TRX_AXI: AMBA AXI Transactor for GPIF2MST

Version 0 Revision 5

Dec. 30, 2019 (April 27, 2018)

Future Design Systems, Inc.
www.future-ds.com / contact@future-ds.com

Abstract

This document addresses AMBA AXI transactor for gpif2mst of GPIF-II master.

Table of Contents

1 Overview

As shown in Figure 1, this is about AMBA AXI transactor (trx_axi) that interacts with GPIF-II master (i.e., gpif2mst) through transactor interface.

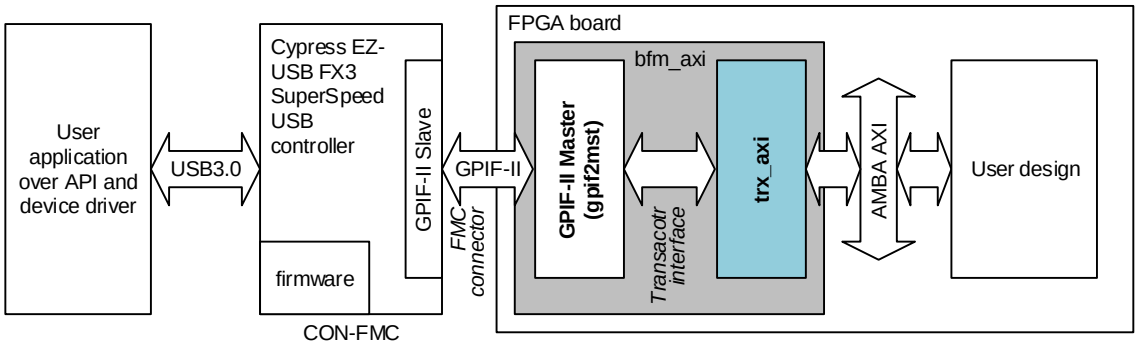

Figure 1: Overview

The trx_axi is a kind of BFM (Bus Functional Module) and generates AMBA AXI transactions.

Followings are features supported by the transactor.

- AXI 3 and AXI 4 are supported
- Configurable AXI address bit width
- Configurable AXI data bit width: 32, 64

Followings are limitations.

- Each starting address of a burst should be aligned to its transfer size
- Read and write data bit width should be the same
- 32-bit data is handled even though AXI data width is bigger than 32-bit
- Big-endian data ordering is not implemented yet, i.e., little-endian is used

2 Transactor

Figure 2 shows how the transactor connects with its related blocks, where the transactor interacts with GPIF-II master through transactor interface and generates AMBA AXI transactions as an AMBA AXI master.

Unlike other AXI master, this block gets ‘MID[…]’ as input in order to support any master ID.

