GPCIe Development Kit User's Guide

for GPCIe DevKit version 0.8.1

Last modified at : Jan. 3, 2009

K & F Computing Research Co.
E-mail: support@kfcr.jp

Contents

1 Abstract

This document describes usage of the GPCIe Development Kit.

2 Contents of the Kit

GPCIe is a PCI Express IP core developed by K&F Computing Research Co. (hereafter KFCR). It provides a simple interface to the backend logic designed by the user. Combining GPCIe with the backend logic, the user can easily implement an interface to other PCI Express devices without detailed knowledge about PCI Express protocol.

The development kit includes the following three logic designs (all logic designs are provided as VHDL sources):

1. Host Interface Bridge (HIB) at the topmost layer of the GPCIe design hierarchy. It provides a simple and easy-to-use interface to the backend logic designed by the user.

2. GPCIe core, which implements the Transaction layer, the Data Link layer, and the PHY MAC sub layer defined by the PCI Express Specification, as well as the "Application layer" built over these three layers. PCI configuration registers and DMA controllers are built in this layer.

3. PHY, which implements the PHY PCS and PHY PMA sub layers using embedded Gigabit transceiver of Altera's FPGA devices.

The development kit also includes reference designs (i.e. a sample logic) to show usage of HIB, as well as its device driver and control library which run on Linux OS.

The kit contains the following items:

00readme	This file.
00readme-j	Japanese translation of this file.
00license	License agreement of this kit.
00license-j	Japanese translation of 00license.
doc/	User's guide and other documents.
hib.vhd	Host Interface Bridge (HIB), a wrapper for GPCIe.
gpcie.vhd	Main body of the GPCIe IP core, which includes logics for the Application layer, the Transaction layer, the Data Link layer, and the PHY MAC sublayer.
phy.vhd	the PHY PCS and PHY PMA sublayers for embedded transceivers of Altera's FPGA devices, namely, Arria GX and Stratix II GX.
ifpga_{agx,s2gx}{8,4,1}.vhd	Reference designs for PCI Express interface designed with HIB.
synth/	Files used for synthesis of the reference designs (.qpf, .qsf, .sdc).
Makefile	A makefile to generate hib.vhd, gpcie.vhd, and phy.vhd from VHDL source templates.
templates/	VHDL source templates.
scripts/	Utilities to install/uninstall softwares for HIB.
include/	Header files for HIB control library (for Linux).
lib/	HIB control library (for Linux).
driver/	A source code of the HIB device driver (for Linux).
hibutil/	A source code of the HIB control library (for Linux).
sample/	A sample program to show usage of HIB control library.

In section 3, we briefly overview GPCIe. In section 4, basic usage of GPCIe with HIB wrapper and with embedded transceiver is shown. In section 5, we give description for advanced usage, such as directly handle GPCIe IP core without HIB wrapper. Section 6 is devoted for detailed description of the VHDL entities.

Hereafter, all file locations are shown as relative path from the top directory of the development kit, `gpciepkg/`, unless otherwise specified.

3 Function Overview

3.1 Supported PCI Express device type

GPCIe operates as an Endpoint. It does not operate as a Switch nor a Rootport (of a Root Complex).

3.2 Supported FPGA devices

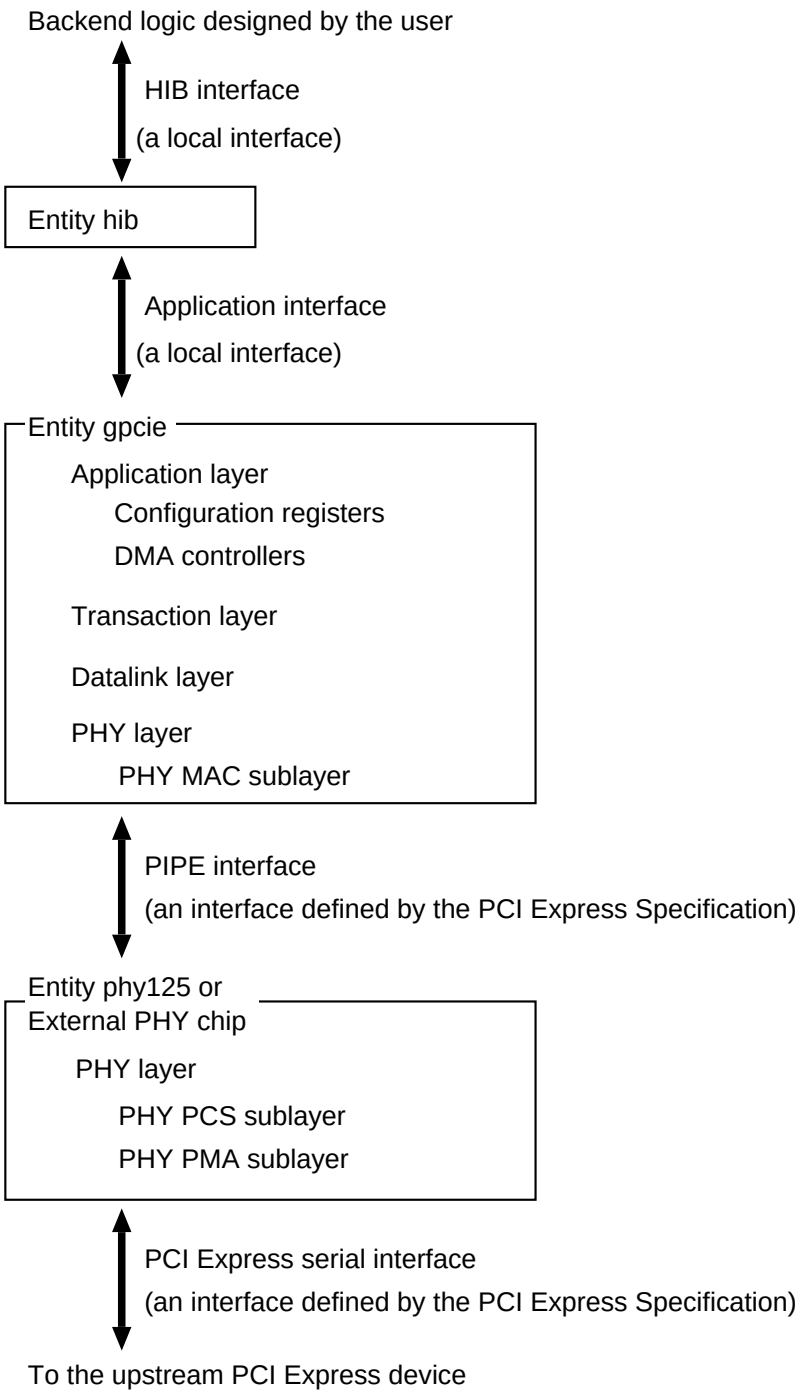
We have tested GPCIe on Altera's Arria GX and Stratix II GX. It is designed to operate also on Altera's FPGA devices without embedded transceivers ($e.g.$ Cyclone II, III, Stratix II), when combined with external PHY chips. However, we haven't tested such configuration yet.

