SysWip

Alternative Verification

SPI Serial Peripheral Interface Verification IP User Manual

Release 1.0

Table of Contents

1. Introduction

The Serial Peripheral Interface or SPI bus is a synchronous serial data link standard named by Motorola that operates in full duplex mode. Devices communicate in master/slave mode where the master device initiates the data frame. Multiple slave devices are allowed with individual slave select signals.

The SPI Verification IP described in this document is a solution for verification of SPI master and slave devices. The provided SPI verification package includes master and slave verification IPs and examples. It will help engineers to quickly create verification environment end test their SPI master and slave devices.

Package Hierarchy

After downloading and unpacking package you will have the following folder hierarchy:

- spi_vip
  - docs
  - examples
    - sim
    - testbench
  - verification_ip
    - master
    - slave

The Verification IP is located in the *verification_ip* folder. Just copy the content of this folder to somewhere in your verification environment.

Features

- Easy integration and usage
- Supports SPI bus specification as defined in M68HC11 user manual rev 5.0
- Operates as a Master or Slave
- Supports multiple slaves
- Supports clock polarity selections
- Supports CPHA selection
- Supports both MSB and LSB data transmissions
- Fully configurable and accurate bus timing
- Supports single and burst transfers
- Supports different word sizes
- Supports wait states injection

2. SPI Master

The SPI Master Verification IP (VIP) initiates transfers on the SPI bus. It should be the only master on the bus. Write collision error detection isn't supported.

Commands

Commands marked as a *queued* will be put in the one queue and executed sequentially.

Commands marked as a *blocking* will block other commands execution until finishing its own process.

- **Configuration Commands**
  - **setTiming(): -** *non queued, non blocking*
  - **setConfig(): -** *non queued, non blocking*
  - **setMode()***: - non queued, blocking*
- **Data Transfer Commands**
  - **readWriteData(): -** *queued, blocking*
  - **busIdle(): -** *queued, non blocking*
- **Other Commands**
  - **startEnv(): -** *non queued, non blocking*

Commands Description

All commands are *SPI_m_env* class methods and will be accessed only via corresponding objects.

- **setTiming()**
  - **Syntax**
    - *setTiming(t_clk, t_ss_h2l, t_ss_l2h, t_ss_high)*
  - **Arguments**
    - *t_clk:* An *int* variable which specifies SPI clock period
    - *t_ss_h2l:* An *int* variable which specifies the minimum time before the first SPI clock edge after *SS* negedge
    - *t_ss_l2h:* An *int* variable which specifies the minimum time after the last SPI clock edge until *SS* posedge
    - *t_ss_high:* An *int* variable which specifies the minimum idle time between transfers (minimum SS high time)
  - **Description**
    - Sets SPI bus custom timings. For more information about SPI timings see bus specification.

- **setConfig()**
  - **Syntax**
    - *setConfig(msb, burstSize)*
  - **Arguments**
    - *msb:* An *int* variable which specifies MSB or LSB data transmissions
    - *burstSize:* An *int* variable which specifies data burst size
  - **Description**
    - Sets MSB or LSB data transmission mode. For MSB transmission set *msb* argument to 1. For LSB set it 0. Sets data burst size. The amount of bytes which will be transmitted in one burst.
- **setMode()**
  - **Syntax**
    - *setMode(mode)*
  - **Arguments**
    - *mode:* An *int* variable which specifies SPI bus mode
  - **Description**
    - Sets SPI bus mode (Polarity and Phase)
- **readWriteData()**
  - **Syntax**
    - *readWriteData(slaveNum, dataInBuff, dataOutBuff , dataLength)*
  - **Arguments**
    - *slaveNum:* An *int* variable that specifies SPI slave device number
    - *dataInBuff:* 8 bit vector queue that contains data buffer which should be transferred
    - *dataOutBuff:* 8 bit vector queue that contains read data buffer
    - *dataLength:* An *int* variable which specifies the amount of bytes which should be read
  - **Description**
    - Generates SPI Read/Write transactions. Writes complete data buffer to the specified slave. Reads data from slave and returns them via dataOutBuff buffer. If *dataLength* is not specified the transaction length will be equal to *dataInBuff* buffer size.
- **busIdle()**
  - **Syntax**

        - *busIdle(idleTime)*
    - **Arguments**
        - *idleTime:* A *time* variable which specifies wait time
    - **Description**
        - Holds the bus in the idle state for the specified time
- **startEnv()**
    - **Syntax**
        - *startEnv()*
    - **Description**
        - Starts SPI master environment. Don't use data transfer commands before the environment start.

Integration and Usage

The SPI Master Verification IP integration into your environment is very easy. Instantiate the *spi_m_if* interface in you testbench file and connect interface ports to your DUT. Then during compilation don't forget to compile *spi_m.sv* and *spi_m_if.sv* files located inside the *spi_vip/verification_ip/master* folder.