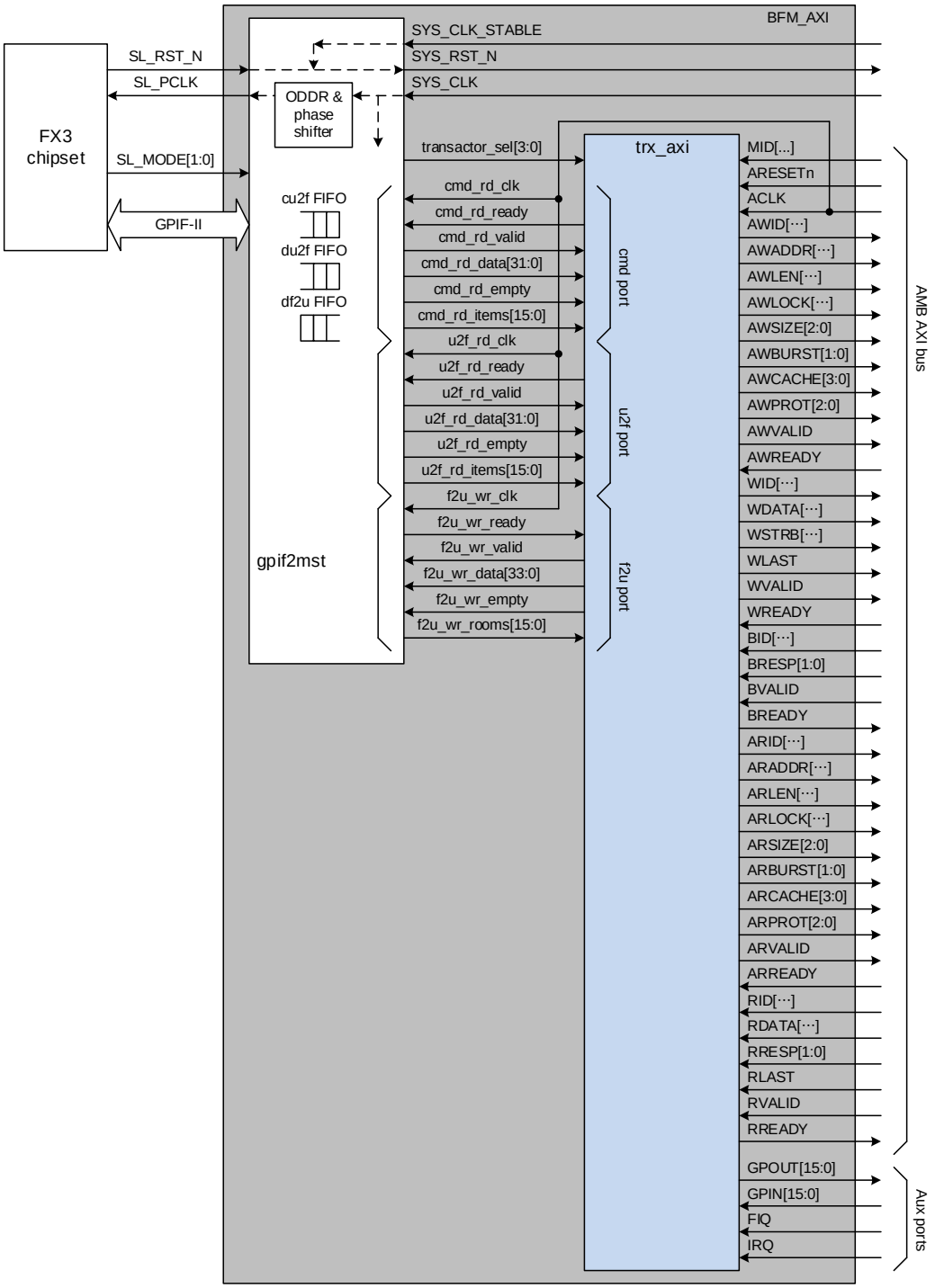

Figure 2: TRX_AXI connection

In order to the transactor works correctly, ‘SL_MODE[1:0]’ should be 2’b00, which indicates the GPIF-II master controls through three FIFO interface, i.e., transactor interface.

The following macros determines if a specific feature of AXI is used or not.

Table 1: Macros

Macro name	Default value	Meaning
AMBA_AXI4	Not defined	AXI4 feature enabled when defined (AxLEN, AxLOCK, AxQOS, AxREGION)
AMBA_AXI_CACHE	Not defined	'AxCACHE[…]' used when defined
AMBA_AXI_PROT	Not defined	'AxPROT[…]' used when defined
ENDIAN_BIG	Not defined	Not implemented yet.

The following parameters are used in order to configure this module.

Table 2: Parameters

Parameter name	Default value	Meaning
AXI_MST_ID	1	Determines master identification
AXI_WIDTH_CID	4	Additional ID fields for AXI slave that will be extra bits of AWID, WID, BID, ARID, RID.
AXI_WIDTH_ID	4	Determines width of AWID, WID, BID, ARID, RID.
AXI_WIDTH_AD	32	Determines width of AWADDR, ARADDR.
AXI_WIDTH_DA	32	Determines width of WDATA and RDATA. It can be 32 or 64.

3 Transactor interface

Transactor interface connects GPIF-II master (gpif2mst) and AMBA AXI transactor (trx_axi) and consists of three ready-valid FIFO ports.

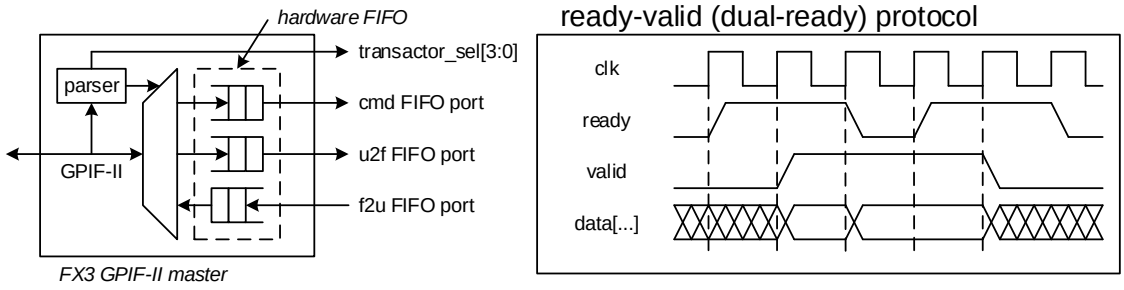

Figure 3: Tranactor interface

In ready-valid (dual-ready) protocol, data are transferred when 'ready' and 'valid' are both 1.