List of supported FPGA devices:

PCIe revision	Link width	PIPE I/F	Arria GX	Stratix II GX
Gen1				
(2.5Gbps)	x8	128b@125MHz	**	**
	x4	64b@125MHz	**	**
	x1	16b@125MHz	**	**
Gen2				
(5.0Gbps)	x8	128b@250MHz	-	*
	x4	64b@250MHz	-	*
	x1	16b@250MHz	-	*

** : Supported * : Supported by GPCIe2 (not supported by GPCIe).

3.3 Structure of the Design



GPCIe consists of three entities, namely, `hib`, `gpcie`, and `phy125`.

Entity `hib` is located at the topmost layer, which provides a simple interface to the backend logic designed by the user.

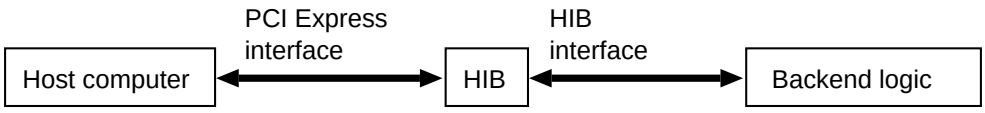
Entity `gpcie` implements the Transaction layer, the Data Link layer, and the PHY MAC sub layer defined by the PCI Express Specification, as well as the "Application layer" built over these three layers. PCI configuration registers and DMA controllers are built in this layer.

Entity `phy125` implements the PHY PCS and PHY PMA sub layers using embedded transceiver of Altera's FPGA devices. This entity is not used when external PHY chips are used. In such a case, the PIPE interface of entity `gpcie` is connected to the PHY chips.

4 Basic Usage

In this section, basic usage of GPCIe with HIB wrapper and embedded transceiver is shown. For usage without HIB, and usage with external PHY chips, see session 5.

In order to use GPCIe from a logic designed by the user (hereafter backend logic), create an instance of entity `hib`. The backend logic communicate with the host computer ($i.e.$ the upstream PCI Express device) via the HIB interface. HIB bridges data transfer via the PCI Express link and that via the HIB interface.



Data transfer from the host computer to HIB is performed by Program I/O (PIO) write. Data transfer from HIB to the host computer is performed by Direct Memory Access (DMA) write. Softwares to control these transfers are included in the development kit. Usage of the softwares will be described later.

Data transfer between HIB and the backend logic is performed using four signals (`hib_we`, `hib_data`, `backend_we`, `backend_data`) synchronized to a 125MHz clock `clk_out`. The backend writes to HIB using data bus `backend_data`, and its enable signal `backend_we`. HIB writes to the backend using data bus `hib_data`, and its enable signal `hib_we`.

Write from the backend to HIB :

```
clk_out       __~~__~~__~~__~~__~~__~~__~~__~~__~~
backend_we    ______~~~~~~~~~~~~~~~~______
backend_data        <d0><d1><d2><d3>
```

Write from HIB to the backend :

```
clk_out       __~~__~~__~~__~~__~~__~~__~~__~~__~~
hib_we        ______~~~~~~~~~~~~~~~~______
hib_data            <d0><d1><d2><d3>
```

The backend cannot insert any delay during a write burst from HIB. The backend must receive all data whenever `hib_we` is asserted.

HIB has an internal buffer to temporary store data written by the backend (By default, size of the buffer is set to 8k bytes). HIB sends contents of the buffer to the host computer, when HIB receives DMA write request from the host. The backend should watch over the buffer status. HIB itself does not check the buffer overflow.

DMA controllers and a PIO write controller reside in HIB. These controller can be accessed via HIB local registers mapped to the PCI Base Address Register 0 (BAR0) space.

The host computer performs PIO write onto the BAR2 space. HIB is designed so that it can achieve high transfer speed, if the page attribute of the BAR2 space region in use is set to write-combining mode (see the source code `templates/hibctl.vhd` for implementation detail). A procedure to set the page attribute will be described in section 4.2.4.

You can find a sample logic at `ifpga_{agx,s2gx}{8,4,1}.vhd`, which shows the actual usage of HIB. In the following, we describe how to synthesise the sample logic, and how to control it from the host computer.

4.1 Logic Synthesis

Use Altera's Quartus II for logic synthesis. Synthesis using other tools may be possible, but are not tested.

You can find Quartus Project Files (.qpf) and a Quartus Setting Files (.qsf) at:

```
synth/ifpga_{agx,s2gx}{8,4,1}.qpf
synth/ifpga_{agx,s2gx}{8,4,1}.qsf,
```

with which you can synthesize sample logics

```
ifpga_{agx,s2gx}{8,4,1}.vhd
```

to obtain an SRAM Object Files (.sof). These files can be used to configure KFCR's evaluation boards GPCIe-Eval-AGX8 and GPCIe2-Eval-S2GX8.

Note that only two VHDL source files:

```
hib.vhd
ifpga_{agx,s2gx}{8,4,1}.vhd
```

are used for the synthesis. Although HIB internally uses entity `gpcie` and `phy125`, `gpcie.vhd` and `phy.vhd` are not necessary. These are included into `hib.vhd`, just for user's convenience.

4.2 Softwares for HIB

The development kit includes softwares to control HIB from the host computer. The softwares consist of two components: HIB device driver and HIB control library. Installation procedure and usage of the softwares are described in this section.

Note : For now, the softwares are available only for Linux OS, and other platforms are currently not supported. However, this DOES NOT imply that design of HIB is Linux OS dependent. HIB is designed independent of any specific OS, and can be controlled from platforms other than Linux, if appropriate softwares are provided.

4.2.1 Software Installation

In order to install the softwares, run `scripts/install.csh` and follow its instruction.

```
kawai@localhost[1]>./scripts/install.csh
-----------------------------------------------
 Host Interface Bridge (HIB) software package
 installation program.
-----------------------------------------------

How many HIBs are you installing?: 1

Confirm your choice.
  number of HIBs you are installing : 1
Are they correct? (y/n): y

-------------------------------
Preparing for installation...
-------------------------------

...

gcc -O0 -g -I. -I../include -o hibtest hibtest.c hibutil.c -lm
gcc -O0 -g -I. -I../include -o lsgrape lsgrape.c hibutil.c -lm

done
```

Note that a complete source tree of the Linux kernel is required for successful installation.

