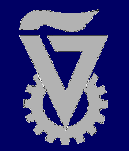
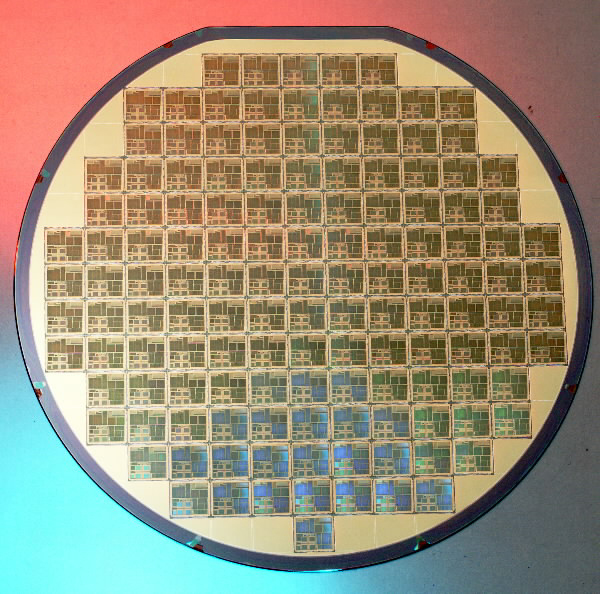
**VLSI SYSTEMS RESEARCH CENTER**

**VLSI LABORATORY**



**DEPARTMENT OF ELECTRICAL ENGINEERING DEPARTMENT OF COMPUTER SCIENCE**

**TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY**



**SPI – Serial Peripheral Interface**

**Final Report**

Submitted by:

Omer Shaked Beeri Schreiber

021618152 039848916

Supervised by:

Leon Polishuk

Tal Yahav

Submitted on: Date

Table of Contents

[List of Tables 4](#_Toc310801865)

[List of Figures 5](#_Toc310801866)

[Abstract 6](#_Toc310801867)

[List of Abbreviations 7](#_Toc310801868)

[1. Introduction 8](#_Toc310801869)

[1.1 Project Description 8](#_Toc310801870)

[1.2 SPI Protocol 8](#_Toc310801871)

[1.2.1 Interface 8](#_Toc310801872)

[1.2.2 Data Transmission 8](#_Toc310801873)

[1.2.3 Clock Polarity and Phase 9](#_Toc310801874)

[1.3 Wishbone Bus 9](#_Toc310801875)

[2. SPI Implementation 10](#_Toc310801876)

[2.1 SPI Master Design 10](#_Toc310801877)

[2.1.1 Activation Requirements 10](#_Toc310801878)

[2.1.2 Design Pinout 11](#_Toc310801879)

[2.1.3 Generic Parameters 12](#_Toc310801880)

[2.1.4 Resources 12](#_Toc310801881)

[2.2 SPI Master Waveforms 13](#_Toc310801882)

[2.2.1 Burst 13](#_Toc310801883)

[2.2.2 Start and End of Burst 14](#_Toc310801884)

[2.3 SPI Slave Design 15](#_Toc310801885)

[2.3.1 Activation Requirements 15](#_Toc310801886)

[2.3.2 Design Pinout 16](#_Toc310801887)

[2.3.3 Generic Parameters 16](#_Toc310801888)

[2.3.4 Resources 17](#_Toc310801889)

[2.4 SPI Slave Waveforms 17](#_Toc310801890)

[3. Communication over SPI 18](#_Toc310801891)

[3.1 Top Architecture 18](#_Toc310801892)

[3.2 Message Structure 18](#_Toc310801893)

[3.3 Master Host Design 18](#_Toc310801894)

[3.4 Slave Host Design 18](#_Toc310801895)

[4. Verification 19](#_Toc310801896)

[4.1 SPI Master Verification 19](#_Toc310801897)

[4.2 SPI Slave Verification 19](#_Toc310801898)

[4.3 SPI Top Verification 19](#_Toc310801899)

[4.4 Architecture Top Verification 19](#_Toc310801900)

[5. Summary 20](#_Toc310801901)

[6. Bibliography 21](#_Toc310801902)

[7. Appendix A – Block-Level Design 22](#_Toc310801903)

[7.1 Wishbone Slave 22](#_Toc310801904)

[7.2 Message Pack Encoder 22](#_Toc310801905)

[7.3 Message Pack Decoder 22](#_Toc310801906)

[7.4 RAM Controller 22](#_Toc310801907)

# List of Tables

Table 1 - SPI Master Pinout Description 11

Table 2 - SPI Master Generic Parameters 12

Table 3 - SPI Slave Pinout Description 16

Table 4 - SPI Slave Generic Parameters 17

# List of Figures

Figure 1 - SPI Interface 8

Figure 2 - SPI Protocol Diagram 9

Figure 3 - SPI Master Pinout 11

Figure 4 - SPI Master Burst Waveform 13

Figure 5 - SPI Master Start and End of Burst Waveform 14

Figure 6 - SPI Slave Pinout 16

# Abstract

The communication links across components of a chip or a board may be either serial or parallel. Serial communication is performed over fewer interconnecting cables, thus enables to save a significant amount of space and reduce the number of connecting pins. These reasons have led the serial communication interfaces to play a major role in the field of embedded systems.

This project implements the Serial Peripheral Interface, a "four-wire" serial data link standard, which operates in full duplex mode. The communication is performed in a master / slave mode, where the master device initiates the transactions.

In order to avoid using of gated-clocks, and reducing the standard's dependency on synchronizers, it was implemented as Asynchronous data link.

In order to demonstrate the use of serial data links, there was also designed a communication protocol. The protocol receives a data-exchange request from parallel Wishbone bus, and performs it over the Serial Peripheral Interface.

# List of Abbreviations

SPI Serial Peripheral Interface

RAM Random Access Memory

UVM Universal Verification Methodology

IP Intellectual Property

SOC System On Chip

LSB Least Significant Bit

MSB Most Significant Bit

FIFO First In First Out

MOSI Master Output Slave Input

MISO Master Input Slave Output

TB Test Bench

# 1. Introduction

The field of electronic systems is facing a constant demand for better performance achieved by systems becoming smaller in size. This results in the increase of both complexity and density of electronic hardware, including semiconductor chips and circuit boards. Those trends are emphasizing the importance of saving both space and power.

One common way of saving both space and power, within a chip or a board, is to use serial communication. In contrast to parallel communication, which demands a large amount of wires and connecting pins, serial communication can be performed by using only minimal amount of interconnections.

## 1.1 Project Description

Within this project, a serial data link standard named Serial Peripheral Interface was implemented using VHDL, and later verified using SystemVerilog and UVM. The straightforward implementation requires the use of gated-clocks, which wanted to be avoided, and therefore a different design approach was taken. With this approach, the SPI was implemented as A-synchronous serial data link.

