SysWip

Alternative Verification

I2C Serial Bus Verification IP User Manual

Release 1.0

Table of Contents

1. Introduction

The I2C Verification IP described in this document is a solution for verification of I2C master and slave devices. The provided I2C verification package includes master and slave verification IPs and examples. It will help engineers to quickly create verification environment end test their I2C master and slave devices.

Package Hierarchy

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

- i2c_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 I2C bus specification Rev. 03 - 19 June 2007
- Supports standard, fast, and fast plus speed modes
- Operates as a Master or Slave
- Supports multiple slaves
- Supports 7 and 10 bit addressing
- Supports Software Reset
- Fully configurable and accurate bus timing
- Random delay insertion
- Detects not acknowledg errors
- Supports wait states injection

Limitations

- Does not support Multi-master

- Does not support Clock stretching
- Does not support General Call address

2. I2C Master

The I2C Master Verification IP(VIP) initiates transfers on the I2C bus. It should be the only master on the bus. Multi-master mode isn't supported. Before activating any other I2C master device on the bus the current one should be disabled.

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**
  - **setSpeedMode(): -** *non queued, non blocking*
  - **setRndDelay(): -** *non queued, non blocking*
  - **setTiming(): -** *non queued, non blocking*
  - **setErrMsg(): -** *non queued, non blocking*
  - **setAddrMode(): -** *non queued, non blocking*
- **Data Transfer Commands**
  - **writeData(): -** *queued,*
  - **readData(): -** *queued, blocking*
  - **busIdle(): -** *queued, non blocking*
  - **softReset(): -** *queued, non blocking*
  - **waitCommandDone():** *- queued, blocking*
- **Other Commands**
  - **startEnv(): -** *non queued, non blocking*
  - **printStatus(): -** *non queued, non blocking*

Commands Description

All commands are *I2C_m_env* class methods and will be accessed only via class objects.

- **setSpeedMode()**
  - **Syntax**
    - *setSpeedMode(busSpeedMode)*
  - **Arguments**

    - *busSpeedMode*: A *string* variable that specifies I2C bus speed mode. Can only have the following values: *“Standart”, “Fast”, “FastPlus”*
  - **Description**
    - Sets the I2C bus speed mode.
- **setRndDelay()**
  - **Syntax**
    - *setRndDelay(t_max, rndLevel)*
  - **Arguments**
    - *t_max*: An *int* variable that specifies maximum delay
    - *rndLevel:* An *int* variable that specifies randomization level
  - **Description**
    - Sets randomization level and enables I2C bus random timings.
- **setTiming()**
  - **Syntax**
    - *setTiming(t_Low, t_High, t_HoldStart, t_SetupStart, t_HoldData, t_SetupData, t_Buff, t_Valid, t_SetupStop)*
  - **Arguments**
    - *t_Low, t_High:* Low and High periods of the SCL clock
    - *t_HoldStart, t_SetupStart:* Hold and setup time for start condition
    - *t_HoldData, t_SetupData:* Data hold and setup time
    - *t_Buff:* Bus free time between stop and start conditions
    - *t_Valid:* The maximum data valid time from SCL low
    - *t_SetupStop:* Set up time for stop condition
  - **Description**
    - Sets I2C bus custom timings. For more information about I2C timings see I2C bus specification.
- **setErrMsg()**
  - **Syntax**
    - *setErrMsg(errMsgStr)*
  - **Arguments**
    - *errMsgStr:* A *string* variable that specifies the error message
  - **Description**
    - The specified string will be displayed when not acknowledge error is detected
- **setAddrMode()**

