SysWip

Alternative Verification

AMBA Peripheral Bus Verification IP User Manual

Release 1.0

Table of Contents

1. Introduction

The APB Verification IP described in this document is compliant to AMBA3.0 Peripheral Bus (APB). The provided APB verification package includes master and slave verification IPs with examples. The examples can be very useful during integration and usage.

Installation

- The APB Verification IP is available at: www.syswip.com from downloads page.
- Download *apb_vip.tar.gz* file and unpack it.
- This Verification IP is written in SystemVerilog and was tested under:
  - Linux using VCS2008.12
  - WindowsXP using QuestaSim6.4
- For future support don't hesitate to contact me: http://syswip.com/wp/contacts

Package Hierarchy

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

- apb_vip
  - docs
  - examples
    - master
      - rtl
      - sim
      - testbench
    - slave
      - 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.

2. APB Master

The APB Master Verification IP(VIP) is generating transactions on the APB bus. It can also be considered as a AHB/APB bridge replacement used in real designs. Please note that it does not have AHB/APB bridge functionality and can only be considered bridge replacement from APB side.

The APB VIP supports only one slave device. For connecting more slaves external decoder is needed.

Features

The APB Master VIP has the following features:

- Configurable data bus width 8, 16 and 32 bits wide
- Misaligned transfers
- Fixed and random transfer delay insertion
- Configurable pready time out
- Slave error detection and configurable error messages
- Complete data packet transaction read/wite
- Bus idle cycle insertion
- Address polling
- Failed transaction buffer

Configuration Commands

All configuration commands listed bellow have zero cycle execution time and executing immediately without waiting in the queue.

- **APB data bus width:** Can be set only during APB object creation and cannot be changed during simulation.
- **setPreadyTimeOut():**
- **setPollTimeOut():**
- **setDelay():**
- **rndDelayEn():**
- **setErrMsgs():**

Data Transfer 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 own process.

- **writeData(): -** *queued,*
- **writeWord(): -** *queued, non blocking*
- **readData(): -** *queued, blocking*
- **readWord(): -** *queued, blocking*
- **pollWord(): -** *queued, blocking*
- **busIdle(): -** *queued, non blocking*

Other Commands

- **startEnv(): -** *non queued, blocking, should be called only once for current object*
- **EnvReset(): -** *non queued, blocking*
- **printFailedTrInfo(): -** *non queued, non blocking*

Commands Description

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

- **setPreadyTimeOut()**
  - **Syntax**
    - *setPreadyTimeOut(timeOutCycles)*
  - **Arguments**
    - *timeOutCycles*: An *int* variable that specify *pready* signal waiting clock cycles.
  - **Description**
    - Sets the maximum number of wait clock cycles for *pready* signal. After activating *psel* VIP waits valid *pready* signal. If *pready* is not valid during specified clock cycles than VIP will generate error message and continue with next transaction.
- **setPollTimeOut()**
  - **Syntax**
    - *setPollTimeOut(timeOutCycles)*
  - **Arguments**
    - *timeOutCycles*: An *int* variable that specify maximum poll clock cycles.
  - **Description**
    - Sets the maximum number of poll clock cycles after which poll task will be stopped and poll time out error message generated. The only task in APB Master VIP where poll time out is used is *pollWord()*.