3.1 CMD-FIFO

It carries command data from USB to Transactor.

- cmd_rd_clk: clock with rising-edge synchronized operation; it should be HCLK.
- cmd_rd_ready: transactor is ready to accept data when 1.
- cmd_rd_valid: master is driving data in ‘cmd_data[]’.
- cmd_data[31:0]: valid data when ‘cmd_rd_valid’ is 1.
- cmd_rd_items[15:0]: the number of words available when ‘cmd_rd_valid’ is 1.

3.2 U2F-FIFO

It carries pure data from USB to Transactor.

- u2f_rd_clk: clock with rising-edge synchronized operation; it should be HCLK.
- u2f_rd_ready: transactor is ready to accept data when 1.
- u2f_rd_valid: master is driving data in ‘u2f_data[]’.
- u2f_data[31:0]: valid data when ‘u2f_rd_valid’ is 1.
- u2f_rd_items[15:0]: the number of words available when ‘u2f_rd_valid’ is 1.

3.3 F2U-FIFO

It carries command data from Transactor to USB.

- f2u_rd_clk: clock with rising-edge synchronized operation; it should be HCLK.
- f2u_rd_ready: master is ready to accept data when 1.
- f2u_rd_valid: transactor is driving data in ‘f2u_data[]’.
- f2u_data[31:0]: valid data when ‘f2u_rd_valid’ is 1.
- f2u_rd_rooms[15:0]: the number of rooms available when ‘f2u_rd_valid’ is 1.

3.4 Communication modes

This design supports two communication models.

- Pseudo DMA mode: It uses command packet to control communication.
- Stream mode: It uses one or two streams without command.

This communication mode is set by ‘SL_MODE[1:0]’.

- SL_MODE[0] enables stream USB-to-FPGA.
- SL_MODE[1] enables stream FPGA-to-USB.

‘SL_MODE[1:0]’ can be 2’b11 or 2’b00, where 2’b00 means pseudo DMA and 2’b11 enables both streams. ‘SL_RST_N’ should be driven a few cycles before driving ‘SL_MODE[1:0]’.

4 AMBA AXI BUS

Refer to AMBA AXI specification.

5 Transactor packet

5.1 Command packet

There two categories of command; one is external and the other is internal access. The external access command generates AMAB AXI transactions, while the internal access command is for other purposes.

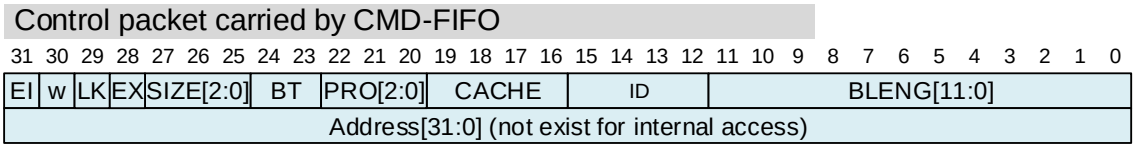

Figure 4: Control packet format

Table 3: AXI access command field

bit field	name	meaning
31	internal or external	External (i.e., AXI) access when 1
30	write or read	Write when 1 and read when 0
29	lock	Lock when 1
28	exclusive	Exclusive when 1
27:25	size	Num of bytes per beat: 1<<size 3'b000: 1-byte 3'b001: 2-byte … 3'b111: 128-byte
24:23	burst type	0 for fixed, 1 for incremental, 2 for wrapping
22:20	protection	The same as AXI specification
19:16	cache	The same as AXI specification
15:12	transaction id	Determines AWID or ARID
11:0	burst length	Num of beats in a burst ● 0 means 1-beat ● 1 means 2-beat ● N means (N+1)-beat It can be 15 for AXI 3, i.e, 16-beat. It can be 255 for AXI 4, i.e., 256-beat. More than 256, the transactions will be divided into a series of 256 bursts for AXI4.

5.2 Write packet

It uses CMD-FIFO and U2F-FIFO to get command and data, respectively. As shown Figure 5, command and starting address come from CMD FIFO, and then a series of data comes from U2F-FIFO. Each data contains the number of justified bytes specified by (size) bit field. The address of each data is determined as follows.