4.2.2 Device Driver Configuration

Everytime the host computer is restarted, the HIB device driver need to be configured into the Linux kernel. In order to do this, change directory to `driver/`, and type `make installmodule` (You need the root permission).

```
[root@localhost driver]# make installmodule
./install0.csh
```

```
-- install module hibdrv --
hibdrv: 1 HIB(s) found.

rm -f /dev/hibdrv[0-9]

/sbin/insmod -f hibdrv.ko
mknod /dev/hibdrv0 c 253 0

chgrp wheel /dev/hibdrv0

chmod 666 /dev/hibdrv0
crw-rw-rw- 1 root wheel 253, 0 Jul  9 12:59 /dev/hibdrv0
-- done --
```

This should plug-in the HIB device driver `hibdrv` into the kernel. You can use a command `/sbin/lsmod` to check the driver status. Output of the command should have a line that contains a word `hibdrv`.

```
kawai@localhost[2]>lsmod
Module                  Size  Used by
hibdrv                 39608  0
...                    ...    ...
```

Once the device driver is properly configured, softwares running on the userland can access to HIB via the driver.

4.2.3 Functionality Test

A command `hibutil/hibtest` can be used to check functionality of the HIB installed into the system. Run `hibtest` without argument to show its usage:

```
kawai@localhost[3]>./hibtest
usage: ./hibtest <test_program_ID>
   0) show contents of config & HIB-local registers [devid]
   1) reset DMA and FIFO [devid]
   2) clear HIB-internal FIFO [devid]
   3) show DMA status [devid]
   4) read config register <addr> [devid]
   5) write config register <addr> <val> [devid]
   6) read HIB local registers mapped to BAR0 <addr> [devid]
   7) write HIB local registers mapped to BAR0 <addr> <val> [devid]
   8) read backend memory space mapped to BAR1 <addr> [devid]
   9) write backend memory space mapped to BAR1 <addr> <val> [devid]
  10) check DMA read/write function <size> <sendfunc> [devid] (host <-> HIB)
  11) measure DMA performance <sendfunc> [devid] (host <-> HIB)
  12) measure DMA write performance [devid] (host <- HIB; bypass internal FIFO)
  13) measure DMA read performance <sendfunc> [devid] (host -> HIB; bypass internal FIFO)
  14) reset backend [devid]
```

```
  15) raw PIO r/w & DMA r/w [devid]
  16) measure DMA performance with multiple HIBs <sendfunc>  <# of hibs>
      (host <-> HIBs internal FIFO)
  17) measure DMA write performance with multiple HIBs <# of hibs> [devid offset]
      (host <- HIBs; bypass internal FIFO)
  18) measure DMA read performance with multiple HIBs <sendfunc> <# of hibs> [devid offset]
      (host -> HIBs; bypass internal FIFO)
  19) erase configuration ROM (EPCS64) [devid]
  20) write .rpd to configuration ROM (EPCS64) <rpd-file> [devid]
  21) read configuration ROM ID (0x10:EPCS1 0x12:EPCS4 0x14:EPCS16 0x16:EPCS64) [devid]
  22) set pipeline clock frequency to (PCI-X_bus_freq * N / M) <N> <M> [devid]
```

Run `hibtest 0` to show contents of the PCI configuration registers of the HIB:

```
kawai@localhost[4]>./hibtest 0
## hib0:
protocol : PCIe
link width negotiated : x8
            supported : x8
link speed negotiated : 2.5 Gb/s
           supported  : 2.5 Gb/s
max payload size negotiated : 128 byte
                 supported  : 256 byte
max read request size : 256 byte

configuration register:
0x00000000: 0x0e701b1a
0x00000004: 0x00100007
0x00000008: 0xff000001
0x0000000c: 0x00000008
0x00000010: 0xdf608008 0xdf608000
0x00000014: 0xdf610008 0xdf610000
0x00000018: 0xdf600008 0xdf600000
0x0000001c: 0x00000000 0x00000000
0x00000020: 0x00000000
0x00000024: 0x00000000
0x00000028: 0x00000000
0x0000002c: 0x0e701b1a
0x00000030: 0x00000000
0x00000034: 0x00000080
0x00000038: 0x00000000
0x0000003c: 0x000000ff
PCI Express Capability Register:
0x00000080: 0x00110010
0x00000084: 0x00000001
0x00000088: 0x00001000
0x0000008c: 0x00000481
```

```
0x00000090: 0x00810000
```

Run `hibtest 10 10 1` to test loopback transfer. This will send 10 * 8 byte data from the host computer by PIO write transfer. The HIB receives the data, and then send it back to the host computer by DMA write transfer. The host computer compares the data transmitted and received, and print `OK` if these are completely matched, print `NG` otherwise.

```
kawai@localhost[5]>./hibtest 10 10 1

# check hib[0] DMA read/write (host <-> HIB internal FIFO)

size 10

# hib[0] PIO write, and then DMA write (host <-> HIB internal FIFO)
clear DMA buf...
DMA read size: 10 words (80 bytes)
will dmar...

rbuf[0000]: 0x1111111111111111  wbuf[0000]: 0x1111111111111111
rbuf[0001]: 0x2222222222222222  wbuf[0001]: 0x2222222222222222
rbuf[0002]: 0x3333333333333333  wbuf[0002]: 0x3333333333333333
rbuf[0003]: 0x4444444444444444  wbuf[0003]: 0x4444444444444444
rbuf[0004]: 0x5555555555555555  wbuf[0004]: 0x5555555555555555
rbuf[0005]: 0x6666666666666666  wbuf[0005]: 0x6666666666666666
rbuf[0006]: 0x123456789abc0006  wbuf[0006]: 0x123456789abc0006
rbuf[0007]: 0x123456789abc0007  wbuf[0007]: 0x123456789abc0007
rbuf[0008]: 0x123456789abc0008  wbuf[0008]: 0x123456789abc0008
rbuf[0009]: 0x123456789abc0009  wbuf[0009]: 0x123456789abc0009
---- transfer size reached ----
rbuf[0010]: 0x123456789abc000a  wbuf[0010]: 0xfedcba987654000a
rbuf[0011]: 0x123456789abc000b  wbuf[0011]: 0xfedcba987654000b
done
 10 words (80 bytes).
OK
```

