Co-Simulation Library for AMBA AXI BFM using DPI/VPI

Version 0 Revision 1

Aug. 20, 2021 (Aug. 20, 2021)

Ando Ki, Ph.D.
andoki@gmail.com

Copyright

Abstract

This document presents a co-simulation Direct Programming Interface (DPI) and Verilog Programming Interface (VPI) library for AMBA AXI system, which connect C/Python program and SystemVerilog/Verilog simulator through Bus Functional Model (BFM). This co-simulation library uses Inter-Process Communication (IPC) mechanism to exchange messages between two independent processes in full-duplex fashion.

Table of Contents

1 Introduction

As most digital hardware blocks are connected to the system bus, it is effective to use BFM (Bus Functional Model, Bus Functional Module) to verify the functionality of the hardware block. As shown in Figure 1, it is more realistic to verify the functionality of the block at the bus transaction level rather than at the signal level.

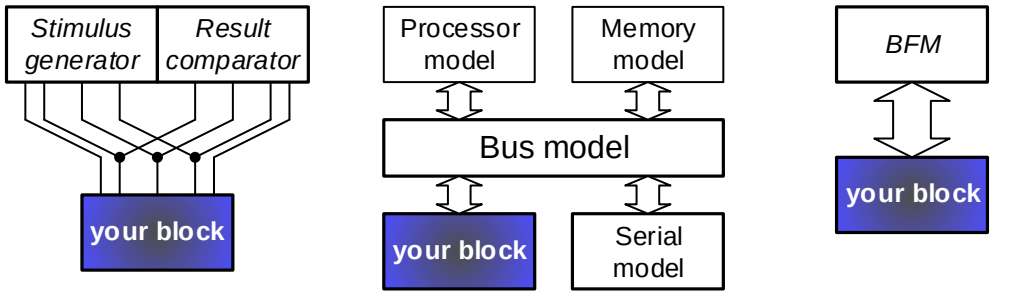

Figure 1: BFM-based verification

Among the various methods using BFM, the method of co-simulation of C program and HDL (Hardware Description Language) simulator is considered as shown in Figure 2. The C program calls BFM library API (Application Programming Interface) in order to send/receive transaction information to/from HDL simulator, which generates bus transaction.

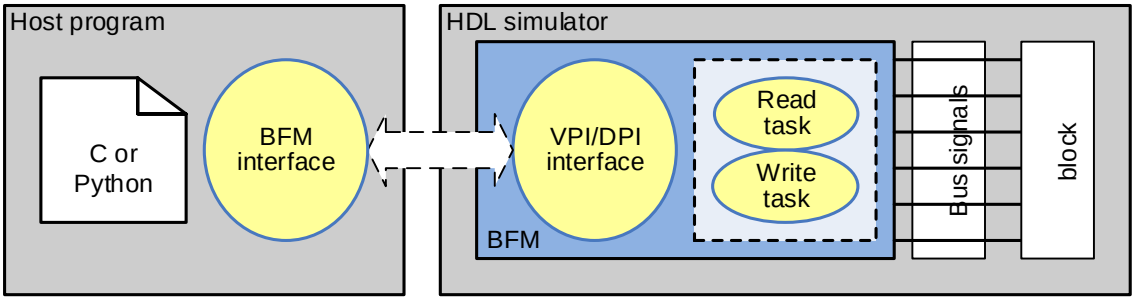

Figure 2: Co-simulation using BFM

2 Getting started

This section shows how to run this library in short, where a hardware model consisting of BFM and Memory is tested using C program as shown in Figure 3.

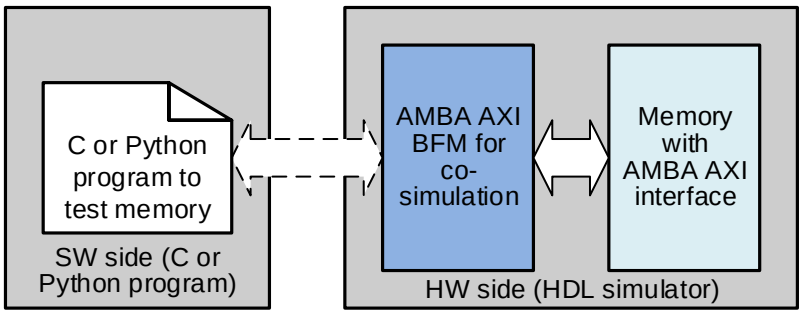

Figure 3: Memory testing example

2.1 DPI case on Ubuntu

This case uses Xilinx simulator, i.e., Vivado xsim, since a license free version is available and supports DPI.