  - **Syntax**
    - *setAddrMode(addrMode)*
  - **Arguments**
    - *addrMode:* An *int* variable that specifies the address mode.
  - **Description**
    - Sets I2C bus address mode. 0 specifies 7 bit and 1 specifies 10 bit modes.
- **writeData()**
  - **Syntax**
    - *writeData(address, dataInBuff, stop)*
  - **Arguments**
    - *address:* An *int* variable that specifies I2C slave device address
    - *dataInBuff:* 8 bit vector queue that contains data buffer which should be transferred
    - *stop:* An *int* variable that specifies if stop condition will be generated after writing the last byte
  - **Description**
    - Writes complete data buffer on specified slave. If stop is enabled generate stop condition. If stop is not enabled, the repeat start will be generated during the next read or write.
- **readData()**
  - **Syntax**
    - *readData(address, dataOutBuff, lenght, stop)*
  - **Arguments**
    - *address:* An *int* variable that specifies I2C slave device address
    - *dataOutBuff:* 8 bit vector queue that contains read data buffer
    - *length:* An *int* variable which specifies the amount of bytes which should be read
    - *stop:* An *int* variable that specifies if stop condition will be generated after reading the last byte
  - **Description**
    - Reads specified amount of data and returns them via *dataOutBuff* buffer. If stop is enabled, generate stop condition. If stop is not enabled the repeat start will be generated during next read or write.
- **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
- **softReset()**
    - **Syntax**
        - *softReset()*
    - **Arguments**
        - *non*
    - **Description**
        - Sends soft reset instruction to the I2C slave devices on the bus
- **waitCommandDone()**
    - **Syntax**
        - *waitCommandDone()*
    - **Description**
        - Waits until all commands in the command buffer are finished
- **startEnv()**
    - **Syntax**
        - *startEnv()*
    - **Description**
        - Starts I2C master environment. Don't use data transfer commands before the environment start.
- **printStatus()**
    - **Syntax**
        - *printStatus()*
    - **Arguments**
        - *non, returns error numbers*
    - **Description**
        - Displays all errors occurred during simulation. This function returns the number of all errors in the buffer. Please note that this function clears failed transactions' buffer.

Integration and Usage

The I2C Master Verification IP integration into your environment is very easy. Instantiate the *i2c_m_if* interface in you testbench top file and connect interface ports to your DUT. Then during compilation don't forget to compile *i2c_m.sv* and *i2c_m_if.sv* files located inside the *i2c_vip/verification_ip/master* folder.

For usage the following steps should be done:

1. Import *I2C_M* package into your test.
   - **Syntax**: *import I2C_M::*;*
2. Create *I2C_m_env* class object
   - **Syntax**: *I2C_m_env i2c = new(i2c_ifc_m, "Standart");*
   - **Description:** *i2c_ifc_m* is the reference to the I2C Master interface instance name. The second argument is I2C bus speed mode.
3. Start I2C Master Environment.
   - **Syntax:** *i2c.startEnv();*

This is all you need for I2C master verification IP integration.

3. I2C Slave

The I2C Slave Verification IP models I2C slave device. It has an internal memory which is accessible by master devices as well as by corresponding functions. After detecting start condition the slave VIP checks I2C address. If the address is correct it will get data from master device and write it to the memory or get data from memory and provide it to the master device.

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**
  - **setSpeedMode(): -** *non queued, non blocking*
  - **setAddress(): -** *non queued, non blocking*
  - **setAckMode(): -** *non queued, non blocking*
  - **setPollTimeOut():-** *non queued, non blocking*
- **Data Processing Commands**
  - **putWord(): -** *non queued, non blocking*
  - **putData(): -** *non queued, non blocking*
  - **getWord(): -** *non queued, non blocking*
  - **getData(): -** *non queued, non blocking*

- **pollWord(): -** *non queued, non blocking*
- **Other Commands**
  - **startEnv(): -** *non blocking, should be called only once for current object*
  - **printStatus(): -** *non blocking*

Commands Description

All commands are *I2C_s_env* class methods and will be accessed only via class objects.

- **setSpeedMode()**
  - **Syntax**
    - *setSpeedMode(busSpeedMode)*
  - **Arguments**
    - *busSpeedMode*: A *string* variable that specifies I2C bus speed mode. Can only have the following values: *"Standart", "Fast", "FastPlus"*
  - **Description**
    - Sets the I2C bus speed mode.
- **setAckMode()**
  - **Syntax**
    - *setAckMode(ackMode)*
  - **Arguments**
    - *ackMode*: An *int* variable that enables or disables not acknowledge generation during memory overflow.
  - **Description**
    - Enables or disables not acknowledge generation during memory overflow. If it is enabled not acknowledge will be generated and memory will not be overwritten.
- **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.
- **putWord()**

- **Syntax**
  - *putWord(address, dataWord)*
- **Arguments**
  - *address:* 8 bit vector that specifies internal memory address.
  - *dataWord:* 8 bit vector that specifies write data word.
- **Description**
  - Writes one byte data word to the internal memory on specified address.