In addition, as an example of possible use, a communication protocol was created and designed using VHDL. This protocol demonstrates the ability to transform incoming parallel communication from a Wishbone bus, into serial communication that is being transferred using the SPI protocol.

## 1.2 SPI Protocol

The Serial Peripheral Interface bus is a serial data link standard named by "Motorola" that operates in full duplex mode. Devices communicate in [master / slave](http://en.wikipedia.org/wiki/Master-slave_(technology)) mode where the master device initiates the [data frame](http://en.wikipedia.org/wiki/Data_frame). Multiple slave devices are allowed with individual [slave select](http://en.wikipedia.org/wiki/Slave_select) ([chip select](http://en.wikipedia.org/wiki/Chip_select)) lines.

### 1.2.1 Interface

The SPI bus specifies four logic signals:

* **SPI\_CLK**: Serial Clock (output from master).
* **SPI\_MOSI, SPI\_SIMO**: Master Output, Slave Input (output from master).
* **SPI\_MISO, SPI\_SOMI**: Master Input, Slave Output (output from slave).
* **SPI\_SS**: [Slave Select](http://en.wikipedia.org/wiki/Slave_Select) ([active low](http://en.wikipedia.org/wiki/Logic_level), output from master).



Figure - SPI Interface

### 1.2.2 Data Transmission

To begin communication, the master first configures the clock, using a frequency less than or equal to the maximum frequency the slave device supports. Such frequencies are commonly in the range of 1–70 MHz.

The master then pulls the chip select (SPI\_SS) low for the desired chip.

During each SPI clock cycle, a [full duplex](http://en.wikipedia.org/wiki/Full_duplex) data transmission occurs:

* The master sends a bit on the MOSI line. The slave reads it from that same line.
* The slave sends a bit on the MISO line. The master reads it from that same line.

### 1.2.3 Clock Polarity and Phase

In addition to setting the clock frequency, the master must also configure the clock polarity (CPOL) and phase (CPHA) with respect to the data.

* At CPOL=0 the base value of the clock is '0'.
  + For CPHA=0, data are captured on the clock's rising edge (low → high clock transition) and data are propagated on a falling edge (high → low clock transition).
  + For CPHA=1, data are captured on the clock's falling edge and data are propagated on a rising edge.
* At CPOL=1 the base value of the clock is '1' (inversion of CPOL=0).
  + For CPHA=0, data are captured on the clock's falling edge and data are propagated on a rising edge.
  + For CPHA=1, data are captured on clock's rising edge and data are propagated on a falling edge.

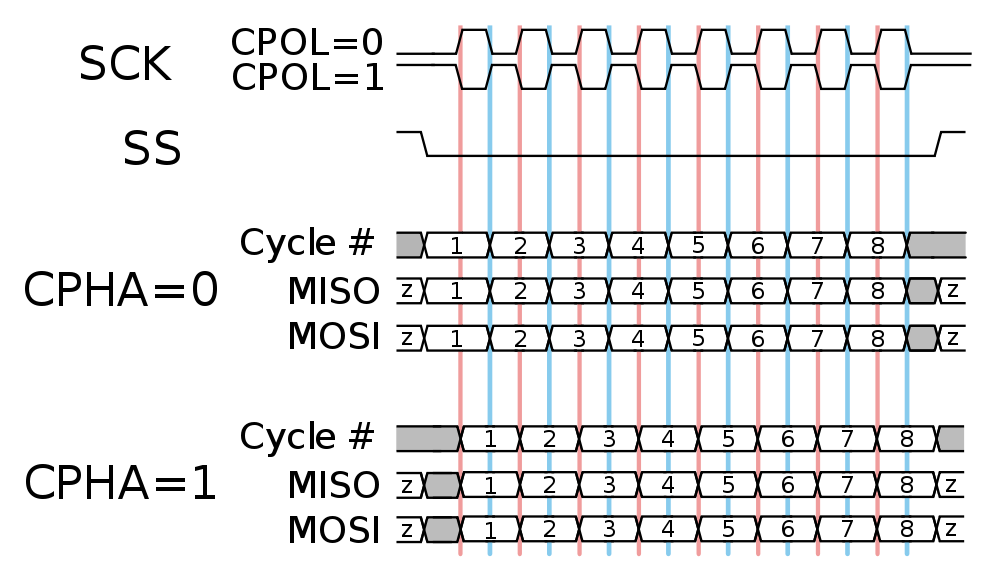


Figure 2 - SPI Protocol Diagram

That is, CPHA=0 means sample on the leading (first) clock edge, while CPHA=1 means sample on the trailing (second) clock edge, regardless of whether that clock edge is rising or falling. Note that with CPHA=0, the data must be stable for a half cycle before the first clock cycle. For all CPOL and CPHA modes, the initial clock value must be stable before the chip select line goes active.

The MOSI and MISO signals are usually stable (at their reception points) for the half cycle until the next clock transition. SPI master and slave devices may well sample data at different points in that half cycle.

This adds more flexibility to the communication channel between the master and slave.

## 1.3 Wishbone Bus

The Wishbone Bus, created by "Silicore Corporation" is an open source hardware computer bus, intended to allow parallel communication between the parts of an integrated circuit. This System-on-Chip interconnection architecture is used in order to create a common interface between different IP cores.

The Wishbone interconnect is intended as a general purpose interface. As such, it defines a master / slave standard for data exchange between IP core modules, in terms of signals, clock cycles, and high & low levels.

# 2. SPI Implementation

The SPI design contains two separate cores – the **SPI Master** core and the **SPI Slave** core. The SPI protocol allows the use of a single master, together with multiple slaves. All the slaves MISO output signals are connected to the same wire via Tri-State Buffers. Therefore, while the SPI Slave Select signal of a slave is not active, the slave's MISO output port MUST be at 'High Z' state.

Usually, the SPI Master core will be attached to the system's main processor, and will be used to communicate with either on-board or on-chip (in SOC's) peripherals, that will incorporate SPI Slave cores.

The SPI design supports all four CPOL and CPHA configurations. In addition, it allows a variety of features:

* Connecting different number of SPI Slaves.
* Changing word sizes.
* Sending first the LSB or the MSB.

The user of the SPI cores MUST maintain coherency of CPOL and CPHA values for both SPI Master and all of the SPI Slaves, in order for the protocol to work properly.

The SPI design supports **maximum SPI\_CLK frequency of SYSTEM\_CLK / 4**, as both the master and the slaves share the same SYSTEM\_CLK. If different clock domains are used by the master and slaves, the maximum supported frequency is SLAVE\_SYSTEM\_CLK / 4.

***Note***: In case of different clock domains, it is essential to add filter for the SPI signals, since the MASTER's signals, in this case, are a-synchronous to the SLAVE's signals.