- 1st address: ‘address’ given at command
- 2nd address: ‘address + (1<<size)
- 3rd address: ‘address + (1<<size) * 2
- …
- Nth address: ‘address + (1<<size) * (N – 1)

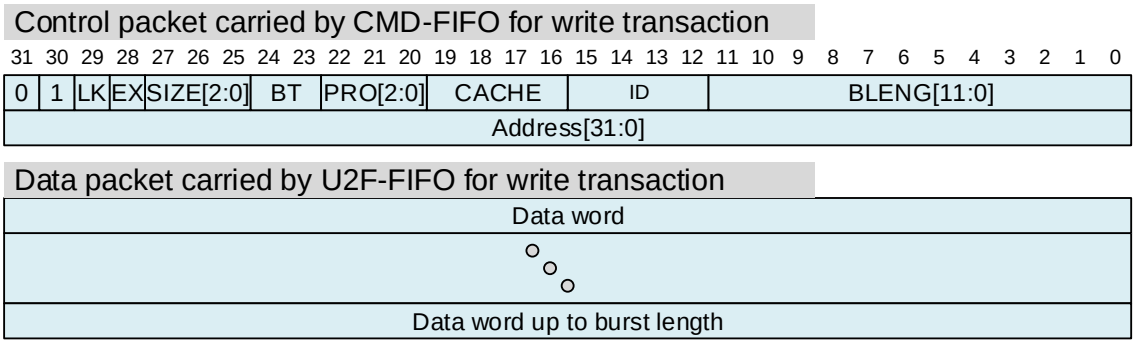

Figure 5: AMBA AXI write transaction

5.3 Read packet

It uses CMD-FIFO and F2U-FIFO to get command and put data, respectively. As shown Figure 6, command and starting address come from CMD FIFO, and then a series of data puts through F2U-FIFO. Each data contains the number of justified bytes specified by (size) bit field. The address of each data is determined as follows.

- 1st address: ‘address’ given at command
- 2nd address: ‘address + (1<<size)
- 3rd address: ‘address + (1<<size) * 2
- …
- Nth address: ‘address + (1<<size) * (N – 1)

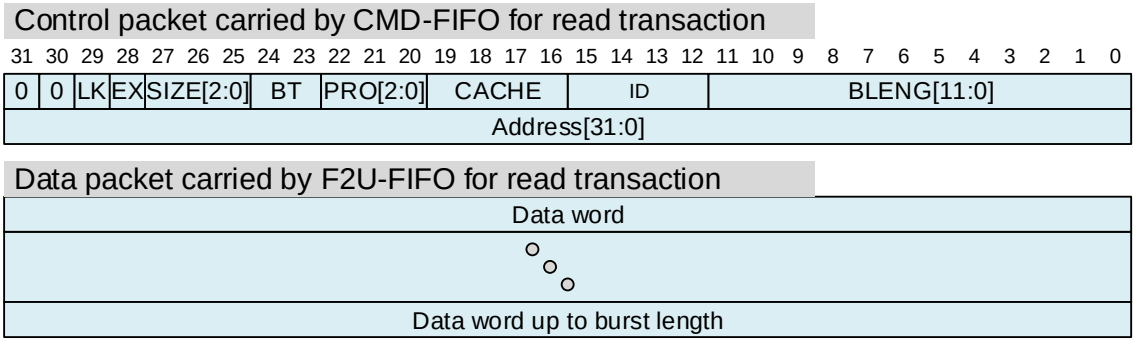

Figure 6: AMBA AXI read transaction

5.4 Internal write packet

When command carries the data shown in 오류! 참조 원본을 찾을 수 없습니다., it drives the data in bit field ’15:0’ to GPOUT port.

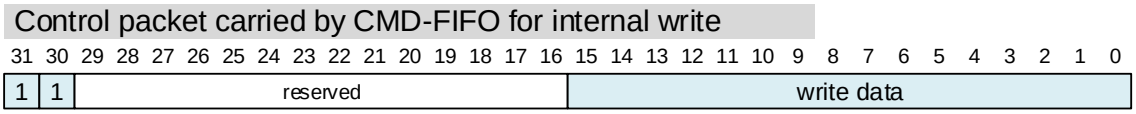

Figure 7: Internal write

5.5 Internal read packet

When command carries the data shown in 오류! 참조 원본을 찾을 수 없습니다., it returns informative data through READ FIFO, which contains the following.