- **setDelay()**
  - **Syntax**
    - *setDelay(burstLength, waitCycle)*
  - **Arguments**
    - *burstLengt:* An *int* variable that specify the maximum number of transactions when *psel* signal can be continuously high. For disabling delays put this value to 0.
    - *waitCycle:* An *int* variable that specify the number of wait cycles. When *burstLengt is 0 waitCycle* will be ignored.
  - **Description**
    - Sets APB bus timing delay. Which means to set some constraints to *psel* signal. The value specified in the *burstLengt* argument is the maximum number of transactions when *psel* signal can be continuously high. Then *psel* will go to low and wait several clock cycles specified in the *waitCycle* argument. For disabling delay call this function with 0 as a first argument. The second one will be ignored.
- **rndDelayEn()**
  - **Syntax**
    - *rndDelayEn(minBurst, maxBurst, minWaitCycle, maxWaitCycle)*
  - **Arguments**
    - *minBurst:* An *int* variable that specify the minimum value for *burstLengt*
    - *maxBurst:* An *int* variable that specify the maximum value for *burstLengt*
    - *minWaitCycle:* An *int* variable that specify the minimum value for *waitCycle*
    - *maxWaitCycle:* An *int* variable that specify the maximum value for *waitCycle*
  - **Description**
    - Enables APB bus timing random delays. After each transaction new random value will be generated for *burstLengt* and *waitCycle*. The generated random value will be in the range specified in the function arguments. For disabling random delays call setDelay() function with 0 value in the first argument.
- **setErrMsgs()**
  - **Syntax**
    - *setErrMsgs(timeOutMsg, slaveErrorMsg)*
  - **Arguments**
    - *timeOutMsg:* A *string* variable that specify the message which will be displayed during time out error.

- *SlaveErrorMsg:* A *string* variable that specify the message which will be displayed during APB bus slave error detection.

- **Description**
  - Sets *pready* time out and slave error messages. The message will be displayed in you terminal if corresponding error ocured.

- **writeWord()**
  - **Syntax**
    - *writeWord(address, dataWord)*
  - **Arguments**
    - *address:* A 32 bit vector that specifies APB address.
    - *dataWord:* A 32 bit vector that specifies APB write data word.
  - **Description**
    - Writes one data word on specified address. If APB data width is less then 32 bits then only corresponding least bits will be used as a data. This task creates only one write transaction even APB data width is less then 32 bits. So if data bus is 8 bits and this task was called then only *dataWord[7:0]* will be written.

- **writeData()**
  - **Syntax**
    - *writeData(address, dataInBuff, addrIncr)*
  - **Arguments**
    - *address:* A 32 bit vector that specifies APB write start address
    - *dataInBuff:* A 8 bit vector queue that contains data buffer which should be transferred
    - *addrIncr:* An *int* variable that specify will address be incremented or no during complete data buffer transfer. If you don't specify this argument it will be considered as 1.
  - **Description**
    - Writes complete data buffer on specified address. If *addrIncr* is 0 then all data will be written on the same address. It can be useful if you want to write FIFO. If *addrIncr* is 1 or not specified then address will be incremented after each transaction. Incremented value depends on APB data bus width.

- **readWord()**
  - **Syntax**
    - *readWord(address, dataWord)*
  - **Arguments**

- *address:* A 32 bit vector that specifies APB address.
- *dataWord:* A 32 bit vector that specifies APB read data word.

- **Description**
  - Reads one data word at the specified address and returns the data via *dataWord* variable. This task creates only one read transaction even APB data width is less then 32 bits.

- **readData()**
  - **Syntax**
    - *readData(address, dataOutBuff, lenght, addrIncr)*
  - **Arguments**
    - *address:* 32 bit vector that specifies APB read start address
    - *dataOutBuff:* A 8 bit vector queue that contains read data buffer
    - *length:* An *int* variable which specify the amount of bytes should be read
    - *addrIncr:* An *int* variable that specify will address be incremented or no during complete data buffer read. If you don't specify this argument it will be considered as 1.
  - **Description**
    - Reads specified amount of data and returns them via *dataOutBuff* buffer. If *addrIncr* is 0 then all data will be read at the same address. It can be useful if you want to read a FIFO. If *addrIncr* is 1 or not specified then address will be incremented after each transaction. Incremented value depends on APB data bus width. Please not that *length* is not the same as transfer size. Those are the same only when APB data bus width is 8 bits.

- **pollWord()**
  - **Syntax**
    - *pollWord(address, pollData)*
  - **Arguments**
    - *address:* A 32 bit vector that specifies APB address
    - *pollData:* A 32 bit vector that will be compared with read data
  - **Description**
    - Reads data word at the specified address and compares with expected one. Repeat until those are equal. Returns only if those are equal or when time out occurred. During time out displays error message.