## 2.1 SPI Master Design

SPI Master is consumed of the following interfaces:

* Clock and Reset Interfaces.
* FIFO Interface - Data from FIFO will be transmitted through the SPI\_MOSI line.
* Registers Interface, to control internal SPI Core Configurations.
* SPI Interface, according to SPI Protocol.
* Received Data Interface, to transfer the data received from SPI Slave.
* Miscellaneous – SPI Slave Address, Busy signal.

As long as the input FIFO is not empty, data will be transmitted to the relevant SPI Slave, which is determined by SPI slave address.

When transaction is active, 'BUSY' signal will be asserted ('1').

The following registers may be modified during normal operation, when there is no active transmission:

* Configuration Register - to configure CPOL and CPHA:
  + Bit 0 - CPHA
  + Bit 1 - CPOL
* Clock Divide Register - to configure system clock divide factor to the SPI clock. I.e. suppose divide factor is 2, then each two system clock cycles (4 system clocks events), SPI\_CLK will change its polarity.

### 2.1.1 Activation Requirements

In order for the SPI Master to work properly, the following requirement must be met:

* FIFO should assert *FIFO\_DIN\_VALID* within one system clock from *FIFO\_REQ\_DATA*.

### 2.1.2 Design Pinout

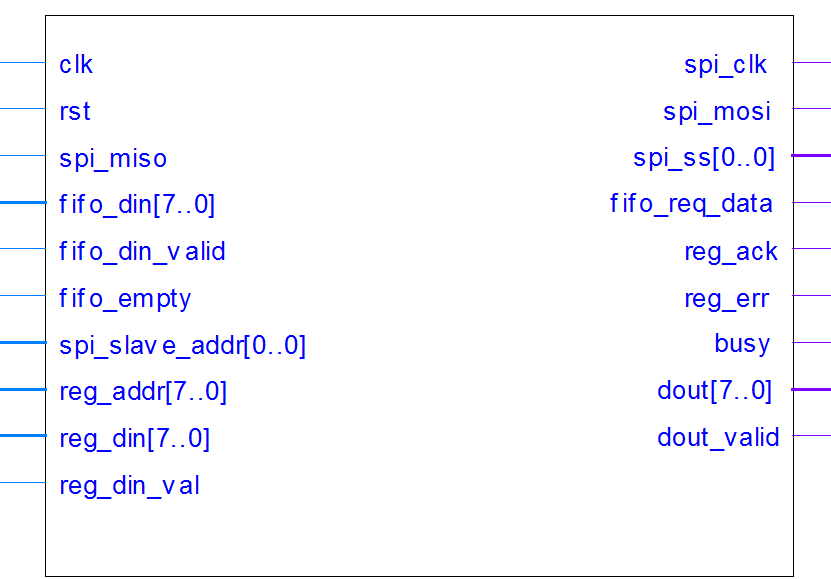


Figure - SPI Master Pinout

| Pin Name | Direction | Description |
| --- | --- | --- |
| clk | In | System Clock |
| rst | In | Reset (active according to Reset polarity generic) |
| spi\_clk | Out | SPI Output Clock |
| spi\_mosi | Out | SPI Master Output Slave Input |
| spi\_miso | In | SPI Master Input Slave Output |
| spi\_ss [] | Out | SPI Slave Select |
| fifo\_req\_data | Out | Request for data from FIFO |
| fifo\_din [] | In | Input data from FIFO |
| fifo\_din\_valid | In | '1' when input data from FIFO is valid (wil be '1' for one clock cycle, with delay of one clock cycle from *fifo\_req\_data*) |
| fifo\_empty | In | '1' when FIFO is empty |
| spi\_slave\_addr [] | In | SPI Addressed Slave. Feed with the NUMBER of the slave (i.e, for slave number 3, set value of "11") |
| reg\_addr [] | In | Address to Internal Configurations Registers |
| reg\_din [] | In | Input data to Configurations Registers |
| reg\_din\_val | In | Data to registers is valid (should be asserted for one clock cycle) |
| reg\_ack | Out | Register Data has been acknowledged |
| reg\_err | Out | Error while writing data to registers |
| busy | Out | SPI Master is in the middle of a transaction |
| dout [] | Out | Output data, which has been received from SPI Slave |
| dout\_valid | Out | Asserted when *dout* is valid |

Table - SPI Master Pinout Description

### 2.1.3 Generic Parameters

| Generic Parameter | Default Value | Description |
| --- | --- | --- |
| **reset\_polartiy\_g** | '0' | Reset active in this polarity |
| **ss\_polarity\_g** | '0' | Slave select polarity is active at this value |
| **data\_width\_g** | 8 | Input / Output parallel data |
| bits\_of\_slaves\_g | 1 | Number of bits for SPI\_SS |
| reg\_width\_g | 8 | Number of bits in SPI Configurations Register |
| dval\_conf\_reg\_g | 0 | Default (initial) value of Configuration Register |
| dval\_clk\_reg\_g | 2 | Default (initial) value of Clock Divide Register (Divide system clock by 2 is the minimum) |
| reg\_addr\_width\_g | 8 | Registers Configuration Address Width |
| reg\_din\_width\_g | 8 | Registers Configuration Input Data Width |
| **first\_dat\_lsb** | true | TRUE: Transmit and Receive LSB first. FALSE - MSB first |

Table - SPI Master Generic Parameters

**NOTE:** The highlighted generic parameters must be assigned the same values in both SPI Master and SPI Slave cores.

### 2.1.4 Resources

Required resources, when synthesizing, using Synplify, for Altera's Cyclon II FPGA:

* Total combinational functions:  102
* Logic element usage by number of inputs:
  + 4 input functions: 46
  + 3 input functions: 29
  + [=2 input functions: 27
* Logic elements by mode:
  + Normal mode:             85
  + Arithmetic mode: 17
* Total registers: 81
* I/O pins: 47