- Value of IRQ and FIQ ports
- Status of the last transaction, i.e., WRESP or RRESP.
- Value of master ID
- AMBA AXI4 or not
- Value of bit-width of data bus and related to ‘WIDTH_DA’ parameter.
  - 3’b000: 0ne-byte
  - 3’b001: two-byte
  - 3’b010: four-byte (32-bit)
  - 3’b011: eight-byte (64-bit)
  - 3’b100: sixteen-byte (128-bit)

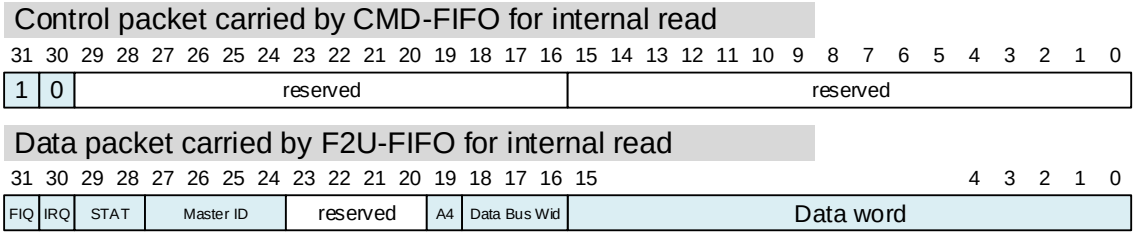

Figure 8: Internal read

6 C API

There are only two API to control AXI transaction and three API to control extra features.

6.1 BfmWrite()

'BfmWrite()' generates a write burst transaction of 'length' beats starting from 'addr' and each beat writes 'size' bytes of data.

```
void BfmWrite( con_Handle_t handle
             , unsigned int  addr
             , unsigned int *data
             , unsigned int  size
             , unsigned int  length);
```

- handle: CON-FMC hander
- addr: starting address; should be aligned at the size of bytes to transfer in a beat[1].
- data[]: array containing data to be written and it is justified
- size: number of bytes at each beat; 1, 2, 4.
- length: number of beats in a burst[2].

Each data in 'data' array will be written at different address as follows.

- Data[0] → MEM[address+size-1:address]
- Data[1] → MEM[address+size*2-1:address+size]
- Data[2] → MEM[address+size*3-1:address+size*2]
- …
- Data[n] → MEM[address+size*(n+1)-1:address+size*n]

This API only handles 32-bit data. When AXI data bus (WDATA) is 64-bit wide, upper or lower half word (4-byte) is used at each transaction depending on address. In other words, AWSIZE can be 3'b000, 3'b001, 3'b010.

```
#define GET_CMD(CMD,EI,WR,LK,EX,SZ,BT,PR,CA,ID,BL)\
        CMD    = ((EI)&0x1)<<31\
               | ((WR)&0x1)<<30\
               | ((LK)&0x1)<<29\
               | ((EX)&0x1)<<28\
               | ((SZ)&0x7)<<25\
               | ((BT)&0x3)<<23\
               | ((PR)&0x7)<<20\
```

