SysWip

Alternative Verification

Avalon-MM Interface Verification IP User Manual

Release 1.0

Table of Contents

1. Introduction

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

Package Hierarchy

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

- avalon_mm_vip
  - docs
  - examples
    - sim
    - testbench
  - verification_ip

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
- Free SystemVerilog source code
- Compliant to the Avalon Interface specification
- Operates as a Master or Slave
- Supports 1, 2, 4, 8, 16, 32, 64 and 128 data block size
- Supports burst read and write
- Supports misaligned transfers
- Supports wait request and read data available
- Supports wait states injection
- Supports full random timings

Limitations

- Doesn't support *lock, debugaccess* and *begintransfer* signals

2. Avalon Master

The Avalon Master Verification IP (VIP) initiates transfers on the bus.

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**
  - **setRandDelay(): -** *non queued, non blocking*
  - **setTimeOut(): -** *non queued, non blocking*
- **Data Transfer Commands**
  - **writeData(): -** *queued, non blocking*
  - **readData(): -** *queued, non blocking*
  - **getData**(**): -** *queued, blocking*
  - **pollData**(): - *queued, blocking*
  - **busIdle**(): - *queued, non blocking*
- **Other Commands**
  - **startEnv(): -** *non queued, non blocking*
  - **waitCommandDone():-** *queued, blocking*
  - **printStatus():-** *non queued, non blocking*

Commands Description

All commands are *Avalon_m_env* class methods.

- **setRandDelay()**
  - **Syntax**
    - *setRandDelay(minBurst, maxBurst, minWait, maxWait)*
  - **Arguments**
    - *minBurst:* An *int* variable which specifies minimum value for burst length
    - *maxBurst:* An *int* variable which specifies maximum value for burst length
    - *minWait:* An *int* variable which specifies the minimum value for wait cycles
    - *maxWait:* An *int* variable which specifies the maximum value for wait cycles
  - **Description**
    - Enables/Disables bus random timings. To disable random timing all arguments should be set to zero.
- **setTimeOut()**
  - **Syntax**
    - *setTimeOut(waitReqTimeOut, validTimeOut)*
  - **Arguments**
    - *waitReqTimeOut:* An *int* variable which specifies the maximum wait clock

cycles for slave wait request signal.

- *validTimeOut:* An *int* variable which specifies maximum wait clock cycles for the slave read data available signal.

- **Description**
  - Sets the maximum clock cycles for slave slave response. If the slave delays response the error message will be generated.

- **writeData()**
  - **Syntax**
    - *writeData(addr, inBuff, burstEn)*
  - **Arguments**
    - *addr:* An *int* variable that specifies the start byte address. The misaligned addresses will be aligned
    - *inBuff:* 8 bit vector array that contains data buffer which should be transferred
    - *burstEn:* An *int* variable that enables burst mode
  - **Description**
    - Writes data buffer. The output addresses are always aligned. The *byteenable* signales indicate the valid bytes on the data bus.

- **readData()**
  - **Syntax**
    - *readData(addr, dataLength, burstEn)*
  - **Arguments**
    - *addr:* An *int* variable that specifies the start byte address The misaligned addresses will be aligned
    - *dataLength:* An *int* variable that specifies the amount of bytes which should be read.
    - *burstEn:* An *int* variable that enables burst mode
  - **Description**
    - Generate read transactions. The read data will be stored in the internal buffer and then can be read by *getData()* command.

- **getData()**
  - **Syntax**
    - *getData(dataLength, outBuff)*
  - **Arguments**
    - *dataLength:* An *int* variable that specifies the amount of bytes which should be got from the internal buffer.

        - *outBuff:* 8 bit vector array that contains read data buffer
    - **Description**
        - Returns read data from the internal buffer. If the *dataLength* is more than data in the internal buffer the error message will be generated.

- **pollData()**
    - **Syntax**
        - *pollData(addr, pollData, pollTimeOut)*
    - **Arguments**
        - *addr:* An *int* variable that specifies the start address
        - *pollData:* 8 bit vector array that contains data buffer which should be compared with read data buffer
        - *pollTimeOut:* An *int* variable that specifies the poll time out.
    - **Description**
        - Read data buffer starting from *addr* and compare it with *pollData* buffer. Repeat until buffers are equal or until time out occurred.
- **busIdle()**
    - **Syntax**
        - *busIdle(idleCycles)*
    - **Arguments**
        - *idleCycles:* A *int* variable which specifies wait clock cycles
    - **Description**
        - Holds the bus in the idle state for the specified clock cycles
- **waitCommandDone()**
    - **Syntax**
        - *waitCommandDone()*
    - **Description**
        - Wait until all transactions in the buffer are finished
- **printStatus()**
    - **Syntax**
        - *printStatus()*
    - **Description**
        - Prints all errors occurred during simulation time and returns error count.
- **startEnv()**

- **Syntax**
  - *startEnv()*
- **Description**
  - Starts the Avalon master environment. Don't use data transfer commands before the environment start.