**Maximum Working Frequency**: 269MHz

## 2.2 SPI Master Waveforms

### 2.2.1 Burst

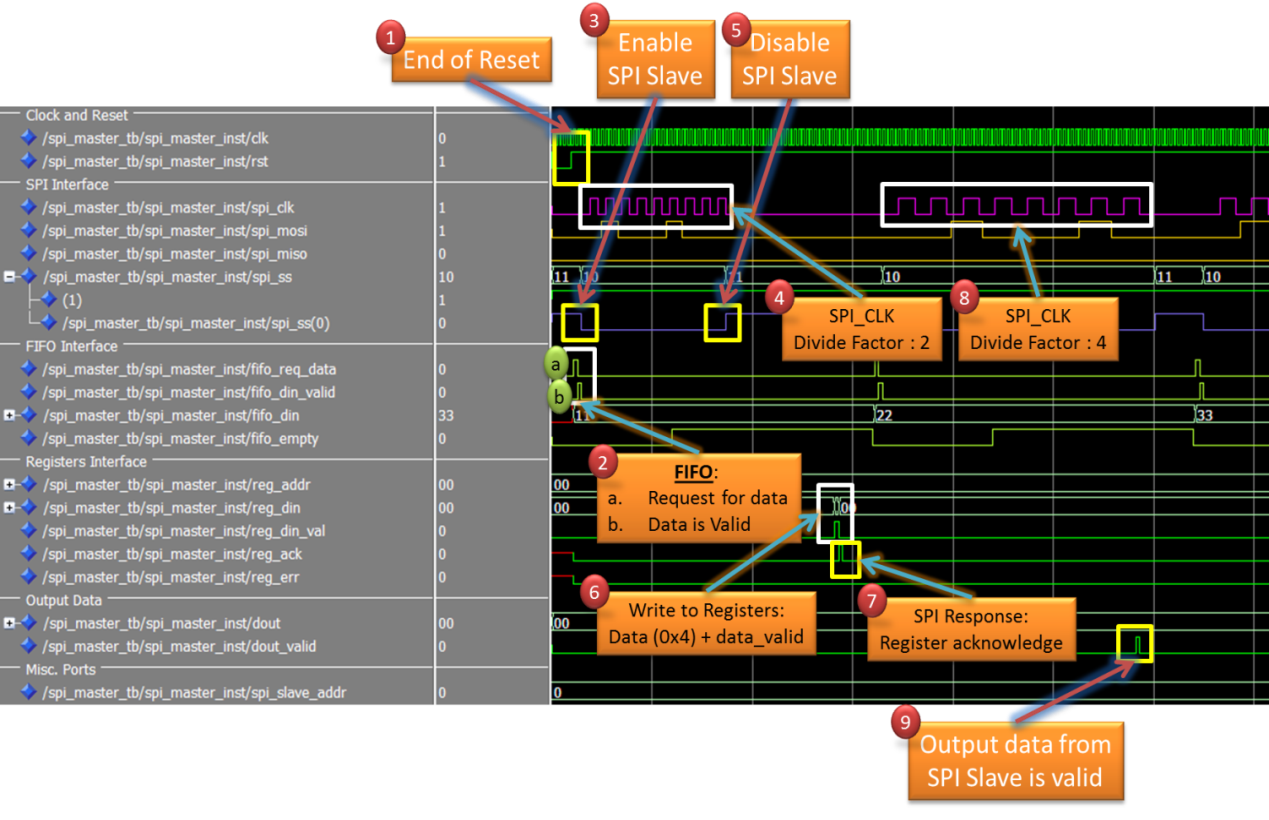


Figure - SPI Master Burst Waveform

1. End of System Reset.
2. FIFO Interface:
   1. SPI Master requests for data from FIFO.
   2. FIFO confirms that output data is valid.
3. SPI Slave Select for Slave [0] is asserted ('0').
4. SPI Clock is generated. In this waveform, the divide factor, from whole clock cycle, is 2.
5. End of first burst – negate ('1') SPI\_SS.
6. Configure SPI Master: Change clock divide factor to 4.
7. SPI Master response – reg\_ack is asserted.
8. SPI Clock is generated, this time with divide factor of 4.
9. Data from SPI Slave has been received, and is now valid as parallel data from SPI Master.

### 2.2.2 Start and End of Burst

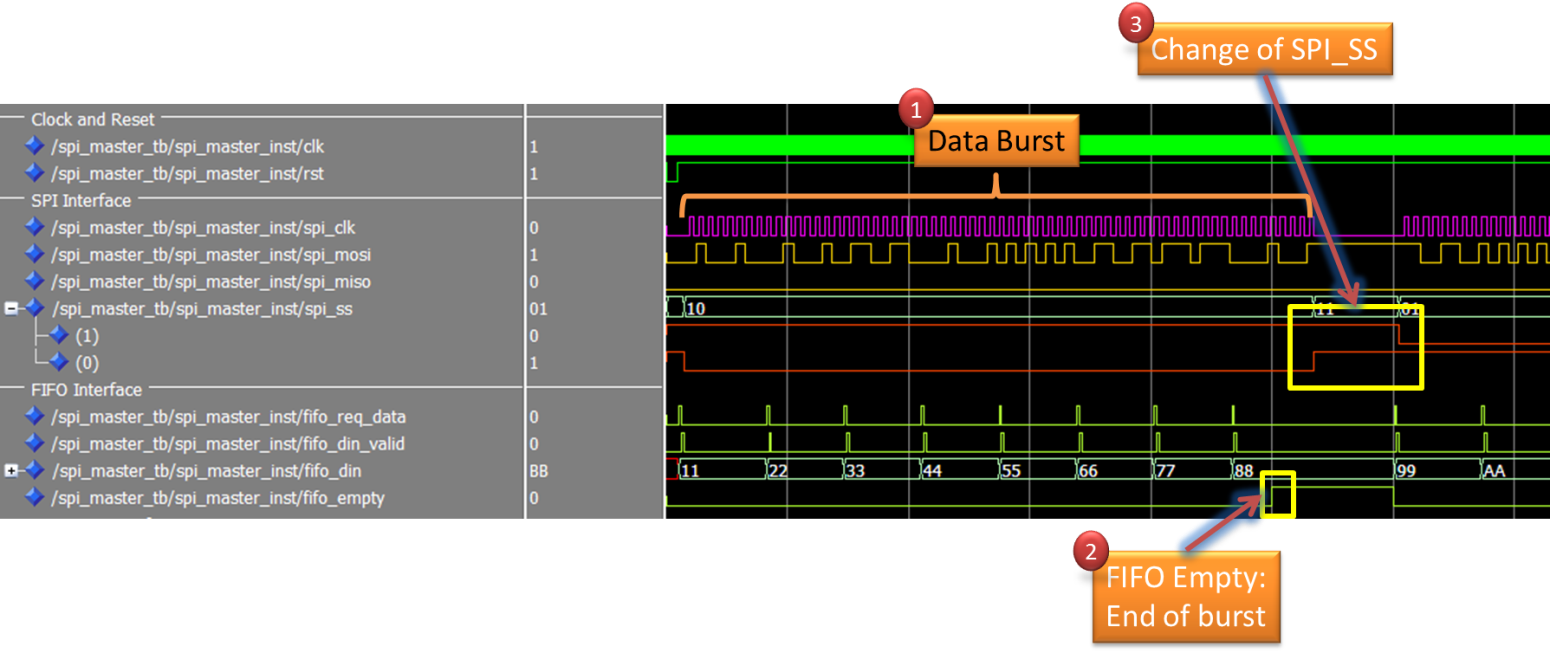


Figure - SPI Master Start and End of Burst Waveform

1. Data Burst.
2. FIFO is empty and causes the end of burst.
3. New data is available. This time, the second slave is addressed.

## 2.3 SPI Slave Design

SPI Slave is consumed of the following interfaces:

* Clock and Reset Interfaces.
* FIFO Interface - Data from FIFO will be transmitted through the SPI\_MISO line.
* Registers Interface, to control internal SPI Core Configurations.
* SPI Interface, according to SPI Protocol.
* Received Data Interface, to transfer the data received from SPI Master.
* Miscellaneous – Busy signal, Timeout signal and Interrupt signal.