[1] When 'size' is 4, 'addr' should be a multiple of 4.
[2] Since AMBA AXI3 only supports a burst of up to 16 beats, this API makes several bursts when 'length' is larger than 16.

```
                    | ((CA)&0xF)<<16\
                    | ((ID)&0xF)<<12\
                    | ((BL)&0xFFF)

//-------------------------------------------------------------
// It generates 'length' incremental write transactions
// from the address in 'addr' with data in 'data[]'.
// Note that 'data[x]' contains 'size'-bytes in justified fashion.
void BfmWrite( con_Handle_t handle, unsigned int  addr
            , unsigned int *data // pointer to the array of justified data
            , unsigned int  size // num of bytes in an item
            , unsigned int  length)
{
#ifdef RIGOR
   if (data==NULL) {
      printf("BfmWrite invalid buffer\n");
      return;
   }
   switch (size) {
   case 1: case 2: case 4: break;
   default: printf("BfmWrite cannot support %d-byte transfer\n", size);
          return;
   }
   if ((length>0x1000)&&(length<0)) {
      printf("BfmWrite can support up to 4096 for length\n");
      return;
   }
#endif
   // to push BFM command for write
   // - control-flit for command
   // - command-flit for bfm write
   // - address-flit for bfm write
   unsigned int cbuf[4];
   cbuf[0] = (2<<16) // command+address
          | ((0b0010&0xF)<<12) // control packet
          | ((0x0&0xF)<<4); // transactor
   GET_CMD(cbuf[1], 0, 1, 0, 0, size>>1, 1, 0, 0, 1, length-1);
          //       EI,WR,LK,EX, SZ     ,BT,PR,CA,ID,   BL
   cbuf[2] = addr;
   // to push BFM data for write
   // - control-flit for data
   cbuf[3] = (length<<16) //  command+data
          | ((0b0100&0xF)<<12) // control packet
          | ((0x0&0xF)<<4); // transactor

   unsigned int done=0;
   if (conStreamWrite(handle, cbuf, 4, &done, 0) || (done!=4)) {
      printf("BfmWrite() something went wrong: %d\n", done);
      return;
   }
//conZlpWrite(handle);

   // to push BFM data for write
   unsigned int num;
   unsigned int *pbuf=data;
   for (num=length, done=0; num>0; num -= done, pbuf += done) {
       unsigned int zlp = ((num*4)%handle->usb.bulk_max_pkt_size_out) ? 0 : 1;
```

```
//printf("num=%d zlp=%d\n", num, zlp);
        if (conStreamWrite(handle, (void *)pbuf, num, &done, zlp)) {
            printf("BfmWrite() something went wrong: %d\n", done);
            return;
        }
if (done<num) printf("num=%d zlp=%d done=%d\n", num, zlp, done);
   }
}
```

6.2 BfmRead()

'BfmRead()' generates a read burst transaction of 'length' beats starting from 'addr' and each beat reads 'size' bytes of data.

```
void BfmRead ( con_Handle_t hande
                    , unsigned int  addr
                    , unsigned int *data
                    , unsigned int  size
                    , unsigned int  length);
```

- handle: CON-FMC hadler
- addr: starting address; should be aligned at the size of bytes to transfer in a beat.
- data[]: array to be contain data to be read and it is justified.
- size: number of bytes at each beat; 1, 2, 4.
- length: number of beats in a burst[3].

Each data in 'data' array will be read from different address as follows.

- Data[0] ←MEM[address+size-1:address]
- Data[1] ← MEM[address+size*2-1:address+size]
- Data[2] ← MEM[address+size*3-1:address+size*2]
- …
- Data[n] ← MEM[address+size*(n+1)-1:address+size*n]

This API only handles 32-bit data. When AXI data bus (RDATA) is 64-bit wide, upper or lower half word (4-byte) is used at each transaction depending on address. In other words, ARSIZE can be 3'b000, 3'b001, 3'b010.

```
#define GET_CMD(CMD,EI,WR,LK,EX,SZ,BT,PR,CA,ID,BL)\
        CMD    = ((EI)&0x1)<<31\
                | ((WR)&0x1)<<30\
                | ((LK)&0x1)<<29\
                | ((EX)&0x1)<<28\
                | ((SZ)&0x7)<<25\
                | ((BT)&0x3)<<23\
                | ((PR)&0x7)<<20\
```