- Step 1: install Vivado Webpack, say 2019.2 version[1]
- Step 2: prepare DPI library, where '$COSIM_HOME' stands for the directory for this project

```
$ cd $COSIM_HOME/lib_bfm
$ make -f Makefile.xsim
$ make -f Makefile.xsim install
```

- Step 3: prepare two command windows
- Step 4: at one command window; run xsim simulator

```
$ cd $COSIM_HOME/verification/test_axi_dpi_vpi/hw/sim/xsim
$ make
```

- Step 5: at the other command window; run software

```
$ cd $COSIM_HOME/verification/test_axi_dpi_vpi/sw
$ make
$ make run
```

2.2 VPI case on Ubuntu

This case uses Icarus Verilog simulator[2], since it supports VPI.

- Step 1: install Icarus Verilog, if required
- Step 2: prepare VPI library, where '$COSIM_HOME' stands for the directory for this project

```
$ cd $COSIM_HOME/lib_bfm
$ make -f Makefile.iverilog
$ make -f Makefile.iverilog install
```

- Step 3: prepare two command windows
- Step 4: at one command window; run iverilog simulator

```
$ cd $COSIM_HOME/verification/test_axi_dpi_vpi/hw/sim/iverilog
$ make
```

[1] Xilinx Vivado WebPack from https://www.xilinx.com/support/download.html
[2] Icarus Verilog simulator: http://iverilog.icarus.com/

- Step 5: at the other command window; run software

```
$ cd $COSIM_HOME/verification/test_axi_dpi_vpi/sw
$ make
$ make run
```

3 Installation

3.1 DPI case of Xsim on Ubuntu

This case uses Xilinx simulator, i.e., Vivado xsim, since a license free version is available and supports DPI.

- Step 1: install Vivado Webpack, say 2019.2 version[3]
- Step 2: prepare DPI library, where '$COSIM_HOME' stands for the directory for this project
  'DIR_INSTALL=path' is not given, ".." by default.

```
$ cd $COSIM_HOME/lib_bfm
$ make -f Makefile.xsim
$ make -f Makefile.xsim install DIR_INSTALL=path
```

- Step 3: check installation
  - ✧ Header file: $COSIM_HOME/include
  - ✧ DPI library file: $COSIM_HOME/lib/xsim/linux_x86_64
  - ✧ BFM RTL file: $COSIM_HOME/include/verilog/cosim_bfm_axi_dpi.sv
  - ✧ Python file: $COSIM_HOME/include/python/cosim_bfm.py

3.2 VPI case of iverilog on Ubuntu

This case uses Icarus Verilog simulator[4], since it supports VPI.

- Step 1: install Icarus Verilog, if required
- Step 2: prepare VPI library, where '$COSIM_HOME' stands for the directory for this project
  'DIR_INSTALL=path' is not given, ".." by default.

```
$ cd $COSIM_HOME/lib_bfm
$ make -f Makefile.iverilog
$ make -f Makefile.iverilog install DIR_INSTALL=path
```

- Step 3: check installation
  - ✧ Header file: $COSIM_HOME/include
  - ✧ DPI library file: $COSIM_HOME/lib/iverilog/linux_x86_64
  - ✧ BFM RTL file: $COSIM_HOME/include/verilog/cosim_bfm_axi_vpi.v
  - ✧ Python file: $COSIM_HOME/include/python/cosim_bfm.py

3.3 Code relationship

[3] Xilinx Vivado WebPack from https://www.xilinx.com/support/download.html
[4] Icarus Verilog simulator: http://iverilog.icarus.com/

Figure 1 shows a rough view of how the sources and libraries are related.