As long as the SPI Slave Select signal is active, data will be transmitted from the SPI Slave through the SPI\_MISO line, and data from the SPI Master will be received through the SPI\_MOSI line.  
In each transaction, one data word is being transferred. At the end of each transaction, SPI\_SS signal can remain active, and therefore the next transaction will immediately follow the previous one. At the beginning of each word transmission, the SPI Slave will act as follows:

* If FIFO is not empty – data from the FIFO will be loaded into the SPI Slave, and transmitted to the SPI Master via the SPI protocol.
* If FIFO is empty – default data word will be transmitted to the SPI Master via the SPI protocol.

The configuration register includes the following fields:

* Bit 0 – CPHA
* Bit 1 – CPOL

The register can be modified at any time, but if it was modified during active transmission, the modification will only take effect at the end of the transmission.

In addition, the SPI Slave also outputs the following signals:

* Timeout signal is used to handle the case of SPI\_SS getting stuck at active value, while the master isn't initiating any transmission. A timeout counter counts the number of SYSTEM\_CLK cycles that went through without any SPI\_CLK toggle. If the counter reaches its limit, the Timeout signal turns active.
* Interrupt signal is used to indicate that the SPI\_SS signal turned not active during the middle of a transmission, and therefore the SPI Slave didn't receive a full valid data word.

### 2.3.1 Activation Requirements

In order for the SPI Slave to work properly, the following requirement must be met:

* FIFO should assert *FIFO\_DIN\_VALID* within one system clock from *FIFO\_REQ\_DATA*.

### 2.3.2 Design Pinout

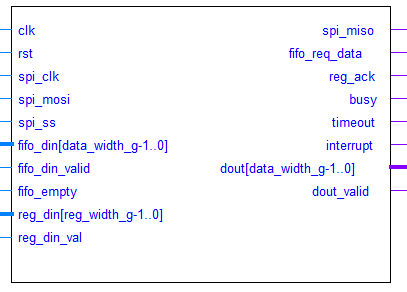


Figure - SPI Slave Pinout

| Pin Name | Direction | Description |
| --- | --- | --- |
| clk | In | System Clock |
| rst | In | Reset (active according to Reset polarity generic) |
| spi\_clk | In | Input SPI Clock from SPI Master |
| spi\_mosi | In | SPI Master Output Slave Input |
| spi\_miso | Out | SPI Master Input Slave Output |
| spi\_ss | In | SPI Slave Select (active according to SS polarity generic) |
| fifo\_req\_data | Out | Request for data from FIFO |
| fifo\_din [] | In | Input data from FIFO |
| fifo\_din\_valid | In | '1' when input data from FIFO is valid (will be '1' for one clock cycle, with delay of one clock cycle from *fifo\_req\_data*) |
| fifo\_empty | In | '1' when FIFO is empty |
| reg\_din [] | In | Input data to Configuration Register |
| reg\_din\_val | In | Data to register is valid (should be asserted for one clock cycle) |
| reg\_ack | Out | Register Data has been acknowledged |
| busy | Out | SPI Slave is in the middle of a transaction |
| timeout | Out | '1' when *spi\_clk* is stuck at the same value for a defined number of consecutive System Clock cycles |
| interrupt | Out | '1' when *spi\_ss* turned NOT Active in a middle of a transaction |
| dout [] | Out | Output data, which has been received from SPI Master |
| dout\_valid | Out | Asserted when *dout* is valid |

Table - SPI Slave Pinout Description

### 2.3.3 Generic Parameters

| Generic Parameter | Default Value | Description |
| --- | --- | --- |
| **reset\_polartiy\_g** | '0' | Reset is active in this polarity |
| **ss\_polarity\_g** | '0' | Slave select is active at this value |
| **data\_width\_g** | 8 | Width in bits of a basic data word (Determines the size of Input / Output Shift Registers) |
| reg\_width\_g | 8 | Number of bits in SPI Configuration Register |
| dval\_cpha\_g | '0' | Default (initial) value of CPHA |
| dval\_cpol\_g | '0' | Default (initial) value of CPOL |
| **first\_dat\_lsb** | true | TRUE: Transmit and Receive LSB first. FALSE - MSB first |
| default\_sat\_g | 0 | Default data transmitted to SPI Master when the FIFO is empty |
| spi\_timeout\_g | 0x"020" | The number of System Clock cycles without *spi\_clk* toggle that causes the activation of *timeout* signal |
| timeout\_en\_g | '1' | If '1' timeout is enabled, else disabled |
| dval\_miso\_g | '0' | Default value of *spi\_miso* internal signal |

Table - SPI Slave Generic Parameters

**NOTE:** The highlighted generic parameters must be assigned the same values in both SPI Master and SPI Slave cores.

### 2.3.4 Resources

Required resources, when synthesizing, using Synplify, for Altera's Cyclon II FPGA:

* Total combinational functions:  116
* Logic element usage by number of inputs:
  + 4 input functions: 75
  + 3 input functions: 20
  + [=2 input functions: 21
* Logic elements by mode:
  + Normal mode:             100
  + Arithmetic mode: 16
* Total registers: 61
* I/O pins: 39

**Maximum Working Frequency**: 219.7MHz

## 2.4 SPI Slave Waveforms

### 2.4.1 Single Transaction

Figure - SPI Slave Single Transaction Waveform

1. End of System Reset, SPI Slave Select is NOT active, and therefore the output of *spi\_miso* is at 'High Z' value.
2. FIFO Interface:
   1. SPI Slave requests for data from FIFO.
   2. FIFO confirms that output data is valid after 1 system clock cycle.
3. SPI Master activates the *spi\_ss* signal and starts a new transaction.
4. During the transaction, the SPI Slave is waiting for *spi\_clk* edges:
   1. On a falling edge – the next output data bit is propagated into the *spi\_miso* line.
   2. On a rising edge – the next input data bit is sampled from the *spi\_mosi* line.
5. End of the transaction – a valid data word received from the SPI Master is transferred to the *dout* bus.
6. The master ends the connection – *spi\_ss* is de-activated and therefore *spi\_miso* returns to 'High Z'.

### 2.4.2 Multiple Transmissions Burst

Figure - SPI Slave Multiple Transmissions Burst

1. End of System Reset, SPI Slave Select is NOT active, and therefore the output of *spi\_miso* is at 'High Z' value.
2. SPI Slave loads data from the FIFO, which then turns empty.
3. SPI Master activates the *spi\_ss* signal and starts a new transaction.
4. The last *spi\_clk* of the 1st transaction.
5. End of the 1st transaction – a valid data word received from the SPI Master is transferred to the *dout* bus.
6. The master keeps the *spi\_ss* signal activated, and the *spi\_clk* starts to toggle:
   1. New transaction immediately starts.
   2. Since the FIFO is empty – default data is being sent.