- **busIdle()**
  - **Syntax**
    - *busIdle(idleCycles)*

- **Arguments**
  - *idleCycles:* An *int* variable which specify number of clock cycles
- **Description**
  - Holds the bus in the idle state for the specified number of clock cycles.

- **startEnv()**
  - **Syntax**
    - *startEnv()*
  - **Arguments**
    - *non*
  - **Description**
    - Starts APB master environment. Don't use data transfer commands before environment starting.
- **EnvReset()**
  - **Syntax**
    - *EnvReset()*
  - **Arguments**
    - *non*
  - **Description**
    - Reset APB master environment. This command will empty all transaction buffers and set *psel, penable* and *pwrite* signals to low.
- **printFailedTrInfo()**
  - **Syntax**
    - *printFailedTrInfo()*
  - **Arguments**
    - *non, returns number of failed transactions*
  - **Description**
    - Displays all errors (pready, polling timeout, slave error) occurred during full simulation time. This function returns the number of all errors in the buffer. Please not that this function clears failed transactions buffer.

Integration and Usage

The APB Master Verification IP integration into your environment is very easy. Instantiate the *apb_m_if* interface in you testbench top file and connect interface ports to your DUT. Then during compilation don't forget to compile *apb_m.sv* and *apb_m_if.sv* files located inside the

apb_vip/verification_ip/master folder.

For usage the following steps should be done:

1. Import *APB_M* package into your test.
    - **Syntax***: import APB_M::*;*
2. Create *ABP_m_env* class object
    - **Syntax***: ABP_m_env apb = new(apb_ifc, 4);*
    - **Description:** *apb_ifc* is the reference to the APB Master interface instance name. The second argument is APB bus data width in bytes.
3. Start APB Master Environment.
    - **Syntax:** *apb.startEnv();*

This is the all you need for integration. Now you can configure the VIP and start data transfers.

3. APB Slave

The APB Slave Verification IP does not initiate any transfers and only responds to transactions initiated by the APB Master. The APB Slave VIP has a memory element to hold transfer data. It also has FIFO mode support. In FIFO mode all read data words will be popped from output FIFO and all write data words will be pushed to input FIFO.

Features

The APB Slave VIP has the following features:

- Configurable data bus width 8, 16 and 32 bits wide
- Misaligned transfers
- Fixed and random delay insertion for pready signal
- Configurable poll time out
- Slave error insertion for specified addresses
- Input and Output FIFO mode
- Failed transaction buffer

Configuration Commands

All configuration commands listed bellow have zero cycle execution time and executing immediately without waiting in the queue.

- **APB data bus width:** Can be set only during APB object creation and cannot be changed during simulation.
- **setPollTimeOut():**
- **setDelay():**
- **rndDelayEn():**
- **setFifoAddr():**
- **disableFifoMode():**
- **setSlaveErrAddr():**
- **disableSlaveErr():**

Data Processing 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 own process.

- **putWord(): -** *non blocking*
- **putData(): -** *non blocking*

- **getWord(): -** *non blocking*
- **getData(): -** *non blocking*
- **pollWord(): -** *blocking*
- **writeFifoWord(): -** *non blocking*
- **writeFifo(): -** *non blocking*
- **readFifoWord(): -** *blocking*
- **readFifo(): -** *blocking*

Other Commands

- **startEnv(): -** *non blocking, should be called only once for current object*
- **fifoReset(): -** *non blocking*
- **printFailedTrInfo():** - *non blocking*

Commands Description

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

- **setPollTimeOut()**
  - **Syntax**
    - *setPollTimeOut(pollTimeOut)*
  - **Arguments**
    - *pollTimeOut*: An *int* variable that specify maximum poll clock cycles.
  - **Description**
    - Sets the maximum number of poll clock cycles after which poll task will be stopped and poll time out error message generated. If set to zero time out never occurs.
- **setDelay()**
  - **Syntax**
    - *setDelay(preadyDelay)*
  - **Arguments**
    - *preadyDelay:* An *int* variable that specify the *pready* delay cycles after valid *psel* signal.
  - **Description**
    - Sets *pready* timing delay. After valid *psel* signal the *pready* will be delayed specified amount of clock cycles. If set to zero then the *pready* will always be valid.
- **rndDelayEn()**

- **Syntax**
  - *rndDelayEn(minPready, maxPready)*
- **Arguments**
  - *minPready:* An *int* variable that specify the minimum value for *preadyDelay*
  - *maxPready:* An *int* variable that specify the maximum value for *preadyDelay*
- **Description**
  - Enables *pready* timing random delays. After each transaction new random value will be generated for *preadyDelay.* The generated random value will be in the range specified in the function arguments. For disabling random delays call setDelay() function.