Run hibtest 12 to measure performance of the DMA write transfer (write from HIB to the host).

```
kawai@localhost[6]>./hibtest 12

# hib[0] DMA write (host <- HIB)
size: 1024 DMA write: 1.562367 sec  512.043597 MB/s
size: 2048 DMA write: 1.101087 sec  726.554697 MB/s
size: 4096 DMA write: 0.857353 sec  933.104598 MB/s
size: 8192 DMA write: 0.739353 sec  1082.027209 MB/s
size: 16384 DMA write: 0.680854 sec  1174.995203 MB/s
size: 32768 DMA write: 0.651100 sec  1228.690060 MB/s
```

Run hibtest 13 1 to measure performance of the PIO write transfer (write from the host to HIB).

```
kawai@localhost[7]>./hibtest 13 1

# hib[0] PIO write (host -> HIB)
size: 64 PIO write: 2.037641 sec  392.610858 MB/s
size: 128 PIO write: 1.233335 sec  648.647763 MB/s
size: 256 PIO write: 0.822831 sec  972.253211 MB/s
size: 512 PIO write: 0.639186 sec  1251.591587 MB/s
size: 1024 PIO write: 0.620417 sec  1289.455073 MB/s
size: 2048 PIO write: 0.620460 sec  1289.365885 MB/s
size: 4096 PIO write: 0.620398 sec  1289.495211 MB/s
size: 8192 PIO write: 0.620425 sec  1289.438721 MB/s
size: 16384 PIO write: 0.620416 sec  1289.457550 MB/s
```

Usage of hibtest not shown above, see the source code `hibutil/hibtest.c`.

4.2.4 MTRR Set up

The host computer performs PIO write via the BAR2 space. HIB is designed so that it can achieve high transfer speed, if the page attribute of the BAR2 space region in use is set to the *write-combining* mode. If the mode is not set, the speed would be reduced to 20% or lower of the peak.

The mode of the BAR2 space can be set via MTRR (memory type range register) of the host computer. In order to set the mode to write-combining, run `scripts/setmtrr.csh` (You need the root permission). The script searches for a free (*i.e.* not used by other device) MTRR, and using that MTRR, set the BAR2 space to the write-combining mode.

```
[root@localhost driver]# ./setmtrr.csh

Searching for HIB(s)... Found 0 PCI-X HIB(s). Found 1 PCIe HIB(s).
Found 1 HIB(s) in total.

Trying to set 1 MTRR(s)...
    echo "base=0xdf600000 size=0x1000 type=write-combining" > /proc/mtrr
Done.

current setting of MTRRs:
reg00: base=0x00000000 (   0MB), size=2048MB: write-back, count=1
reg01: base=0x80000000 (2048MB), size=1024MB: write-back, count=1
reg02: base=0x100000000 (4096MB), size=200704MB: write-back, count=1
reg03: base=0x200000000 (8192MB), size=1024MB: write-back, count=1
reg04: base=0xdf600000 (3574MB), size=   4KB: write-combining, count=1
```

The output should include a line containing "base=0xAAAAAAAA (XXXXMB), size = 4kB: write-combining", where AAAAAAAA denote the start address of the BAR2 space of HIB. The value can be checked by `hibtest 4 18`:

```
kawai@localhost[8]>../hibutil/hibtest 4 18
hib[0] config 0x00000018: 0xdf600008
```

The BAR2 space may not be set up to the write-combining mode, if, for example, all 8 existing MTRRs are already used by other devices, or, the total size of the main memory exceeds 4GB and the chipset cannot handle I/O remapping. Depending on the chipset, this problem may be avoided ($e.g.$ by setting I/O remapping of the main memory to address higher than 4GB, or setting memory hole granularity to a larger value). Refer to the manual of the chipset or the mother board.

Running `hibtest 13 1` before and after MTRR set up, you can see improvement of the PIO write performance:

Before MTRR set up (x8 link) :

```
kawai@localhost[9]>./hibtest 13 1
# hib[0] PIO write (host -> HIB)
size: 64 PIO write: 7.319836 sec  109.292068 MB/s
size: 128 PIO write: 6.857664 sec  116.657799 MB/s
size: 256 PIO write: 6.597888 sec  121.250922 MB/s
size: 512 PIO write: 6.458101 sec  123.875423 MB/s
size: 1024 PIO write: 6.404411 sec  124.913905 MB/s
size: 2048 PIO write: 6.397210 sec  125.054514 MB/s
size: 4096 PIO write: 6.387041 sec  125.253617 MB/s
size: 8192 PIO write: 6.390173 sec  125.192230 MB/s
size: 16384 PIO write: 6.384816 sec  125.297269 MB/s
```

After MTRR set up (x8 link) :

```
kawai@localhost[10]>./hibtest 13 1
# hib[0] PIO write (host -> HIB)
size: 64 PIO write: 2.037641 sec  392.610858 MB/s
size: 128 PIO write: 1.233335 sec  648.647763 MB/s
size: 256 PIO write: 0.822831 sec  972.253211 MB/s
size: 512 PIO write: 0.639186 sec  1251.591587 MB/s
size: 1024 PIO write: 0.620417 sec  1289.455073 MB/s
size: 2048 PIO write: 0.620460 sec  1289.365885 MB/s
size: 4096 PIO write: 0.620398 sec  1289.495211 MB/s
size: 8192 PIO write: 0.620425 sec  1289.438721 MB/s
size: 16384 PIO write: 0.620416 sec  1289.457550 MB/s
```

4.2.5 HIB Control Library Usage

HIB control library provides an API to handle data transfer between the host computer and HIB. In order to use the library, include a header file `include/hibutil.h` into your own source code (written in C or C++), and link `lib/libhib.a`.

Descriptions for substantial functions provided by the library are given below. For usages of other functions, look inside the source code `hibutil/hibutil.c`.

Hib* hib_openMC(int devid) Obtains access permission of a HIB that has a device ID `devid`. If the HIB is already obtained by another process, this function blocks. The device ID `devid` is a small integer uniquely assigned to each HIB. When n HIBs are installed in the system, one of device IDs from 0 to $n-1$ is assigned to each of them.

hib_openMC(void) returns a pointer to a variable of type `Hib`. The variable stores information necessary to manage the HIB device opened. Some API functions require the pointer as their argument (*cf.* `hib_dmawMC`).

void hib_closeMC(int devid) Release access permission of a HIB that has a device ID `devid`, so that other process can obtain it.

void hib_piowMC(int devid, int size, UINT64 *buf) writes data stored in the main memory to a HIB that has a device ID `devid`. Size of the data is given by `size` (in 8-byte unit), and the start address is given by `buf`.