# 3. Communication over SPI

One common way of using serial communication interfaces is by breaking down messages that were sent over a parallel interface, and sending them over a serial interface instead. This enables us to save a significant amount of connecting wires and pins, and is mainly useful when connecting two components that are located far away from each other.

The scenario implemented within this project can be described as a main processing unit that wants to communicate with a remote memory. The data-exchange messages from the main unit are sent via parallel Wishbone interface, but instead of driving the wide Wishbone bus over the entire board, the required actions are being performed over a serial interface.

The system is consumed of master and slave hosts. The master host implements a Wishbone Slave interface. It receives the messages via the Wishbone parallel interface, and uses the SPI protocol to connect with a slave host, that is attached to the external RAM. The slave host then performs the required actions, and if required, sends data back to the master host via SPI interface. After the action has been performed, the master host confirms the successful completion of the instruction over the Wishbone interface.

The implemented architecture supports 3 different data-exchange instructions:

* Read from memory – data is read from the external RAM, and sent back over the Wishbone interface.
* Write to memory – data is received over the Wishbone interface, and written to the external RAM.
* Configure register – new CPOL, CPHA values are written into the configuration register of the SPI Slave. This enables to configure the SPI Slave internally.

In order to perform those instructions over a serial interface, a new message structure had to be defined, and handled by both the master and slave hosts. The master host is gathering all the received information from the Wishbone interface, including action type, data and address, and builds a message that contains all the required information for the slave host. The slave host then unpacks the message, performs the action, and in case of RAM read request, sends a message back to the master host.

## 3.1 Top Architecture

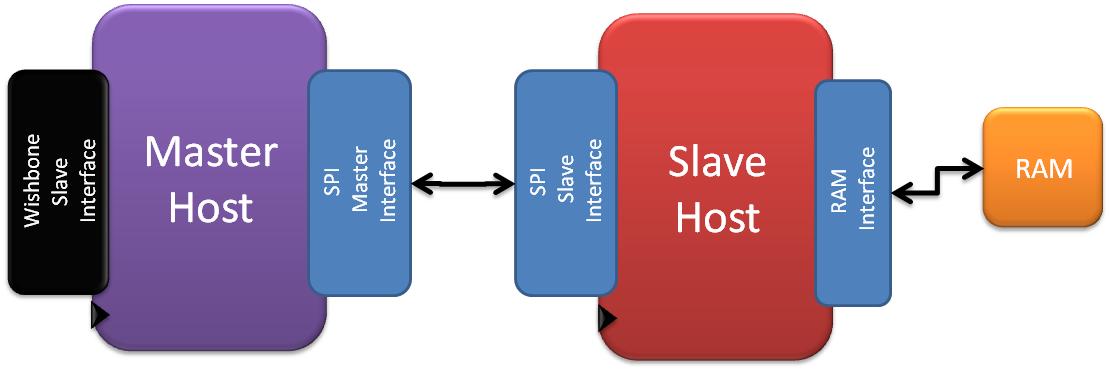


Figure - Top Architecture

The system consists of 3 main blocks:

* Master Host – has two main interfaces:
  + Wishbone Slave Interface – through this interface the master host communicates of a Wishbone Master that initiates the parallel Wishbone transactions.
  + SPI Master – through this interface data is transferred between the master host and the slave host.
* Slave Host – has two main interfaces:
  + SPI Slave – connects the slave host and the master host.
  + RAM Interface – through this interface data is read from, or written to the external RAM.
* External RAM.

## 3.2 Message Structure

Several different fields are received by the master host via the Wishbone Slave interface. These fields include:

* Instruction type – RAM read, RAM write or Register Configuration.
* Address – for RAM read or write.
* Burst size – the number of bytes to be written into or read from the RAM.
* Data – in case of RAM write or Register Configuration.

These instruction fields must be delivered over the SPI to the slave host, in order for the slave host to perform the correct actions. This requires the definition of a communication protocol that will be used between the master and slave host.

The protocol defines a **SPI Message** that is consumed of the following blocks:

1. **SOF** – Start of Frame (Head).
2. **Type** – Message type (data read / data write / configuration).
3. **Address** – The base address for the external RAM read or write.
4. **Data length** – Data (Payload) length, the size of the Data block within the message:
   1. RAM Write – the field will contain the number of data bytes that needs to be written into the RAM, reduced by one. Meaning, that if we take the first data byte to have the address of 0x00, then in a case of 256 bytes burst, the address of the last byte is 0xFF, and this will be the value of the Data Length field.
   2. RAM Read / Configuration – the field will contain a value of 0x00, meaning that the Data payload is of one byte only.
5. **Data (Payload)** – the data block. It's contents depend on the type of the message:
   1. RAM Write – the block will contain all the data that will be written to the RAM, starting from the base address.
   2. RAM Read – the block will contain the size of the read burst reduced by one. For example, if the value of the data byte is 0xFF, then 256 bytes needs to be read from the RAM, starting from the base address.
6. **Checksum** – Checksum will be used as a method of detecting errors. Checksum value will be calculated from the Type block to the Data block, inclusive.
7. **EOF** – End of frame (Tail).

Figure - Message Structure

## 3.3 Master Host Design

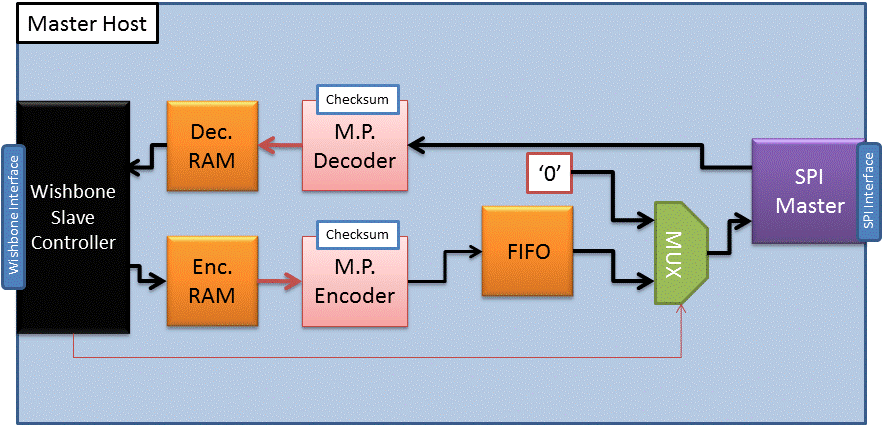


Figure - Master Host Design Blocks

The Master Host consists of the following blocks:

* Wishbone Slave Controller.
* Encoder and Decoder RAM's.
* Message Pack Encoder and Decoder.
* Checksum Computation Unit.
* FIFO.
* SPI Master.

Detailed block description can be found at Appendix A.