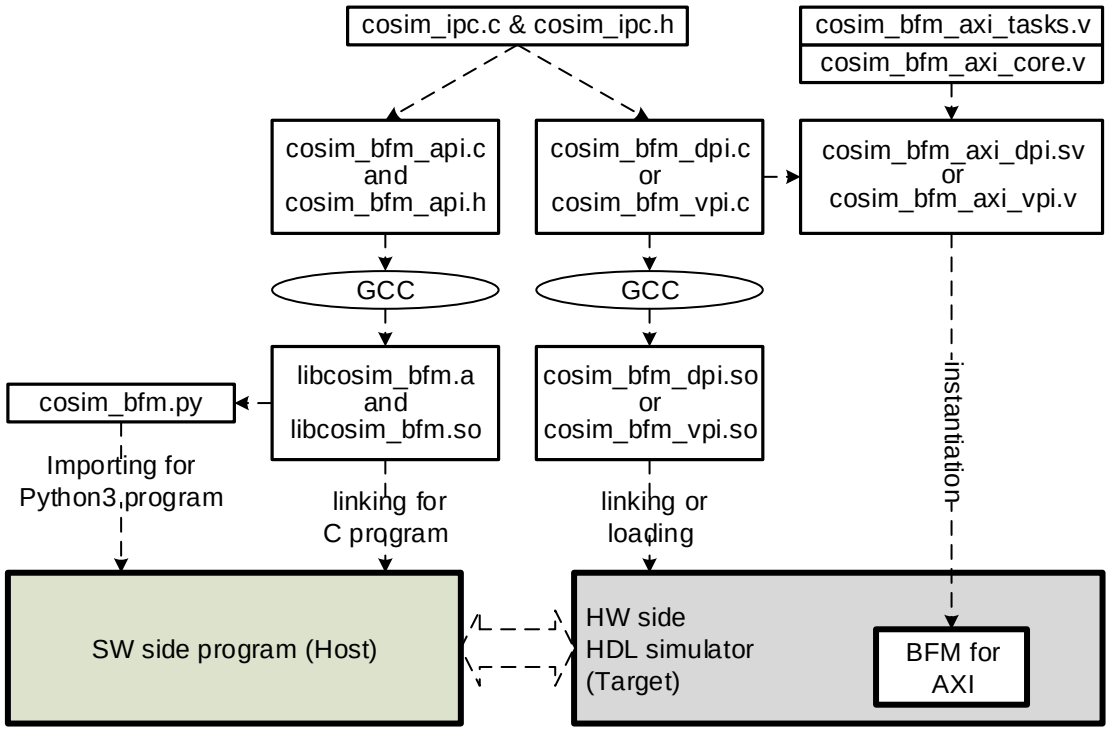

Figure 4: Code relationship

4 C API for software side

For software side, there are two group of BFM co-simulation API as follows:

- Management routines including channel-management and others
- Sending and receiving packets in pre-defined order that contains all necessary information of bus transaction

Following code segment shows an example usage of C API, where ‘bfm_open()’ and ‘bfm_barrier()’ should be called in order to synchronize hardware side.

```
#include “cosim_bfm_api.h”

int main() {
    int cid = 0;
    bfm_set_verbose(1); // optionally set verbosity level
    bfm_open(cid);
    bfm_barrier(cid);
    // now bfm_write(), bfm_read() can be used to access hw
    bfm_close(cid);
    return 0;
}
```

BFM API is a wrapper function of IPC API to exchange message of pre-defined packet and order.

4.1 API for management

4.1.1 Open: bfm_open()

'bfm_open()' creates a communication channel.

```
int bfm_open(int cid);
```

- cid: channel identification, 0 by default.
- It returns 0 on success, otherwise negative on failure.

4.1.2 Close: bfm_close()

'bfm_close()' close and remove the communication channel.

```
int bfm_close(int cid);
```

- cid: channel identification, 0 by default.
- It returns 0 on success, otherwise negative on failure.

4.1.3 Barrier: bfm_barrier()

'bfm_barrier()' waits for joining of hw side.

```
int bfm_barrier(int cid);
```

- cid: channel identification, 0 by default.
- It returns 0 on success, otherwise negative on failure.

The host (SW side) and the target (HW side) exchange pre-defined messages in order to synchronize each other.

Host (SW side)	Target (HW side)
1. send a message containing its PID (process identification number) to the target 2. wait for a message from the target, which contains the target PID 3. send a message containing the target PID to target 4. wait for message from the target	1. wait for message from the host, which contains the host PID 2. send a message containing its PID to host 3. wait for message from the host 4. send a message containing the host PID to the target

4.1.4 Verbosity set: bfm_set_verbose()

‘bfm_set_verbose()’ sets verbosity level.

```
int bfm_set_verbose(int level);
```

- level: verbosity level, 0 by default with no or minimal message
- It returns 0 on success, otherwise negative on failure.