[3] Since AMBA AXI3 only supports a burst of up to 16 beats, this API makes several bursts when 'length' is larger than 16.

```
                    | ((CA)&0xF)<<16\
                    | ((ID)&0xF)<<12\
                    | ((BL)&0xFFF)

//--------------------------------------------------------------------------
// It generates 'length' incremental read transactions
// from the address in 'addr' with data in 'data[]'.
// Note that 'data[x]' contains 'size'-bytes in justified fashion.
void BfmRead( con_Handle_t handle, unsigned int  addr
          , unsigned int *data // pointer to the array of justified data
          , unsigned int  size
          , unsigned int  length)
{
#ifdef RIGOR
  if (data==NULL) {
     printf("BfmWrite invalid buffer\n");
     return;
  }
  switch (size) {
  case 1: case 2: case 4: break;
  default: printf("BfmRead cannot support %d-byte transfer\n", size);
         return;
  }
  if ((length>0x1000)&&(length<0)) {
     printf("BfmWrite can support up to 4096 for length\n");
     return;
  }
#endif
  // to push BFM command for write
  // - control-flit for command
  // - command-flit for bfm write
  // - address-flit for bfm write
  unsigned int cbuf[4];
  cbuf[0] = (2<<16) // command+address
        | ((0b0010&0xF)<<12) // control packet
        | ((0x0&0xF)<<4); // transactor
  GET_CMD(cbuf[1], 0, 0, 0, 0, size>>1, 1, 0, 0, 1, length-1);
        //       EI,RD,LK,EX, SZ     ,BT,PR,CA,ID,   BL
  cbuf[2] = addr;
  // to push BFM data for write
  // - control-flit for data
  cbuf[3] = (length<<16) // command+data
        | ((0b0101&0xF)<<12) // control packet
        | ((0x0&0xF)<<4); // transactor

  unsigned int done=0;
  if (conStreamWrite(handle, cbuf, 4, &done, 0) || (done!=4)) {
     printf("BfmRead() something went wrong\n");
     return;
  }

  // to pop BFM data for read
  unsigned int num;
  unsigned int *pbuf=data;
  for (num=length, done=0; num>0; num -= done, pbuf += done) {
      if (conStreamRead(handle, (void *)pbuf, num, &done)) {
         printf("BfmRead() something went wrong\n");
```

```
            return;
        }
    }
}

//-------------------------------------------------------------------------------
// [command fifo for internal write access]
//  31 30 29 28 27-25 24-23 22-20 19-16 15 - 0
// |EI|WR|                          | DA   |
//
//  EI      : Mode (0:external, 1:internal)
//  WR      : Write/Read (0:read, 1:write)
//  DA[15:0]: Data
//-------------------------------------------------------------------------------
// It writes GPIO pins.
//
// Return <0 on failure, 0 on success.
int BfmGpout( unsigned int value ) {
   unsigned int cbuf[1], pbuf[1];
   unsigned int done;
   cbuf[0] = 1<<31
           | 1<<30
           | (value&0xFFFF);
   if (conCmdWrite(handle, (void *)cbuf, 1, &done, 0)) {
      printf("conCmdWrite() something went wrong\n");
      return -1;
   }
   return 0;
}
```

6.3 BfmGpout()

'BfmGpout()' drives 'value' to 'GPOUT' port.

```
unsigned int TrxGpout( con_Handle_t handle, unsigned int value);
```

6.4 BfmGpin()

'BfmGpin()' reads informative data.

```
unsigned int TrxGpin ( con_Handle_t handle, unsigned int *pValue);
```

- bit [31] IRQn: Signal value of IRQn.
- bit [30] FIQn: Signal value of FIQn.
- bit [29:28] STATUS (i.e., RESP): Status of last transaction, i.e. BRESP or RRESP.
- bit [27:24] Master ID: Identification of this module as an AXI master.
- bit [18:16] Data bus bit width (1<<value): Width of data, i.e, AXI_WIDTH_DA.
- bit [15: 0] GPIN: Signal value of GPIN port.

6.5 CheckInterrupt()

'CheckInterrupt()' returns the value in 'IRQn' and 'FIQn'.

```
unsigned int CheckInterrupt( con_Handle_t handle );
```

- Bit [1] IRQn: It is the same as bit[31] of the result of TrxGpin().
- Bit [0] FIQn: It is the same as bit[30] of the result of TrxGpin().

7 Python API

This section addresses Python binding of BFM AXI, which allows the user level API (Application Programming Interface) to be called from Python.

Before using Python API, the correct path should be set at 'PYTHONPATH' environment variable, where 'pyconfmc.pyc' and 'pyconbfmaxi.pyc' reside.

7.1 Typical usage

As shown below, following two modules are required.

- confmc.pyconfmc
- confmc.pyconbfmaxi

```
import confmc.pyconfmc as confmc
import confmc.pyconbfmaxi as axi

hdl = confmc.conInit()

addr=0x00000000
depth=0x100
Wdata=[0x1, 0x2, 0x3]
Rdata=[]
axi.BfmWrite(hdl, addr, Wdata, 4, 2)
axi.BmRead(hdl, addr, Rdata, 4, 2)
print (Wdata, ":", Rdata)

confmc.conRelease(hdl)
```

7.2 BfmWrite()

'BfmWrite()' generates a write burst transaction of 'length' beats starting from 'addr' and each beat writes 'size' bytes of data.

```
BfmWrite( con_handle
                , addr
                , data
                , size=4
                , length=1
                , rigor=0);
```

- con_handle: CON-FMC handle got from ‘confmc.conInit()’
- addr: starting address; should be aligned at the size of bytes to transfer in a beat[4].
- data: array containing data to be written and it is justified
- size: number of bytes at each beat; 1, 2, 4.
- length: number of beats in a burst[5]. The number of elements in ‘data’.
- rigor: check argument validity when it is 1

It returns on success. Otherwise returns negative value.