### 3.3.1 Master Host Packet Walkthrough

1. The Master Host receives a new Wishbone command from the Wishbone Slave interface.
2. The Wishbone Slave Controller determines the value of the Type, Address and Data Length fields of the SPI Message, and writes all the received data into the Encoder RAM.
3. The Message Pack Encoder collects all the information from the Wishbone Slave Controller and all the data from the Encoder RAM, and encodes it into a single SPI Message.
4. The packed SPI Message is written into the FIFO.
5. Since the FIFO contains valid data, the SPI Master initiates a SPI transaction.
6. All the data from the FIFO is read by the SPI Master and sent to the SPI Slave. Meanwhile, the SPI Slave sends default data, which is ignored by the Message Pack Decoder.
7. When the entire SPI Message has been sent, the FIFO empties. In case of RAM write or Configuration commands, the Master Host doesn't wait for confirmation. The SPI Master ends the transaction, and we go directly to stage 11.
8. In case of RAM read command, the SPI Master has to keep the SPI transaction active in order to receive the data, since the SPI Slave can't initiate a transaction. Therefore, default data is being sent to the SPI Slave.
9. After the entire SPI Message, which includes the data that was read from the RAM, will arrive to the Master Host, the Message Pack Decoder will notice the EOF sequence. Then, the SPI Master will be notified to close the SPI transaction.
10. The Message Pack Decoder will write the received data into the Decoder RAM, and validate that no errors occurred in the SPI Message using the Checksum unit.
11. The Wishbone Slave Controller will drive acknowledge signal to the Wishbone interface, in order to notify the Wishbone Master that the command has been completed. In case of RAM read, the data will be read from the Decoder RAM, and will be transmitted through the Wishbone interface.

## 3.4 Slave Host Design

Figure - Slave Host Design Blocks

The Slave Host is consumed of the following blocks:

* Slave Host Controller.
* RAM Controller.
* Encoder and Decoder RAM's.
* Message Pack Encoder and Decoder.
* Checksum Computation Unit.
* FIFO.
* SPI Slave.

Detailed block description can be found at Appendix A.

### 3.4.1 Slave Host Packet Walkthrough

1. The SPI Slave wakes up when the SPI Master starts a transaction.
2. The SPI Slave starts receiving the SPI Message sent from the SPI Master. Meanwhile, the FIFO is empty, therefore default data is sent back to the SPI Master.
3. The Message Pack Decoder receives the SPI Message and writes the data into the Decoder RAM.
4. After receiving EOF byte from the SPI Master, if this is a RAM write or Configuration command, the Master will close the transaction. On a RAM read command, Master will keep the transaction active, and start sending default data, which is ignored by the Message Pack Decoder.
5. When the Message Pack Decoder recognizes EOF, it performs Checksum calculation to validate that the message is valid, and if so it sends an indication that a valid SPI Message has been received.
6. The Slave Host Controller checks the command type register:
   1. RAM Write – the RAM controller will perform the entire action by itself. It will read the address and burst length registers, and then read the data from the Decoder RAM, and write it to the External RAM. The RAM Controller will notify the Slave Host Controller when the action has been completed.
   2. RAM Read – the RAM Controller will read the address register, and read the burst length from the Decoder RAM. Then, the data will be read from the External RAM, and written into the Encoder RAM. After all the data has been successfully read, the RAM Controller will notify the Slave Host Controller that the action has been completed.
   3. Register Configuration – the Slave Host Controller will perform the action. It will read the new register value from the Decoder RAM, and write it into the SPI Slave Configuration Register.
7. In case of RAM Read or Configuration command, the transaction is finished, and the Slave Host is ready to receive a new message. In case of RAM Read, the Slave Host Controller will notify the Message Pack Encoder that all the data is ready, and will send him the length of the data burst that was read.
8. The Message Pack Encoder will build a returning RAM Read SPI Message, containing the Data Length value that was received from the Slave Host Controller, and the data that was written into the Encoder RAM.
9. The packed SPI Message will be written into the FIFO, and sent by the SPI Slave to the SPI Master.
10. When the Master Host will recognize that EOF has been received from the Slave Host, the SPI Master will close the transaction with the SPI Slave. Then, the Slave Host will be ready to receive a new message.

# 4. Verification

As a second part of this project, comprehensive verification has been performed on the implemented design. The major effort was put into verifying the SPI Master and Slave cores, since those are the main deliverables from this project. A secondary effort was put on verifying the entire design architecture, including the Master and Slave Hosts.

Three main methods were used in order to perform the verification:

1. VHDL Test Benches were used to perform basic operation tests on individual design blocks.
2. SystemVerilog verification environment was built and used to perform extensive verification of the SPI Master and Slave cores. The cores were first verified separately, and then verified together at the configuration of one SPI Master that is connected to 4 SPI Slaves.
3. UVM 1.1 class libraries were used in order to build a SystemVerilog testing environment and perform verification of the Top Architecture. As described in the previous chapter, the architecture was designed to support 3 different Wishbone commands: RAM read, RAM write and Register Configuration.

## 4.1 SPI Master Test Bench

Figure - SPI Master Test Bench

The SPI Master Test Bench was written using SystemVerilog. The Environment is consumed of the following blocks:

1. Generator (1) generates a random packet, and drives it to the DUT from the FIFO interface.
2. Receiver (2) receives data, which was transmitted from the SPI Slave (Generator and Driver number 3).
3. The Generator (3) generates incremental data, and drives it to the DUT through the SPI via the *spi\_miso* signal.
4. The Receiver (4) receives data, which was transmitted through the SPI Master DUT via the *spi\_mosi* signal.
5. The Scoreboard (5) compares transmissions, and reports when error is detected. The compared data are:
   1. Transmitted data from (1) with the received from (4).
   2. Transmitted data from (3) with the received from (2).
6. This CFG\_DUT is method activated from the test environment, which is in charge of configuring the SPI Master DUT.
7. This is the SPI Master DUT (VHDL design).

In each test, randomized packets are generated and sent to the DUT by Driver (1). Each packet is consumed of:

* Random SPI Slave address.
* Random burst length.
* Random data to be transmitted.

Each packet simulates one continues SPI transaction. The burst length indicates the number of data words that will be exchanged between the SPI Master and Slave during that transaction. When a packet was fully transmitted into the DUT, the Driver will simulate FIFO empty state, before sending the next packet.