4.1.5 Verbosity check: bfm_get_verbose()

‘bfm_get_verbose()’ returns current verbosity level.

```
int bfm_get_verbose(void);
```

- It returns the current verbosity level.

4.2 API for transactions

4.2.1 Write transaction: bfm_write()

‘bfm_write()’ makes BFM hardware generate a bus write transaction.

```
int bfm_write( uint32_t  addr
             , uint8_t  *data
             , unsigned int sz
             , unsigned int length);
```

- addr: starting address of the transaction and should be ‘sz’ aligned
- data: pointer to the byte-buffer containing ‘sz*length’ bytes in little-endian byte-stream
- sz: number of bytes for each beat of burst
- length: number of beats of the burst
- It returns 0 on success, otherwise return negative number on failure.

4.2.2 Read transaction: bfm_read()

‘bfm_read()’ makes BFM hardware generate a bus read transaction.

```
int bfm_read ( uint32_t  addr
             , uint8_t  *data
             , unsigned int sz
             , unsigned int length);
```

- addr: starting address of the transaction and should be ‘sz’ aligned
- data: pointer to the byte-buffer to contain ‘sz*length’ bytes in little-endian byte-stream
- sz: number of bytes for each beat of burst
- length: number of beats of the burst

- It returns 0 on success, otherwise return negative number on failure.

5 Python functions for software side

Following code segment shows an example usage of Python functions, where 'cosim_bfm_open()' and 'cosim_bfm_barrier()' should be called in order to synchronize hardware side.

```
#!/usr/bin/env python3

import cosim_bfm as cosim

simulator = xsim # can be "iverilog"
cid = 0

cosim.LoadCosimLib(simulator)
cosim.bfm_open(cid)
cosim.bfm_barrier(cid)

... your code using cosim_bfm_write() and cosim_bfm_read() ...

cosim.bfm_close(cid)
```

Python3 should be used for this functions.

5.1 Functions for management

5.1.1 Loading: LoadCosimLib()

'LoadCosimLib()' loads shared library and COSIM_HOME environment variable specifies where it is.

```
LoadCosimLib( simulator, rigor=False, verbose=False )
```

- simulator: specifies HDL simulator and xsim or iverilog is supported for this version.
  - use 'xsim' for DPI case
  - use 'iverilog' for VPI case
- rigor:
- verbose:
- It returns 0 on success, otherwise negative on failure.

5.1.2 Open: bfm_open()

'cosim_bfm_open()' creates a communication channel.

```
bfm_open( cid, rigor=False, verbose=False )
```

- cid: channel identification, 0 by default.
- rigor:
- verbose:
- It returns 0 on success, otherwise negative on failure.

5.1.3 Close: bfm_close()

'bfm_close()' close and remove the communication channel.

```
bfm_close( cid, rigor=False, verbose=False )
```

- cid: channel identification, 0 by default.
- rigor:
- verbose:
- It returns 0 on success, otherwise negative on failure.

5.1.4 Barrier: bfm_barrier()

'bfm_barrier()' waits for joining of hw side.

```
int bfm_barrier( cid, rigor=False, verbose=False )
```

- cid: channel identification, 0 by default.
- rigor:
- verbose:
- It returns 0 on success, otherwise negative on failure.

5.1.5 Verbosity set: bfm_set_verbose()

'bfm_set_verbose()' sets verbosity level.

```
int bfm_set_verbose( rigor=False, verbose=False )
```

- level: verbosity level, 0 by default with no or minimal message
- rigor:
- verbose:
- It returns 0 on success, otherwise negative on failure.

5.1.6 Verbosity check: bfm_get_verbose()

'bfm_get_verbose()' returns current verbosity level.

```
int bfm_get_verbose( rigor=False, verbose=False )
```

- rigor:
- verbose:

- It returns the current verbosity level.