For the buffer pointed by `buf`, you can specify a memory region allocated by a usual method, such as an array of type `UINT64` statically allocated, or a region dynamically allocated by `malloc()`.

void hib_start_dmawMC(int devid, int size, UINT64 *buf) sends a DMA-write request to a HIB that has a device ID `devid`, which will kick off a data transfer from the HIB to the host. Size of the data is given by `size` (in 8-byte unit), and the address of the receiving buffer is given by `buf`.

Note that you CANNOT specify an arbitrary memory region as the receiving buffer. Only a memory region pointed to by `h->dmaw_buf`, or `h->dmaw_buf+offset` can be used as `buf`. Here, `h` denotes a pointer to a variable of type `Hib` returned by `hib_openMC()`, and the value of `offset+size` should not exceed 32k byte. The address pointed by `h->dmaw_buf` is a continuous memory region allocated inside the Linux kernel space, which is mapped to the userland. In order to store the data received from the HIB into a buffer in the userland, such as a statically allocated array, or a region dynamically allocated by `malloc()`, you need to copy the data from `h->dmaw_buf` to the buffer.

int hib_finish_dmawMC(int devid) waits for completion of a DMA write transfer started by `hib_start_dmawMC()`.

UINT32 hib_config_readMC(int devid, UINT32 addr) reads the value of the PCI Configuration Register address `addr` of a HIB that has a device ID `devid`.

void hib_config_writeMC(int devid, UINT32 addr, UINT32 value) writes a `value` to the PCI Configuration Register address `address` of a HIB that has a device ID `devid`.

UINT32 hib_mem_readMC(int devid, UINT32 addr) reads the value of the HIB Local Register address `addr` of a HIB that has a device ID `devid`. See `templates/hibctl.vhd` for the address map of the Local Register.

void hib_mem_writeMC(int devid, UINT32 addr, UINT32 value) writes a `value` to the HIB Local Register address `address` of a HIB that has a device ID `devid`.

An Example Program using the HIB Control Library :

You can find an example of application program at `sample/loopback.c`, which internally uses the HIB control library. It performs a simple loopback transfer: It transmit 10 * 8 byte data from the host computer. HIB receives the data, and then send it back to the host computer. The host computer compares the data transmitted and received, and report the result.

```
kawai@localhost[9]>./loopback
0x0000  sent : 0x123456789abc0000    received : 0x123456789abc0000  OK
0x0001  sent : 0x123456789abc0001    received : 0x123456789abc0001  OK
0x0002  sent : 0x123456789abc0002    received : 0x123456789abc0002  OK
0x0003  sent : 0x123456789abc0003    received : 0x123456789abc0003  OK
0x0004  sent : 0x123456789abc0004    received : 0x123456789abc0004  OK
0x0005  sent : 0x123456789abc0005    received : 0x123456789abc0005  OK
0x0006  sent : 0x123456789abc0006    received : 0x123456789abc0006  OK
0x0007  sent : 0x123456789abc0007    received : 0x123456789abc0007  OK
0x0008  sent : 0x123456789abc0008    received : 0x123456789abc0008  OK
0x0009  sent : 0x123456789abc0009    received : 0x123456789abc0009  OK
```

5 Advanced Usage

In this section, advanced usages of GPCIe are described. In section 5.1, a procedure to apply modifications to the source code of the GPCIe logic design is described. Section 5.2 suggest an alternative design of GPCIe to improve DMA write performance. In section 5.3, necessary modifications to the HIB design are described, in order to use external PHY chips instead of the embedded transceivers. In section 5.4, a method to directly ($i.e.$, without HIB wrapper) control GPCIe IP core is given.

5.1 Source Code Updation

Source code of GPCIe is split into multiple VHDL files in `templates/` directory. For user's convenience, all files which entity `hib` relies on are packed into a single file `hib.vhd`. Similarly, files necessary for entity `gpcie` and `phy125` are packed into `gpcie.vhd` and `phy.vhd`, respectively.

When you modified the source code, change directory to `gpciepkg` and run `make` for updation, so that the modifications are reflected to `hib.vhd`, `gpcie.vhd`, and `phy.vhd`.

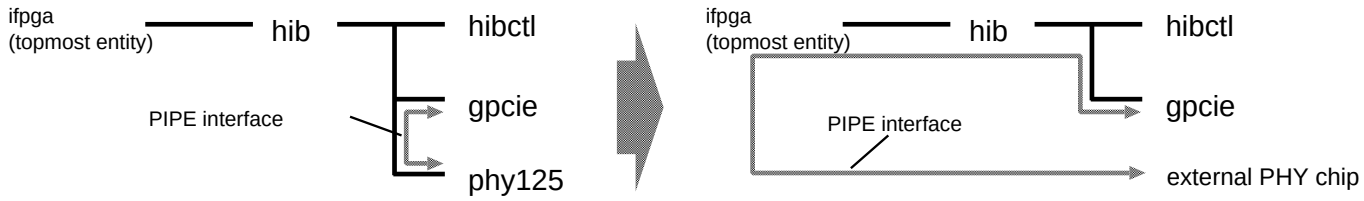
5.2 GPCIe with Faster DMA Write

Source code of GPCIe in `templates/` directory includes highly optimized modules for DMA write transfer. However, these modules are not used by default, since these modules require special care for timing, in order to satisfy timing constraints on Arria GX. In order to activate these modules, change directory to

gpciepkg and run `make fast`. This will generate `hib0.vhd` and `gpcie0.vhd`. These are the optimized version of `hib.vhd` and `gpcie0.vhd`, respectively. At the best case, the speed would be improved about 20% for DMA write transfer. The speeds of PIO write transfer and DMA read transfer are not affected.

5.3 Usage with External PHY Chip

In order to use external PHY chips instead of the embedded transceivers, you need to modify entity `hib` defined in `templates/hibtop.vhd`.



Entity `hib` internally uses instances of three entities: `hibctl`, `gpcie`, and `phy125`. You need two modifications for these instances. First, remove the instance of `phy125`, and also remove the PIPE interface connection between `phy125` and `gpcie`. The instance `phy125` is a wrapper for the embedded transceivers that implements PHY PCS and PHY PMA sub layers. These layers are realized by the external PHY chips, and thus `phy125` is no longer necessary.

Next, connect the PIPE interface of the instance `gpcie` to that of the PHY chips. To do this, you need to hardwire I/O pins of the PHY chips and the FPGA device. Then assign the I/O pins to the ports of the topmost entity, and connect these ports to corresponding ports of the `hib` instance, as well as those of the `gpcie` instance.