7.3 BfmRead()

‘BfmRead()’ generates a read burst transaction of ‘length’ beats starting from ‘addr’ and each beat reads ‘size’ bytes of data.

```
BfmRead ( con_handle
                , addr
                , data
                , size=4
                , length=1
               , rigor=0);
```

- con_handle: CON-FMC handle got from ‘confmc.conInit()’
- addr: starting address; should be aligned at the size of bytes to transfer in a beat.
- data: array to be contain data to be read and it is justified.
- size: number of bytes at each beat; 1, 2, 4.
- length: number of beats in a burst[6]. The number of elements in ‘data’.
- rigor: check argument validity when it is 1

It returns on success. Otherwise returns negative value.

7.4 Memory testing Python API

7.4.1 MemTestAddRAW

Memory test using address in read-after-write fashion and accesses are always 4-word size.

```
MemTestAddRAW( con_handle
```

[4] When ‘size’ is 4, ‘addr’ should be a multiple of 4.
[5] Since AMBA AXI3 only supports a burst of up to 16 beats, this API makes several bursts when ‘length’ is larger than 16.
[6] Since AMBA AXI3 only supports a burst of up to 16 beats, this API makes several bursts when ‘length’ is larger than 16.

```
                , saddr
                , depth
                , rigor=0)
```

- con_handle: CON-FMC handle got from ‘confmc.conInit()’
- saddr: starting address; should be aligned at the size of bytes to transfer in a beat.
- depth: length from the starting address to test.
- rigor: check argument validity when it is 1

Nothing returns.

7.4.2 MemTestAdd

Memory test using address in read-all-after-write-all fashion and accesses are always 4-word size.

```
MemTestAddRAW( con_handle
                , saddr
                , depth
                , rigor=0)
```

- con_handle: CON-FMC handle got from ‘confmc.conInit()’
- saddr: starting address; should be aligned at the size of bytes to transfer in a beat.
- depth: length from the starting address to test.
- rigor: check argument validity when it is 1

Nothing returns.

7.4.3 MemTestRAW

Memory test using random data in read-after-write fashion.

```
MemTest( con_handle
                , saddr
                , depth
                , size=4
                , rigor=0)
```

- con_handle: CON-FMC handle got from ‘confmc.conInit()’
- saddr: starting address; should be aligned at the size of bytes to transfer in a beat.
- depth: length from the starting address to test.
- size: the number of bytes for each beat
- rigor: check argument validity when it is 1

Nothing returns.

7.4.4 MemTest

Memory test using random data in read-all-after-write-all fashion and accesses are always 4-word size.

```
MemTest( con_handle
                    , saddr
                    , depth
                    , size=4
                    , rigor=0)
```

- con_handle: CON-FMC handle got from ‘confmc.conInit()’
- saddr: starting address; should be aligned at the size of bytes to transfer in a beat.
- depth: length from the starting address to test.
- size: the number of bytes for each beat.
- rigor: check argument validity when it is 1

Nothing returns.

8 Trouble shooting

8.1 USB error while calling BfmRead()

- Symptom: runtime error while calling BfmRead()

```
libusb: error [reap_for_handle] reap failed error -1 errno=14
libusb: error [handle_events] backend handle_events failed with error -1
libusb: error [sync_transfer_wait_for_completion] libusb_handle_events failed:
LIBUSB_ERROR_IO, cancelling transfer and retrying
libusb: warning [handle_timeout] async cancel failed -5 errno=22
```

- Reason: the ‘length’ argument might be too big.
- Solution: use ‘length’ argument less than USB allows such as 128, i.e., 128 words.

9 References

[1] ARM Limited, AMBA AXI Protocol Specification, Version 2.0, 2010
[2] Future Design Systems, Cypress EZ-USB FX3 GPIF-II Master Controller, FDS-TD-2018-04-002, 2018.
[3] Future Design Systems, CON-FMC API, FDS-TD-2018-04-004, 2018.
[4] Future Design Systems, PyCONFMC: CON-FMC Python Bining, FDS-TD-2019-12-002, 2019.

Wish list

- ☐

Revision history

- ☐ 2019.12.30: Python API added.
- ☐ 2018.12.05: Figure 2 updated and ‘MID[…]’ port is added, which is input.
- ☐ 2018.05.20: Packet format changed.
- ☐ 2018.05.10: Minor changes
- ☐ 2018.04.27: Started by Ando Ki (adki@future-ds.com)

– End of document –