5.2 Functions for transactions

5.2.1 Write transaction: bfm_write()

'bfm_write()' makes BFM hardware generate a bus write transaction.

```
bfm_write( addr
           , data
           , sz=4
           , length=1
           , rigor=False
           , verbose=False )
```

- addr: starting address of the transaction and should be 'sz' aligned
- data: pointer to the byte-buffer containing 'sz*length' bytes in little-endian byte-stream
- sz: number of bytes for each beat of burst
- length: number of beats of the burst
- rigor:
- verbose:
- It returns 0 on success, otherwise return negative number on failure.

5.2.2 Read transaction: bfm_read()

'bfm_read()' makes BFM hardware generate a bus read transaction.

```
bfm_read ( addr
           , data
           , sz=4
           , length=1
           , rigor=False
           , verbose=False )
```

- addr: starting address of the transaction and should be 'sz' aligned
- data: pointer to the byte-buffer to contain 'sz*length' bytes in little-endian byte-stream
- sz: number of bytes for each beat of burst
- length: number of beats of the burst
- rigor:
- verbose
- It returns 0 on success, otherwise return negative number on failure.

6 DPI functions for hardware side

For hardware side, there are DPI functions that are corresponds to those of C API.

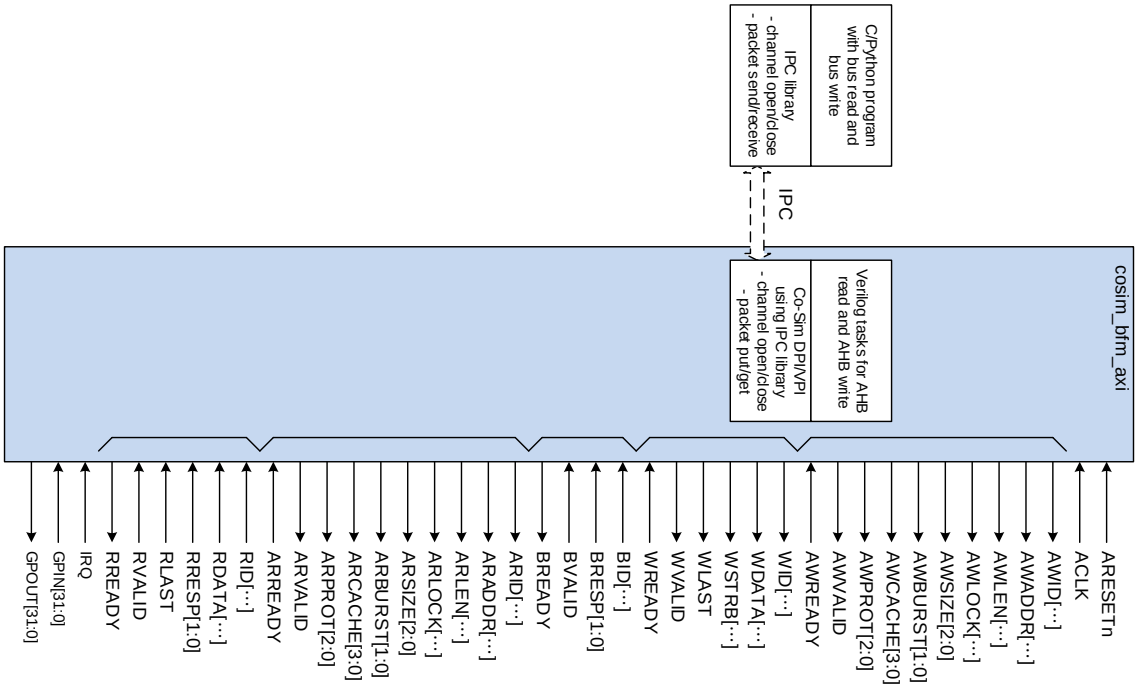

Figure 5: AMBA AXI BFM

Following DPI functions are used in AMBA AXI BFM model and these are wrapper functions of IPC API to exchange message of pre-defined packet and order.

```
import "DPI-C" cosim_ipc_open   =function int cosim_ipc_open   (input int cid);
import "DPI-C" cosim_ipc_close  =function int cosim_ipc_close  (input int cid);
import "DPI-C" cosim_ipc_barrier=function int cosim_ipc_barrier(input int cid);
import "DPI-C" cosim_ipc_get    =function int cosim_ipc_get(
                 input  int       cid  // IPC channel identification
               , output int       pkt_cmd  // see cosim_bfm_defines.vh
               , output int       pkt_size  // 1, 2, 4
               , output int       pkt_length  // burst length
               , output int       pkt_ack
               , output int       pkt_attr
               , output int       pkt_trans_id
               , output int       pkt_addr
               , output bit [7:0] pkt_data[]  // open-array
               );
import "DPI-C" cosim_ipc_put  =function int cosim_ipc_put(
                 input  int       cid
               , input  int       pkt_cmd
               , input  int       pkt_size // 1, 2, 4
               , input  int       pkt_length // burst length
               , input  int       pkt_ack
```

```
                , input  int       pkt_attr