5.4 Direct Handling of GPCIe IP Core

Although the HIB wrapper provides a simple interface to the backend logic, it cannot take full advantage of GPCIe functionalities. For example, in order to:

- implement PIO read/write transfer with address, byte enable, and wait control,
- assign all Base Address Space (BAR0..5) to arbitrary purpose, or,
- use multiple DMA channels (8 channels at max),

you need to handle GPCIe IP core directly (*i.e.,* without HIB) from the backend logic. For this purpose, instantiate entity `gpcie` (which is defined in `gpcie.vhd`) in your design. Then, in order to use the embedded transceivers, instantiate entity `phy125` (defined in `phy.vhd`) also, and connect the PIPE interface of these two instances each other.

6 Details of the VHDL Entities

VHDL entities `hib`, `gpcie`, and `phy125` have various generic parameters and I/O ports. In the following, description for their substantial generic parameters and all I/O ports are given.

6.1 Details of Entity hib

6.1.1 Header Part

For successful compilation, you need to use a package gpciepkg, as well as some other packages defined in standard libraries. The package gpciepkg is defined in gpciepkg.vhd.

Example :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.gpciepkg.all;
```

6.1.2 GENERIC Declaration

Parameter name :	DEVICE
Type :	string
Default value :	"Arria GX"
Function :	Targeting FPGA device. Should be set to "Arria GX" or "Stratix II GX".

Parameter name :	NLANE
Type :	integer
Default value :	–
Function :	Link width of the PCI Express link. Should be set to 1, 4 or 8.

Parameter name :	PIOWBUF_DEPTH
Type :	integer
Default value :	8
Function :	Depth of the PIO write buffer. The default value 8 denotes 256 ($= 2^8$) words, that is, 512, 2048, and 4096 bytes for x1, x4, and x8 link, respectively. User rarely need to modify this value, although it affects the performance of PIO write transfer.

Parameter name :	TXBUF_DEPTH
Type :	integer
Default value :	10
Function :	Depth of the backend_data receiving buffer. Default value 10 denotes 1024 ($= 2^{10}$) words, that is, 2048, 8192, and 16384 bytes for x1, x4, and x8 link, respectively. In order to transfer data exceeding this size in a single DMA write, the backend should implement a flow-control logic to avoid buffer overflow.

Parameter name :	USE_CLK32
Type :	integer
Default value :	1
Function :	Should be set to 1 whenever possible. You may set this value to 0 if you cannot supply clk32 input. Then HIB try to boot without using clk32, at the risk of malfunction.

6.1.3 PORT Declaration

Port name :	phy_linkup
Type :	std_logic
Direction :	out
Function :	Asserted when the PCIe link training in the PHY layer is successfully completed.

Port name :	dl_linkup
Type :	std_logic
Direction :	out
Function :	Asserted when the PCIe link initialization in the Data Link layer is successfully completed.

Port name :	clk100_ext
Type :	std_logic
Direction :	in
Function :	A 100MHz differential input used as a reference clock of the Gigabit transceivers.

Port name :	clk32
Type :	std_logic
Direction :	in
Function :	A clock input used to generate timing for power on reset signals and transceiver calibration. The clock frequency can be any value in the range of 10MHz-125MHz.

Port name :	mperst
Type :	std_logic
Direction :	in
Function :	An active low reset signal.

Port name :	rx_in
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	in
Function :	Input from the PCI Express high-speed serial receiver port.

Port name :	tx_out
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	out
Function :	Output to the PCI Express high-speed serial transmitter port.

Port name :	clk_out
Type :	std_logic
Direction :	out
Function :	A 125MHz clock output generated in the PHY PCS layer based on `clk100_ext` input. All parallel signals inside HIB are synchronized to this clock.

Port name :	wake
Type :	std_logic
Direction :	out
Function :	Not used.

Port name :	hib_we
Type :	std_logic
Direction :	out
Function :	Write enable for `hib_data`, which is driven by HIB.

Port name :	hib_data
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	out
Function :	Data output from HIB to the backend logic.

Port name :	backend_we
Type :	std_logic
Direction :	in
Function :	Write enable for `backend_data`, which is driven by the backend logic.

Port name :	backend_data
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	in
Function :	Data input from the backend logic to HIB.

Port name :	reset_backend
Type :	std_logic
Direction :	out
Function :	Active high reset output to the backend logic.

Port name :	board_info
Type :	std_logic_vector(31 downto 0)
Direction :	in
Function :	Initial value of a mailbox register `board_info`. This register can be read/written by the host computer. It can be used by the backend logic for an arbitrary purpose.

6.2 Details of Entity `gpcie`

6.2.1 Header Part

For successful compilation, you need to `use` a package `gpciepkg`, as well as some other packages defined in standard libraries. The package `gpciepkg` is defined in `gpciepkg.vhd`.

Example :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.gpciepkg.all;
```

6.2.2 GENERIC Declaration

The default values of generic parameters of entity gpcie are optimized for HIB. You may overwrite them depending on your design requirement.

Parameter name :	NLANE
Type :	integer
Default value :	8
Function :	Link width of the PCI Express link. Should be set to 1, 4 or 8.

Parameter name :	NDMACH
Type :	integer
Default value :	2
Function :	The number of DMA channels to be implemented. At maximum, eight channels can be implemented. However, operations only with up to two channels are tested so far.

Parameter name :	MAX_READ_REQ_SIZE
Type :	integer
Default value :	256
Function :	The maximum read request size in byte.

Parameter name :	MAX_PAYLOAD
Type :	integer
Default value :	256
Function :	The maximum payload size supported (in byte unit). The maximum payload size actually used may be smaller than the value set here. The actual size is determined through negotiation with the upstream device at link initialization phase.

Parameter name :	CA_PH_VC0_INIT
Type :	integer
Default value :	16
Function :	Depth of the Rx Flow Control buffer (posted, header). The value is specified by the number of the Transaction-Layer packets (TLPs) which can be stored.

Parameter name :	CA_PD_VC0_INIT
Type :	integer
Default value :	64
Function :	Depth of the Rx Flow Control buffer (posted, data) in 16-byte unit.

Parameter name :	CA_NPH_VC0_INIT
Type :	integer
Default value :	2
Function :	Depth of the Rx Flow Control buffer (non-posted, header). The value is specified by the number of the TLPs which can be stored.

Parameter name :	CA_NPD_VC0_INIT
Type :	integer
Default value :	16
Function :	Depth of the Rx Flow Control buffer (non-posted, data) in 16-byte unit.