For usage the following steps should be done:

1. Import *SPI_M* package into your test.
    - **Syntax***: import SPI_M::*;*
2. Create *SPI_m_env* class object
    - **Syntax***: SPI_m_env spi = new(spi_ifc_m);*
    - **Description:** *spi_ifc_m* is the reference to the SPI Master Interface instance name.
3. Start SPI Master Environment.
    - **Syntax:** *spi.startEnv();*

This is all you need for SPI master verification IP integration.

3. SPI Slave

The SPI Slave Verification IP models SPI slave device. It has an internal buffers which is accessible by master devices as well as by corresponding functions. The data written by master is stored in the output buffer. The output buffer can be read by corresponding function at any time. In the input buffer is stored data which should be read by master. The input buffer can be filled by corresponding function.

Commands

Commands marked as a *queued* will be put in the one queue and executed sequentially.

Commands marked as a *blocking* will block other commands execution until finishing its own process.

- **Configuration Commands**
  - **setConfig(): -** *non queued, non blocking*
  - **setPollTimeOut():-** *non queued, non blocking*
  - **setMode():** - *non queued, non blocking*
- **Data Processing Commands**
  - **putData(): -** *non queued, non blocking*
  - **getData(): -** *non queued, non blocking*
- **Other Commands**
  - **startEnv(): -** *non blocking, should be called only once for current object*

Commands Description

All commands are *SPI_s_env* class methods and will be accessed only via class objects.

- **setPollTimeOut()**
  - **Syntax**
    - *setPollTimeOut(pollTimeOut)*
  - **Arguments**
    - *pollTimeOut:* A *time* variable that specifies poll time out
  - **Description**
    - Sets the maximum poll time after which poll task will be stopped and poll time out error message generated.
- **setConfig()**
  - **Syntax**
    - *setConfig(msb, burstSize)*
  - **Arguments**
    - *msb:* An *int* variable which specifies MSB or LSB data transmissions
    - *burstSize:* An *int* variable which specifies data burst size
  - **Description**
    - Sets MSB or LSB data transmission mode. For MSB transmission set *msb* argument to 1. For LSB set it 0. Sets data burst size. The amount of bytes which will be transmitted in one burst.
- **setMode()**
  - **Syntax**

        - *setMode(mode)*
    - **Arguments**
        - *mode:* An *int* variable which specifies SPI bus mode
    - **Description**
        - Sets SPI bus mode (Polarity and Phase)
- **putData()**
    - **Syntax**
        - *putData(dataInBuff)*
    - **Arguments**
        - *dataInBuff:* 8 bit vector queue that contains data buffer which will be written to the output buffer.
    - **Description**
        - Writes data to the output buffer
- **getData()**
    - **Syntax**
        - *getData(dataOutBuff, lenght)*
    - **Arguments**
        - *dataOutBuff:* 8 bit vector queue that contains read data from input buffer.
        - *length:* An *int* variable that specifies the amount of data (in bytes) which will be read from the buffer.
    - **Description**
        - Reads *length* bytes data from the input buffer.
- **startEnv()**
    - **Syntax**
        - *startEnv()*
    - **Description**
        - Starts SPI slave environment.

Integration and Usage

The SPI Slave Verification IP integration into your environment is very easy. Instantiate the *spi_s_if* interface in you testbench file and connect interface ports to your DUT. Then during compilation don't forget to compile *spi_s.sv* and *spi_s_if.sv* files located inside the *spi_vip/verification_ip/slave* folder.

For usage the following steps should be done:

1. Import *SPI_S* package into your test.
    - **Syntax***: import SPI_S::*;*

2. Create *SPI_s_env* class object
    - **Syntax:** *SPI_s_env spi = new(spi_ifc_s);*
    - **Description:** *spi_ifc_s* is the reference to the SPI Slave interface instance name.
3. Start SPI Slave Environment
    - **Syntax:** *spi.startEnv();*

Now SPI slave verification IP is ready to respond transactions initiated by master device. Use data processing commands to put or get data from the internal buffers.

4. Important Tips

In this section some important tips will be described to help you to avoid VIP wrong behavior.

Master Tips

1. Call *startEnv()* task as soon as you create *SPI_m_env* object before any other commands. You should call it not more then once for current object.
2. Before using Data Transfer Commands be sure that external hardware reset is done. As current release does not support external reset detection feature, the best way is to wait before DUT reset is done.

Slave Tips

1. Call *startEnv()* task as soon as you create *SPI_s_env* object before any other commands. It should be called before the first valid transaction initiated by SPI master. Internal initialization delay also should be considered. This function should be called only once.
2. Set *burst* size equal to the transfer size initiated by master.
3. Set SPI mode and configuration only when *SS* signal is high.