                , input  int       pkt_trans_id
                , input  int       pkt_addr
                , input  bit [7:0] pkt_data[] // open-array
               );
import "DPI-C" cosim_ipc_rcv  =function int cosim_ipc_rcv(
                  input  int       cid
                , output int       bnum
                , output bit [7:0] pkt_data[]  // open-array
                );
import "DPI-C" cosim_ipc_snd  =function int cosim_ipc_snd(
                  input  int       cid
                , input  int       bnum
                , input  bit [7:0] pkt_data[] // open-array
               );
import "DPI-C" cosim_ipc_set_verbose=function int cosim_ipc_set_verbose(
                  input  int       level
               );
```

7 VPI functions for hardware side

For hardware side, there are VPI functions that are corresponds to those of C API.

Following VPI functions are used in AMBA AXI BFM model and these are wrapper functions of IPC API to exchange message of pre-defined packet and order.

```
$cosim_ipc_open(cid);
$cosim_ipc_close(cid);
$cosim_ipc_barrier(cid);
$cosim_ipc_set_verbose(n);
$cosim_ipc_get(cid
              ,pkt_type
              ,pkt_size  // 1, 2, 4
              ,pkt_length  // burst length
              ,pkt_ack
              ,pkt_attr
              ,pkt_trans_id
              ,pkt_addr
              ,pkt_data);// array
$cosim_ipc_put(cid
              ,pkt_type
              ,pkt_size // 1, 2, 4
              ,pkt_leng // burst length
              ,pkt_ack
              ,pkt_attr
              ,pkt_trans_id
              ,pkt_addr
              ,pkt_data);// array
```

```
$cosim_ipc_rcv(cid       // channel id
               ,bnum      // num of bytes received
               ,pkt_data);// array
$cosim_ipc_snd(cid       // channel id
               ,bnum      // num of bytes in 'pkt_data[]'
               ,pkt_data);// data buffer
```

8 IPC flow and packet structure

8.1 IPC flow

In order to send/receive packet over IPC channel, the channel should be built beforehand, as shown in Figure 6. After establishing channel, message passing mechanism is used to communicate between two parties. Each transaction consists of two messages; one is request and the other is response. For write transaction, all necessary information including address and data are packed in to the packet and the packet is forwarded to the receiver. The receiver prepares a return packet, i.e., response packet and sends it back to the sender. For read transaction, request packet is forwarded to the receiver and then the receiver returns response packet that contains read data.

Co-simulation API's are based on IPC API and refer to [1] for more details.