Parameter name :	CA_CH_VC0_INIT
Type :	integer
Default value :	2
Function :	Depth of the Rx Flow Control buffer (completion, header). The value is specified by the number of the TLPs which can be stored.

Parameter name :	CA_CD_VC0_INIT
Type :	integer
Default value :	16
Function :	Depth of the Rx Flow Control buffer (completion, data) in 16-byte unit.

Parameter name :	CL_PH_VC0_INIT
Type :	integer
Default value :	16
Function :	Depth of the Tx Flow Control buffer (posted, header). The value is specified by the number of the TLPs which can be stored.

Parameter name :	CL_PD_VC0_INIT
Type :	integer
Default value :	64
Function :	Depth of the Tx Flow Control buffer (posted, data) in 16-byte unit.

Parameter name :	CL_NPH_VC0_INIT
Type :	integer
Default value :	2
Function :	Depth of the Tx Flow Control buffer (non-posted, header). The value is specified by the number of the TLPs which can be stored.

Parameter name :	CL_NPD_VC0_INIT
Type :	integer
Default value :	16
Function :	Depth of the Tx Flow Control buffer (non-posted, data) in 16-byte unit.

Parameter name :	CL_CH_VC0_INIT
Type :	integer
Default value :	2
Function :	Depth of the Tx Flow Control buffer (completion, header). The value is specified by the number of the TLPs which can be stored.

Parameter name :	CL_CD_VC0_INIT
Type :	integer
Default value :	16
Function :	Depth of the Tx Flow Control buffer (completion, data) in 16-byte unit.

Parameter name :	CFG_VENDOR_ID_INIT
Type :	std_logic_vector(15 downto 0)
Default value :	x"1b1a"
Function :	Vendor ID of KFCR. Do not modify (see the license agreement in section 7).

Parameter name :	CFG_DEVICE_ID_INIT
Type :	std_logic_vector(15 downto 0)
Default value :	x"0e70"
Function :	Device ID. Default value 0E70h is the one KFCR assigned to HIB.

Parameter name :	CFG_REVISION_ID_INIT
Type :	std_logic_vector(7 downto 0)
Default value :	x"01"
Function :	Revision ID.

Parameter name :	CFG_CLASS_CODE_INIT
Type :	std_logic_vector(23 downto 0)
Default value :	x"ff0000"
Function :	PCI class code.

Parameter name :	CFG_BAR0_INIT
Type :	std_logic_vector(31 downto 0)
Default value :	x"ffff8008"
Function :	Initial value of PCI Base Address Register0 (BAR0). The default value x"ffff8008" denotes 32kbyte, prefetchable, 32-bit address, memory space.

Parameter name :	CFG_BAR1_INIT
Type :	std_logic_vector(31 downto 0)
Default value :	x"fffff008"
Function :	Initial value of PCI Base Address Register1 (BAR1). The default value x"fffff008" denotes 4kbyte, prefetchable, 32-bit address, memory space.

Parameter name :	CFG_BAR2_INIT
Type :	std_logic_vector(31 downto 0)
Default value :	x"ffff8008"
Function :	Initial value of PCI Base Address Register2 (BAR2). The default value x"ffff8008" denotes 32kbyte, prefetchable, 32-bit address, memory space.

Parameter name :	CFG_BAR3_INIT
Type :	std_logic_vector(31 downto 0)
Default value :	x"00000000"
Function :	Initial value of PCI Base Address Register3 (BAR3). The default value x"00000000" denotes this register is not used.

Parameter name :	CFG_BAR4_INIT
Type :	std_logic_vector(31 downto 0)
Default value :	x"00000000"
Function :	Initial value of PCI Base Address Register4 (BAR4). The default value x"00000000" denotes this register is not used.

Parameter name :	CFG_BAR5_INIT
Type :	std_logic_vector(31 downto 0)
Default value :	x"00000000"
Function :	Initial value of PCI Base Address Register5 (BAR5). The default value x"00000000" denotes this register is not used.

Parameter name :	CFG_BAR_ROM_INIT
Type :	std_logic_vector(31 downto 0)
Default value :	x"00000000"
Function :	Initial value of PCI Expansion ROM Base Address. The default value x"00000000" denotes this register is not used.

Parameter name :	CFG_SUB_VENDOR_ID_INIT
Type :	std_logic_vector(15 downto 0)
Default value :	x"1b1a"
Function :	Sub vendor ID.

Parameter name :	CFG_SUB_DEVICE_ID_INIT
Type :	std_logic_vector(15 downto 0)
Default value :	x"0e70"
Function :	Sub device ID.

Parameter name :	CFG_INT_PIN_INIT
Type :	std_logic_vector(7 downto 0)
Default value :	x"00"
Function :	Interrupt pin in use. Should be set to 0, since GPCIe currently does not support interrupt signal.

6.2.3 PORT Declaration

Port name :	phy_linkup
Type :	std_logic
Direction :	out
Function :	Asserted when the PCIe link training in the PHY layer is successfully completed.

Port name :	dl_linkup
Type :	std_logic
Direction :	out
Function :	Asserted when the PCIe link initialization in the Data Link layer is successfully completed.

Port name :	clk
Type :	std_logic
Direction :	in
Function :	A 125MHz clock input supplied from the PHY PCS layer. All I/O ports including PIPE interface are synchronized to this clock.

Port name :	rstn
Type :	std_logic
Direction :	in
Function :	An active low reset signal.

Port name :	phystatus
Type :	std_logic
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	powerdown
Type :	std_logic_vector(1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	txdetectrx
Type :	std_logic
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	txdata
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	txdatak
Type :	std_logic_vector(NLANE*2-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	txelecidle
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	txcompl
Type :	std_logic_vector(NLANE-1 downto 0);
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxpolarity
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxdata
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxdatak
Type :	std_logic_vector(NLANE*2-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxvalid
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxelecidle
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxstatus
Type :	std_logic_vector(NLANE*3-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

The Application Interface (as a slave device)

Port name :	slv_readreq
Type :	std_logic
Direction :	out
Function :	Read request. The read will start right at the clock cycle when `slv_accept` is asserted.

Port name :	slv_writereq
Type :	std_logic
Direction :	out
Function :	Write request. The write will start right at the clock cycle when `slv_accept` is asserted.

Port name :	slv_accept
Type :	std_logic_vector(downto 0)
Direction :	in
Function :	Accept for read/write request.

Port name :	slv_read
Type :	std_logic
Direction :	out
Function :	When this is asserted, the backend logic should supply data to `slv_datain` in the next clock cycle.