### 4.1.2 Tests List

|  | Test Name | Description |
| --- | --- | --- |
| 1 | **Normal Burst** | Checks the basic DUT's functionality, with CPOL = CPHA = 0, and SPI\_CLK = SYS\_CLK/4 |
| 2 | **CPOL-CPHA** | Checks the DUT's functionality with all four CPOL and CPHA configurations |
| 3 | **Clock Frequency** | Checks the DUT's functionality with all available SPI\_CLK frequencies, and with all available CPOL and CPHA configurations |
| 4 | **Reset** | Validates the ability of the DUT to recover from reset, and that other input signals don't affect the DUT during reset |
| 5 | **Forbidden Configuration** | Validates that if someone tries to write into the DUT's configuration registers during active transaction, the values won't be written, and the DUT will raise the *reg\_err* signal |
| 6 | **FIFO Error** | Checks that if the FIFO doesn't raise the *fifo\_din\_valid* signal on time, the DUT will issue an error, and the DUT's FSM will return to idle state. |
| **7** | **Illegal Clock Frequencies** | Checks that if someone tries to write invalid values into the system clock divide register, the values won't be written and the DUT will raise the *reg\_err* signal. |

Table - SPI Master Tests List

Need to add the run command line

### 4.1.3 Coverage Goals

The test bench had the following coverage goals:

* A packet was sent to every possible slave.
* The sent packets had every possible length.
* The packets contained every possible data word.

Need to add coverage report

### 4.1.4 Found Bugs List

|  | Description |
| --- | --- |
| 1 | When CPHA = '1' and Clock Divide Factor = 3, there was one missing bit in the transmission |
| 2 | The last data word of the burst, in which the FIFO becomes empty, was not transmitted |
| 3 | Some configurations caused SPI\_CLK to strobe, although it shouldn't, due to internal register error |

Table - SPI Master Found Bugs List

## 4.2 SPI Slave Verification

## 4.3 SPI Top Verification

## 4.4 Architecture Top Verification

# 5. Summary

# 6. Bibliography

# 7. Appendix A – Block-Level Design

## 7.1 Wishbone Slave

## 7.2 Message Pack Encoder

Message Pack Decoder Encoder transmits data from the Type and Address registers, and from the RAM, in a Message Pack format:

1. **SOF** – Start of Frame
2. **Type** – Message type
3. **Address** – Address for the data, or for other purposes
4. **Length** – Data length. Data length CANNOT be less than 1.
5. **Data (Payload)** – The wrapped data
6. **CRC** – CRC of Type, Address, Length and Data blocks
7. **EOF** – End of Frame

### 7.2.1 Input Registers

Input registers of Type, Address and Length are valid, together with *reg\_ready* signal assertion.

### 7.2.2 FIFO Handshake

MP Encoder transmits all the blocks data to the FIFO. From the FIFO, message will be transmitted by some data transmitter – SPI in this project.

In case FIFO is full – MP Encoder will pause the transmission, and will resume the transmission as soon as the FIFO will not be full.

When the FIFO is not FULL, MP Encoder can transmit data into the FIFO, together with the *enc\_dout\_val* flag.

Figure 14 – Message Pack Encoder <--> FIFO Handshake

### 7.2.3 CRC Handshake

MP Encoder transmits Type, Address, Length and Payload data to the CRC block.

### 7.2.4 RAM Handshake

MP Encoder reads the payload data, to be transmitted, from the RAM. Data is read, together with the RAM address and read enable signal. Output data from RAM is companioned with valid signal.

### 7.2.5 Message Pack Encoder Done

*mp\_enc\_done* flag will be asserted as all data has been transmitted to the FIFO.

## 7.3 Message Pack Decoder

Message Pack Decoder receives a message pack, built from the following blocks:

1. **SOF** – Start of Frame
2. **Type** – Message type
3. **Address** – Address for the data, or for other purposes
4. **Length** – Data length. Data length CANNOT be less than 1.
5. **Data** **(Payload)** – The wrapped data
6. **CRC** – CRC of Type, Address, Length and Data blocks
7. **EOF** – End of Frame

The MP sniffs the data line, until is receives SOF block. Then it decodes the message: Type, Address and Length will be stored into registers, and will be valid when the EOF is received. Data will be stored into RAM.

Special problematic SOF words are being handled:

Suppose SOF = 0xAABBCC.

A message of 0xAABBAABBCC... will be decode correctly by the MP decoder.

### 7.3.1 Data from Wishbone Slave

MP Decoder receives parallel data from *din* port, together with the *valid* signal.

When SOF (64hex in this project) is received – the message decoding process is initialized.

### 7.3.2 RAM Handshake

MP Decoder transmits the received payload into the RAM. Data is being transmitted, together with the RAM address and valid signal

### 7.3.3 CRC Handshake

MP Decoder transmits the received Type, Address, Length and Payload data to the CRC block. In this project, Checksum was used instead.

### 7.3.4 Output Registers

When correct EOF is received, *mp\_dec\_done* flag will be asserted. Type, Address and Length will be valid from that point until Reset or next *mp\_dec\_done* assertion.

### 7.3.5 Error Flags

There are two error output flags:

1. **CRC Error** – will be raised in case received CRC and calculated CRC are not equal.
2. **EOF Error** – will be raised in case received EOF and defined EOF (by generic parameter) are not equal. In case such error has occurred – *mp\_dec\_done* flag will not be raised.

### 7.4.6 Message Pack Decoder Done

When correct EOF is received, *mp\_dec\_done* flag will be raised. In case of EOF error – this flag will not be received.

## 7.4 Checksum

### 7.3.1 Checksum Handshake

Two operation modes are supported:

**Option 1** – *req\_checksum* rise at the same clock with the *data\_valid* assertion:

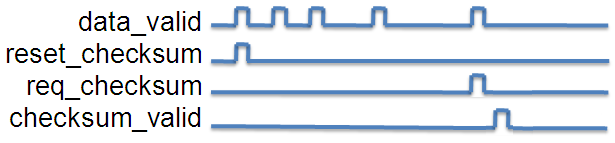


Figure 15 – Checksum Handshake (1)

**Option 2** – *req\_checksum* rise one clock (or more) after the *data\_valid* assertion:

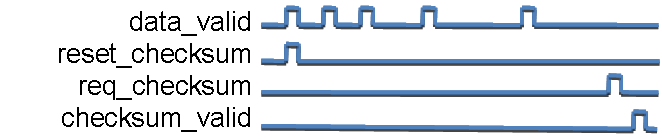


Figure 16 – Checksum Handshake (2)

**Reset Checksum –** Reset the Checksum to the default value, defined by the generic parameter *Checksum\_init\_val\_g*. First *data\_valid* signal can be activated together with the *reset\_Checksum* signal or one (or more) clock after the *reset\_Checksum* signal activation,

**Request for Checksum** – Data will be valid (*Checksum\_valid* activation) one clock after activation of *req\_Checksum*.

### 7.3.2 Signed / Unsigned Checksum

Checksum can be calculated both in signed / unsigned modes. Note that in case of signed Checksum, the MSB bit is reserved as the sign bit, which means that the input and output data width will be sized *(data\_width\_g - 1*).

### 7.3.3 Output Checksum Width

Output Checksum width, set by the generic parameter *Checksum\_out\_width\_g*, may be greater or equal to input Checksum width, set by the generic parameter *data\_width\_g*.

## 7. 5 RAM Controller