SysWip

Alternative Verification

Avalon ST Interface Verification IP User Manual

Release 1.0

Table of Contents

1. Introduction

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

Package Hierarchy

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

- avalonst_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
- Free SystemVerilog source code
- Compliant to Avalon Interface Specifications Ver1.2
- Operates as a Master(Source) or Slave(Sink)
- Supports 1, 2, 4, 8, 16 and 32 bytes data block size
- Configurable endians (little endian or big endian)
- Supports wait states injection
- Supports full random timings
- Supports full random “empty” value generation
- Supports wrong start/end of packet insertion and detection

Limitations

- Doesn't support multiple channels
- Doesn't support error signals
- readyLatency is fixed to zero

2. Avalon-ST Master

The Avalon-ST Master Verification IP (VIP) initiates transfers to the sink 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**
  - **setRndDelay(): -** *queued, non blocking*
  - **setTimeOut(): -** *queued, non blocking*
  - **setEndian(): -** *non queued, non blocking*
  - **setPacketConfig(): -** *non queued, non blocking*
- **Data Transfer Commands**
  - **writeData(): -** *queued, non 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 *AVALONST_m_env* class methods.

- **setRndDelay()**
  - **Syntax**
    - *setRndDelay(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(readyTimeOut)*
  - **Arguments**
    - *readyTimeOut:* An *int* variable which specifies the maximum wait clock cycles for slave *ready* signal.
  - **Description**
    - Sets the maximum clock cycles for slave ready signal. If slave delays ready signal then specified, error message will be generated.
- **setEndian()**
  - **Syntax**
    - *setEndian(dataEndian)*
  - **Arguments**
    - *dataEndian:* An *int* variable which specifies the data block endian.
  - **Description**
    - Sets data block endian. 1 is big endian, 0 is little endian.
- **setPacketConfig()**
  - **Syntax**
    - *setPacketConfig(startofpacketEn, endofpacketEn, emptyRndEn)*
  - **Arguments**
    - *startofpacketEn:* An *int* variable which specifies start of packet mode.
    - *endofpacketEn*: An int variable which specifies end of packet mode.
    - *emptyRndEn*: An int variable which specifies empty signal mode.
  - **Description**
    - Enables/Disables start of packet and end of packet signals. Enables/Disables empty signal random mode. If empty signal random mode enabled, empty signal will be generated randomly after each data block transaction.
- **writeData()**
  - **Syntax**
    - *writeData(inBuff)*
  - **Arguments**
    - *inBuff:* 8 bit vector queue that contains data buffer which should be transferred
  - **Description**

        - Writes data buffer.
- **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**
        - Waits until all commands in the command buffer are finished
- **printStatus()**
    - **Syntax**
        - *printStatus()*
    - **Description**
        - Prints all errors occurred during simulation time and returns error count. If there are no any errors returns zero.
- **startEnv()**
    - **Syntax**
        - *startEnv()*
    - **Description**
        - Starts Avalon-ST master environment. Don't use any other commands before the environment start.

Integration and Usage

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

For usage the following steps should be done:

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

2. Create *AVALONST_m_env* class object
    - **Syntax***: AVALONST_m_env avalon_st= new(avalon_st_ifc_m, dataSize);*
    - **Description:** *avalon_st_ifc_m* is the reference to the AVALON-ST Master Interface instance name. *dataSize* is data block size in bytes. Use only 1, 2, 4, 8, 16 and 32.
3. Start AVALON-ST Master Environment.
    - **Syntax:** *avalon_st.startEnv();*

This is all you need for AVALON-ST master verification IP integration.

3. AVALON-ST Slave

The AVALON-ST Slave(Sink) Verification IP models AVALON-ST slave device. It has an internal buffer which is accessible by master devices 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**
    - **setRndDelay():** - *queued, non blocking*
    - **setTimeOut():**- *non queued, non blocking*
    - **setEndian():** - *non queued, non blocking*
- **Data Processing Commands**
    - **readData():** - *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 *AVALONST_s_env* class methods.

- **setRndDelay()**
    - **Syntax**
        - *setRndDelay(minReadyDelay, maxReadyDelay)*
    - **Arguments**
        - *minReadyDelay:* A *int* variable that specifies minimum value for ready delay
        - *maxReadyDelay:* A *int* variable that specifies maximum value for ready delay

- **Description**
  - Enables/Disables ready random delays. To disable ready delay set all arguments to zero.

- **setTimeOut(***timeOut***)**
  - **Syntax**
    - *setTimeOut(timeOut)*
  - **Arguments**
    - *timeOut:* A *int* variable which specifies read data time out
  - **Description**
    - Sets read data time out.

- **setEndian()**
  - **Syntax**
    - *setEndian(dataEndian)*
  - **Arguments**
    - *dataEndian:* An *int* variable which specifies the data block endian.
  - **Description**
    - Sets data block endian. 1 is big endian, 0 is little endian.

- **readData()**
  - **Syntax**
    - *readData(dataOutBuff)*
  - **Arguments**
    - *dataOutBuff:* 8 bit vector queue that contains read data from buffer.
  - **Description**
    - Reads complete packet from the internal buffer.

- **startEnv()**
  - **Syntax**
    - *startEnv()*
  - **Description**
    - Starts AVALON-ST slave environment.

- **printStatus()**
  - **Syntax**
    - *printStatus()*
  - **Description**

- Prints all errors occurred during simulation time and returns error count. If there are no any errors returns zero.

Integration and Usage

The AVALON-ST Slave Verification IP integration into your environment is very easy. Instantiate the *avalon_st_s_if* interface in you testbench and connect interface ports to your DUT. Then during compilation don't forget to compile *avalon_st_s.sv* and *avalon_st_s_if.sv* files located inside the *avalonst_vip/verification_ip/slave* folder.

For usage the following steps should be done:

1. Import *AVALONST_S* package into your test.
    - **Syntax***: import AVALONST_S::*;*
2. Create *AVALONST_s_env* class object
    - **Syntax:** *AVALONST_s_env avalonst;*

      avalonst = new(avalonst_ifc_s, dataSize);
    - **Description:** *avalonst_ifc_s* is the reference to the AVALON-ST Slave interface instance name. *dataSize* is data block size in bytes. Use only 1, 2, 4, 8,16 and 32.
3. Start AVALON-ST Slave Environment
    - **Syntax:** *avalonst.startEnv();*

Now AVALON-ST slave verification IP is ready to respond transactions initiated by master device. Use data processing commands to read data from the internal buffer.

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 *AVALONST_s_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 *AVALONST_s_env* object before any other commands. It should be called before the first valid transaction initiated by master.