- **putData()**
  - **Syntax**
    - *putData(address, dataInBuff)*
  - **Arguments**
    - *address:* 8 bit vector that specifies internal memory start address.
    - *dataInBuff:* 8 bit vector queue that contains data buffer which will be written to the internal memory.
  - **Description**
    - Writes data buffer to the internal memory starting from specified *address*.
- **getWord()**
  - **Syntax**
    - *dataWord =getWord(address)*
  - **Arguments**
    - *address:* 8 bit vector that specifies internal memory address.
    - *dataWord:* 8 bit vector that contains read data word.
  - **Description**
    - Reads one data byte from the internal memory on the specified address
- **getData()**
  - **Syntax**
    - *getData(startAddr, dataOutBuff, lenght)*
  - **Arguments**
    - *startAddr:* 8 bit vector that specifies internal memory address.
    - *dataOutBuff:* 8 bit vector queue that contains read data buffer.
    - *length:* An *int* variable that specifies the amount of data (in bytes) which will be read from the internal memory.
  - **Description**
    - Reads *length* bytes data from the internal memory starting from specified

address and put it into *dataOutBuff* output buffer.

- **pollWord()**
    - **Syntax**
        - *pollWord(memAddr, expWord)*
    - **Arguments**
        - *memAddr:* 8 bit vector that specifies internal memory address.
        - *expWord:* 8 bit vector that specifies expected data byte.
    - **Description**
        - Polls internal memory address until the data word is equal to the expected data. If time out occurs generates error message and returns.
- **startEnv()**
    - **Syntax**
        - *startEnv()*
    - **Description**
        - Starts I2C slave environment.
- **printStatus()**
    - **Syntax**
        - *errCnt = printStatus()*
    - **Description**
        - Displays all time out and memory overflow errors occurred during simulation time. This function returns the number of all errors.

Integration and Usage

The I2C Slave Verification IP integration into your environment is very easy. Instantiate the *i2c_s_if* interface in you testbench top file and connect interface ports to your DUT. Then during compilation don't forget to compile *i2c_s.sv* and *i2c_s_if.sv* files located inside the *i2c_vip/verification_ip/slave* folder.

For usage the following steps should be done:

1. Import *I2C_S* package into your test.
    - **Syntax**: *import I2C_S::*;*
2. Create *I2C_s_env* class object
    - **Syntax:** *I2C_s_env i2c = new(i2c_ifc_s, "Standart", addr);*
    - **Description:** *i2c_ifc_s* is the reference to the I2C Slave interface instance name. The second argument is the bus speed mode. The third argument is I2C slave device address.
3. Start I2C Slave Environment
    - **Syntax:** *i2c.startEnv();*

Now I2C slave verification IP is ready to respond transactions initiated by master device. Use data processing commands to put or get data from internal memory.

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 *I2C_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.
3. Call *printStatus()* at the end of the test. As this function will empty error buffer you will lose error information if you call it in the middle of the test. Error messages will be lost in your terminal and it will be difficult to find them after test is done.
4. As all configuration commands are non queued and all Data Transfer Commands are queued, you will have synchronization issues if you want to apply specific configuration to the current transaction. Use *waitCommandDone()* for synchronization.

Slave Tips

1. Call *startEnv()* task as soon as you create *I2C_s_env* object before any other commands. It should be called before the first valid transaction initiated by I2C master. Internal initialization delay also should be considered. This function should be called only once.
2. Call *printStatus()* at the end of the test. As this function will empty error buffer, you will lose the errors information when you call it in the middle of the test. Error messages will be lost in you terminal and will be difficult to find after test is done.
3. The internal memory size is 256 byte. When master writes data to the slave the data is stored in the memory starting from 0x00 address. If data size is more than 256 byte the internal address counter will be rolled over and memory will be overwritten. No error or warning will be generated. Use *setAckMode()* to not acknowledge generation. When address counter is rolled over not acknowledge will be generated and memory will never be overwritten.
4. 10 bit addressing, device ID and software reset are not supported in this release.