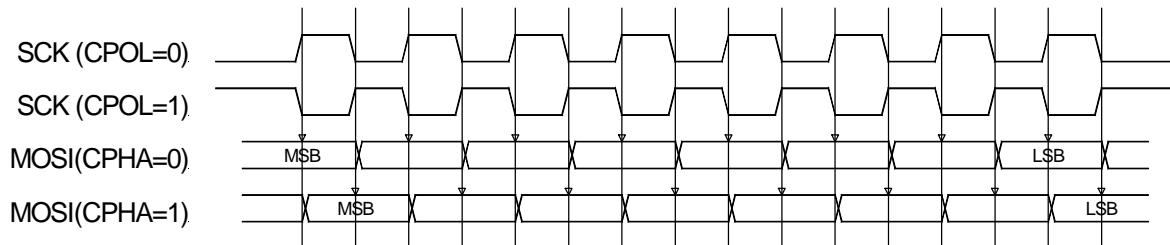
Do not use the reserved '11xx' values; using those leads to unpredictable results.

Operation

4.1 SPI Transfers

During an SPI transfer, data is simultaneously transmitted and received. The serial clock line [SCK] synchronizes shifting and sampling of the information on the two serial data lines. The master places the information onto the MOSI line a half-cycle before the clock edge that the slave device uses to latch the data.

Four possible timing relationships can be chosen by using the Clock Polarity [CPOL] and Clock Phase [CPHA] bits in the Serial Peripheral Control Register [SPCR]. Both master and slave devices must operate with the same timing.



4.2 Initiating transfers

4.2.1 Transmitting data bytes

After programming the core's control register SPI transfers can be initiated. A transfer is initiated by writing to the Serial Peripheral Data Register [SPDR]. Writing to the Serial Peripheral Data Register is actually writing to a 4 entries deep FIFO called the Write Buffer. Each write access adds a data byte to the Write Buffer. When the core is enabled – SPE is set ('1') – and the Write Buffer is not empty, the core automatically transfers the oldest data byte.

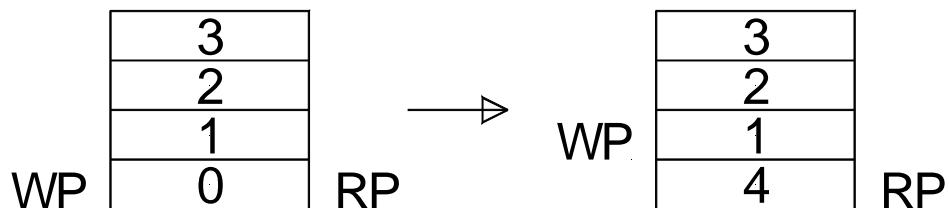
4.2.2 Receiving data bytes

Receiving data is done simultaneously with transmitting data; whenever a data byte is transmitted a data byte is received. For each byte that needs to be read from a device, a dummy byte needs to be written to the Write Buffer. This instructs the core to initiate an SPI transfer, simultaneously transmitting the dummy byte and receiving the desired data. Whenever a transfer is finished, the received data byte is added to the Read Buffer. The Read Buffer is the counterpart of the Write Buffer. It is an independent 4 entries deep FIFO. The FIFO contents can be read by reading from the Serial Peripheral Data Register [SPDR].

4.2.3 FIFO Overrun

Both the Write Buffer and the Read Buffer are FIFOs that use circular memories to simulate the infinite big memory needed for FIFOs. Because of this writing to a FIFO

while it is full overwrites the oldest data byte. Writing to the Serial Peripheral Data Register [SPDR] while the Write Buffer is full sets the WCOL bit, however the damage is already done; the next byte to be transferred is not the oldest data byte, but the latest (newest).



WP = FIFO Write Pointer

RP = FIFO Read Pointer

The only way to recover from this situation is to reset the Write Buffer. Both the Read Buffer and the Write Buffer are reset when the Serial Peripheral Enable [SPE] bit is cleared ('0').

Read Buffer overruns might be less destructive. Especially when the SPI bus is used to transmit data only; e.g. when sending data to a DAC. The received data is simply ignored. The fact that the Read Buffer overruns is irrelevant. If the SPI bus is used to transmit and receive data, it is important to keep the Read Buffer aligned. The easiest way to do this is to perform a number of dummy reads equal to the amount of bytes transmitted modulo 4.

$$N_{\text{dummy_reads}} = N_{\text{transmitted_bytes}} \bmod 4$$

Note that a maximum sequence of 4 bytes can be stored in the Read Buffer before the oldest data byte gets overwritten. It is therefore necessary to empty (read) the Read Buffer every 4 received bytes.

5

Architecture