Integration and Usage

The Avalon Master Verification IP integration into your environment is very easy. Instantiate the *avalon_m_if* interface in you testbench and connect interface ports to your DUT. Then during compilation don't forget to compile *avalon_m.sv* and *avalon_m_if.sv* files located inside the *avalon_mm_vip/verification_ip* folder.

For usage the following steps should be done:

1. Import *AVALON_M* package into your test.
   - **Syntax***: import AVALON_M::*;*
2. Create *Avalon_m_env* class object
   - **Syntax***: Avalon_m_env avalon= new(id_name, avalon_ifc_m, dataSize);*
   - **Description:** *id_name* is the name of the master vip(*string* variable), *avalon_ifc_m* is the reference to the Avalon Master Interface instance name. *dataSize* is data block size in bytes. Use only 1, 2, 4, 8, 16, 32, 64 and 128.
3. Start Avalon Master Environment.
   - **Syntax:** *avalon.startEnv();*

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

3. Avalon Slave

The Avalon Slave Verification IP models Avalon slave device. It has an internal memory which is accessible by the master device as well as by corresponding commands.

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**
  - **setRandDelay(): -** *non queued, non blocking*
- **Data Processing Commands**
  - **putData(): -** *non queued, non blocking*
  - **getData(): -** *non queued, non blocking*

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

Commands Description

All commands are *Avalon_s_env* class methods.

- **setRandDelay()**
    - **Syntax**
        - *setRandDelay(minWaitReqDelay, maxWaitReqDelay, minValidDelay, maxValidDelay=0)*
    - **Arguments**
        - minWaitReqDelay: A *int* variable that specifies minimum value for the wait request delay
        - *maxWaitReqDelay:* A *int* variable that specifies maximum value for the wait request delay
        - *minValidDelay:* A *int* variable that specifies minimum value for the read data available delay
        - *maxValidDelay:* A *int* variable that specifies maximum value for the read data available delay
    - **Description**
        - Enables/Disables slave response (wait request and read data available) random delays. To disable set all arguments to zero.
- **putData()**
    - **Syntax**
        - *putData(startAddr, dataInBuff*)
    - **Arguments**
        - *startAddr:* An *int* variable that specifies the start byte address. Can be misaligned
        - *dataInBuff:* 8 bit vector array that contains data buffer which will be written to the memory.
    - **Description**
        - Puts the data buffer to the the internal memory.
- **getData()**
    - **Syntax**
        - *getData(startAddr, dataOutBuff, lenght)*

- **Arguments**
  - *startAddr:* An *int* variable that specifies the start byte address. Can be misaligned
  - *dataOutBuff:* 8 bit vector array that contains the read data from the memory.
  - *length:* An *int* variable that specifies the amount of bytes which will be read from the internal memory.
- **Description**
  - Reads the data from the internal memory.

- **pollData()**
  - **Syntax**
    - *pollData(address, pollData, pollTimeOut)*
  - **Arguments**
    - *address:* An *int* variable that specifies the start address. Can be misaligned
    - *pollData:* 8 bit vector array that contains data buffer which should be compared with read data buffer
    - *pollTimeOut:* An *int* variable that specifies the poll time out clock cycles.
  - **Description**
    - Read data buffer from internal memory and compare it with *pollData* buffer. Repeat until buffers are equal or until time out occurred.

- **startEnv()**
  - **Syntax**
    - *startEnv()*
  - **Description**
    - Starts Avalon slave environment.
- **printStatus()**
  - **Syntax**
    - *printStatus()*
  - **Description**
    - Prints all errors occurred during simulation time and returns error count.

Integration and Usage

The Avalon Slave Verification IP integration into your environment is very easy. Instantiate the *avalon_s_if* interface in you testbench and connect interface ports to your DUT. Then during compilation don't forget to compile *avalon_s.sv* and *avalon_s_if.sv* files located inside the *avalon_mm_vip/verification_ip* folder.

For usage the following steps should be done:

1. Import *AVALON_S* package into your test.

- **Syntax***: import AVALON_S::*;*

2. Create *Avalon_s_env* class object
    - **Syntax:** ***Avalon****_s_env avalon = new(id_name , avalon_ifc_s, dataSize);*
    - **Description:** *id_name* is the name of the slave vip(*string* variable), *avalon_ifc_s* is the reference to the Avalon Slave interface instance name. *dataSize* is data block size in bytes. Use only 1, 2, 4, 8, 16, 32, 64 and 128.
3. Start Avalon Slave Environment
    - **Syntax:** *avalon.startEnv();*

Now Avalon slave verification IP is ready to respond transactions initiated by master device. Use data processing commands to put or get data from the 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 *Avalon_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 *Avalon_s_env* object before any other commands. It should be called before the first valid transaction initiated by Avalon master.