- **setFifoAddr()**
  - **Syntax**
    - *setFifoAddr(inFifoAddr, outFifoAddr)*
  - **Arguments**
    - *inFifoAddr:* A32 bit vector that specify the input FIFO address.
    - *outFifoAddr:* A32 bit vector that specify the output FIFO address.
  - **Description**
    - Sets addresses for in/out FIFOs and enables FIFO mode. Data word written by master at *inFifoAddr* address will be pushed to the input FIFO. Data word read by master at *outFifoAddr* address will be popped from the output FIFO.
- **disableFifoMode()**
  - **Syntax**
    - *disableFifoMode()*
  - **Arguments**
    - *non*
  - **Description**
    - Disables FIFO mode. Input and output FIFO addresses will be switched to memory addresses.
- **setSlaveErrAddr()**
  - **Syntax**
    - *setSlaveErrAddr(startAddr, endAddr)*
  - **Arguments**
    - *startAddr:* A32 bit vector that specify the start address for slave error space.
    - *endAddr:* A32 bit vector that specify the end address for slave error space.

- **Description**
  - Sets address space for APB slave error generation. All read or write transactions initiated by master inside this space will generate slave error *pslverr* signal will be set.

- **disableSlaveErr()**
  - **Syntax**
    - *disableSlaveErr()*
  - **Arguments**
    - *non*
  - **Description**
    - Disables APB slave error generation.

- **putWord()**
  - **Syntax**
    - *putWord(address, dataWord)*
  - **Arguments**
    - *address:* A 32 bit vector that specifies internal memory address.
    - *dataWord:* A 32 bit vector that specifies memory write data word.
  - **Description**
    - Writes one data word to the internal memory on specified address. Data word size depends on APB data width. It can be 1, 2 and 4 bytes.

- **putData()**
  - **Syntax**
    - *putData(address, dataInBuff)*
  - **Arguments**
    - *address:* A 32 bit vector that specifies internal memory address.
    - *dataInBuff:* A 8 bit vector queue that contains data buffer which will be written to the internal memory. First byte will be written at *address*.
  - **Description**
    - Writes data buffer to the internal memory starting from specified *address*. Each next byte will be written after address increment.

- **getWord()**
  - **Syntax**
    - *dataWord = getWord(address)*
  - **Arguments**

        - *address:* A 32 bit vector that specifies internal memory address.
        - *dataWord:* A 32 bit vector that contains memory read data word.
    - **Description**
        - Reads one data word from the internal memory on specified address
- **getData()**
    - **Syntax**
        - *getData(startAddr, dataOutBuff, lenght)*
    - **Arguments**
        - *startAddr:* A 32 bit vector that specifies internal memory address.
        - *dataOutBuff:* A 8 bit vector queue that contains output data buffer which is read from the internal memory.
        - *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 them into *dataOutBuff* output buffer.
- **pollWord()**
    - **Syntax**
        - *pollWord(pollAddr, pollData)*
    - **Arguments**
        - *pollAddr:* A 32 bit vector that specifies internal memory address.
        - *pollData:* A 32 bit vector that contains expected value which will be compared with memory data.
    - **Description**
        - Reads internal memory specified address until the read data word is equal to expected *pollData* data. If time out occurred generates error message and returns
- **writeFifoWord()**
    - **Syntax**
        - *writeFifoWord(dataWord)*
    - **Arguments**
        - *dataWord:* A 32 bit vector data word.
    - **Description**
        - Pushes one data word to the output FIFO.