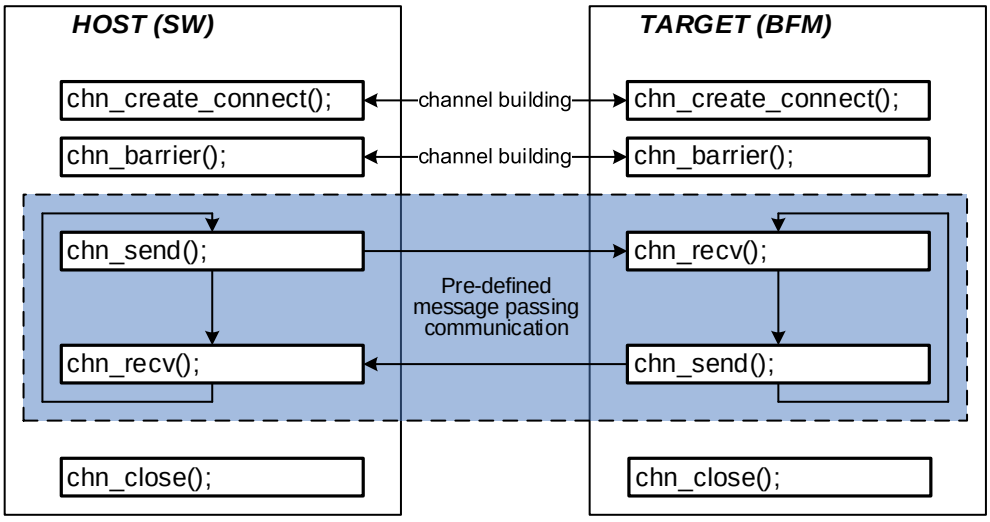

Figure 6: IPC flow

8.2 packet for co-simulation

C API sends and receives message to and from HDL simulator over IPC, in which following packet type is used.

```
// refer to 'cosim_bfm_defines.h
// for 'cmd_type' field
```

```
#define COSIM_CMD_NULL       0x00  // skip
#define COSIM_CMD_RD_REQ     0x01
#define COSIM_CMD_WR_REQ     0x02
#define COSIM_CMD_RD_RSP     0x05
#define COSIM_CMD_WR_RSP     0x06
#define COSIM_CMD_TERM_REQ   0x08
#define COSIM_CMD_TERM_RSP   0x09
#define COSIM_CMD_GET_GP_REQ  0x11
#define COSIM_CMD_PUT_GP_REQ  0x12
#define COSIM_CMD_GET_GP_RSP  0x15
#define COSIM_CMD_PUT_GP_RSP  0x16
#define COSIM_CMD_GET_IRQ_REQ 0x21
#define COSIM_CMD_GET_IRQ_RSP 0x22
//------------------------------------------------------------------------------
// for 'cmd_ack' field
#define COSIM_CMD_ACK_ERR    0x0
#define COSIM_CMD_ACK_OK     0x1
//------------------------------------------------------------------------------
// for 'data[]' field
#define COSIM_DATA_BNUM      1024// 256-beat * 4-byte
```

```
// refer to ‘cosim_bfm_packet.h’
//------------------------------------------------------------------------------
// data[...] contains 'cmd_size*cmd_length' bytes
typedef struct {
  unsigned int cmd_type;
  unsigned int cmd_size;   // num of bytes in a beat
  unsigned int cmd_length; // num of beats, i.e., burst length
  unsigned int cmd_ack;    // ERR(0), OK(1)
  unsigned int attr;       // user-specified attribute
  uint32_t     trans_id;   // transaction identification (for multiple outstanding case)
  uint32_t     addr;
  uint8_t      data[COSIM_DATA_BNUM]; // byte-stream
} bfm_packet_t;
```

‘bfm_packet_t’ data structure is used as payload of packet, which consists as follows:

- ✧ unsigned int cmd_type: specifies what kind of command and one of ‘COSIM_CMD_???’
- ✧ unsigned int cmd_size: number of bytes for each beat
- ✧ unsigned int cmd_length: number of burst length consisting of ‘cmd_size’ of beats.
- ✧ unsigned int cmd_ack: valid when ‘cmd_type’ is response and one of ‘COSIM_CMD_ACK_???
- ✧ unsigned int attr: additional filed that can be used by any purposes

- ✧ uint32_t trans_id: identification of each transaction in order to support multiple outstandinng requests
- ✧ uint32_t addr : address
- ✧ uint8_t data[...] : data buffer in little-endian style byte-stream

9 The 2-Clause BSD License

10 Acknowledgment

This work is developed by Ando Ki (andoki@gmail.com) while he is working for Future Design Systems.

11 Wish list

- Use transaction identification in order to support multiple outstanding feature
- Use requester identification in order to support multiple requesters.
- Use multi-threading feature to handle arbitrary messages.

12 References

[1] Ando Ki, IPC Library for Co-simulation, Aug. 1, 2021.

[2] IEEE standard Verilog hardware description language. IEEE Standard IEEE Std 1364-2001.

[3] IEEE standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language, EEE Std. 2800-2012.

[4] Stuart Sutherland, The Verilog PLI Handbook, second edition", Kluwer Academic Publishers, 2002.

13 Revision history

- ☐ 2021.08.20: V0R1 released by Ando Ki.
- ☐ 2021.07.01: Document started by Ando Ki (andoki@gmail.com)

-- End of Document --