- **writeFifo()**
  - **Syntax**
    - *writeFifo(dataInBuff)*
  - **Arguments**
    - *dataInBuff:* A 8 bit vector queue that contains data buffer which will be written to the output FIFO.
  - **Description**
    - Pushes all data buffer to the output FIFO.
- **readFifoWord()**
  - **Syntax**
    - *readFifoWord(dataWord)*
  - **Arguments**
    - *dataWord:* A 32 bit vector that contains read data word.
  - **Description**
    - Popes one word data from input FIFO. If time out occurred (FIFO is empty for a specified time) then generates error message end returns.
- **readFifo()**
  - **Syntax**
    - *readFifo(dataOutBuff, lenght)*
  - **Arguments**
    - *dataOutBuff:* A 8 bit vector queue that contains read data buffer.
    - *length:* An *int* variable that specifies the read data buffer length in bytes.
  - **Description**
    - Popes *length* bytes data from input FIFO and puts them to the *dataOutBuff* data buffer.
- **startEnv()**
  - **Syntax**
    - *startEnv()*
  - **Arguments**
    - *non*
  - **Description**
    - Starts APB slave environment. This command must be called before first transaction is initiated by APB master.

- **printFailedTrInfo()**
  - **Syntax**
    - *printFailedTrInfo()*
  - **Arguments**
    - *non, returns number of failed transactions*
  - **Description**
    - Displays all time out errors occurred during full simulation time. This function returns the number of all errors in the buffer. Please not that this function clears failed transactions buffer.
- **fifoReset()**
  - **Syntax**
    - *fifoReset()*
  - **Arguments**
    - *non*
  - **Description**
    - Removes all data words from input and output FIFOs.

Integration and Usage

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

For usage the following steps should be done:

1. Import *APB_S* package into your test.
   - **Syntax***: import APB_S::*;*
2. Create *ABP_s_env* class object
   - **Syntax:** *ABP_s_env apb = new(apb_ifc, 4);*
   - **Description:** *apb_ifc* is the reference to the APB Slave interface instance name. The second argument is APB bus data width in bytes.
3. Start APB Slave Environment
   - **Syntax:** *apb.startEnv();*

This is the all you need for integration. Now you can configure the VIP and start data processing.

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 created *ABP_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 so the best way is to wait several clock cycles before DUT reset is done.
3. Call *printFailedTrInfo()* at the end of the test. As this function will empty error buffer so you will loss error information calling it in the middle of test. Error messages will be lost in you terminal and it will be difficult to find them after test is done.
4. If you want to use *rndDelayEn()* function be sure that arguments are correct. The argument specifying minimum value should not be more then the argument specifying maximum value.
5. As all configuration commands are non queued and all Data Transfer Commands are queued so you will have synchronization issues if want to apply specific configuration to current transaction. Here one working around for that. Be sure that transaction buffer is empty. For that you can use readWord() command with dummy address. After this command change configuration and then write your data. After writing data you should wait for write transaction done. Use readWord() command again with dummy address.

Slave Tips

1. Call *startEnv()* task as soon as you created *ABP_s_env* object before any other commands. You should call it not more then once for current object. Also be sure that this task was called before first valid transaction initiated by APB master.
2. Call *printFailedTrInfo()* at the end of the test. As this function will empty error buffer so you will loss error information calling it in the middle of test. Error messages will be lost in you terminal and it will be difficult to find them after test is done.
3. If you want to use *rndDelayEn()* function be sure that arguments are correct. The argument specifying minimum value should not be more then the argument specifying maximum value.
4. For the commands which processing words (*putWord, getWord and pollWord*) the word size is equal to the APB bus data size. It is set during APB VIP object creation. If data size is 2 bytes (16 bits) then *putWord* function will write only 2 bytes to the internal memory. First byte will be written at address provided by argument and the second byte to the incremented address. When you are using *pollWord* command be sure that polling data is not more than allowed maximum value. The allowed maximum value depends on APB bus data size. When APB bus data size is 8 bits the allowed maximum value is 255, if 16 bits then it is 0xffff.