Port name :	slv_write
Type :	std_logic
Direction :	out
Function :	Indicates data is present on `slv_dataout`.

Port name :	slv_bar
Type :	std_logic_vector(6 downto 0)
Direction :	out
Function :	Base address space from/to which current transaction is reading/writing.

Port name :	slv_addr
Type :	std_logic_vector(63 downto 0)
Direction :	out
Function :	Local address from/to which current transaction is reading/writing.

Port name :	slv_bytevalid
Type :	std_logic_vector(NLANE*2-1 downto 0)
Direction :	out
Function :	Byte enables for `slv_dataout`. Valid only for write transaction.

Port name :	slv_bytecount
Type :	std_logic_vector(11 downto 0)
Direction :	out
Function :	Remaining byte count for current transaction.

Port name :	slv_dataout
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	out
Function :	Data output from GPCIe.

Port name :	slv_datain
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	in
Function :	Data input to GPCIe.

The Application Interface (as a master device)

Port name :	ms_wrchannel
Type :	std_logic_vector(NDMACH-1 downto 0)
Direction :	out
Function :	The DMA channel currently occupying the data path for DMA write, `ms_wrdata`.

Port name :	ms_write
Type :	std_logic
Direction :	out
Function :	When this is asserted, the backend logic should supply data to `ms_wrdata` in the next clock cycle. Used for DMA write transfer.

Port name :	ms_wraddr
Type :	std_logic_vector(31 downto 0)
Direction :	out
Function :	Local address which current DMA write transaction is reading from.

Port name :	ms_wrdata
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	in
Function :	Data input from the backend logic to GPCIe. Used for DMA write transfer.

Port name :	ms_rdchannel
Type :	std_logic_vector(NDMACH-1 downto 0)
Direction :	out
Function :	The DMA channel currently occupying the data path for DMA read, `ms_rddata`.

Port name :	ms_read
Type :	std_logic
Direction :	out
Function :	Indicates data is present on `ms_rddata`. Used for DMA read transfer.

Port name :	ms_rdaddr
Type :	std_logic_vector(31 downto 0)
Direction :	out
Function :	Local address which current DMA read transaction is writing to.

Port name :	ms_rddata
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	out
Function :	Data output from GPCIe to the backend logic. Used for DMA read transaction.

The Application Interface (as a DMA controller)

An independent set of interface is provided for each DMA(n) channel, where n is a channel ID in `0..NDMACH-1`. For example, `dma_control` signal for the n-th channel can be accessed via `dma_control(`n`)(6 downto 0)`. The two-dimensional array types used for the definition of these signals, such as `each7b` and `each16b` are defined in a package `gpciepkg`.

Port name :	dma_control
Type :	each7b(NDMACH-1 downto 0)
Direction :	in
Function :	DMA control registers
	dma_control(n)(0) : Write enable for `dma_paddrlow_in(`n`)`
	dma_control(n)(1) : Write enable for `dma_paddrhight_in(`n`)`
	dma_control(n)(2) : Write enable for `dma_laddr_in(`n`)`
	dma_control(n)(3) : Write enable for `dma_size_in(`n`)` Start a DMA transfer when a '1' is written.
	dma_control(n)(4) : Write enable for `dma_param_in(`n`)`
	dma_control(n)(6) : Stop currently running DMA transfer when a '1' is written.

Port name :	dma_param
Type :	each16b(NDMACH-1 downto 0)
Direction :	in
Function :	DMA parameter registers
	dma_param(n)(7 downto 0) : Not used.
	dma_param(n)(8) : Direction of the transfer.
	0 : read from the host computer.
	1 : write to the host computer.
	dma_param(n)(15 downto 9) : Not used.

Port name :	dma_status
Type :	each4b(NDMACH-1 downto 0)
Direction :	out
Function :	DMA status registers
	dma_status(n)(2 downto 0) : Not used.
	dma_status(n)(3) : A flag to indicate completion of a DMA transfer.
	0 : a transfer is in progress.
	1 : no transfer is in progress.

Port name :	dma_fifocnt
Type :	each13b(NDMACH-1 downto 0)
Direction :	in
Function :	Byte count of a DMA transfer.
	For DMA write : The number of bytes the backend logic can supply to GPCIe.
	For DMA read : The number of bytes the backend logic can receive from GPCIe.

Port name :	dma_paddrlow_in
Type :	each32b(NDMACH-1 downto 0)
Direction :	in
Function :	Lower 32-bit of PCI address at which a DMA transfer starts.

Port name :	dma_paddrhigh_in
Type :	each32b(NDMACH-1 downto 0)
Direction :	in
Function :	Higher 32-bit of PCI address at which a DMA transfer starts.

Port name :	dma_laddr_in
Type :	each32b(NDMACH-1 downto 0)
Direction :	in
Function :	Local address at which a DMA transfer starts.

Port name :	dma_size_in
Type :	each32b(NDMACH-1 downto 0)
Direction :	in
Function :	Size of a DMA transfer (in byte).

Port name :	dma_paddrlow_out
Type :	each32b(NDMACH-1 downto 0)
Direction :	out
Function :	Lower 32-bit of PCI address at which a DMA transfer is in progress.

Port name :	dma_paddrhigh_out
Type :	each32b(NDMACH-1 downto 0)
Direction :	out
Function :	Higher 32-bit of PCI address at which a DMA transfer is in progress.

Port name :	dma_laddr_out
Type :	each32b(NDMACH-1 downto 0)
Direction :	out
Function :	Local address at which a DMA transfer is in progress.

Port name :	dma_size_out
Type :	each32b(NDMACH-1 downto 0)
Direction :	out
Function :	Remaining byte count of a DMA transfer in progress.

6.3 Details of Entity phy125

6.3.1 Header Part

Example :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

6.3.2 GENERIC Declaration

Parameter name :	DEVICE
Type :	string
Default value :	"Arria GX"
Function :	Targeting FPGA device. Should be set to "Arria GX" or "Stratix II GX".

Parameter name :	NLANE
Type :	integer
Default value :	–
Function :	Link width of the PCI Express link. Should be set to 1, 4 or 8.

Parameter name :	USE_CLK32
Type :	integer
Default value :	1
Function :	Should be set to 1 whenever possible. You may set this value to 0 if you cannot supply clk32 input. Then PHY try to boot without using clk32, at the risk of malfunction.

6.3.3 PORT Declaration

Port name :	cal_blk_clk
Type :	std_logic
Direction :	in
Function :	A clock input used for transceiver calibration. The clock frequency can be any value in the range of 10MHz-125MHz.

Port name :	clk32
Type :	std_logic
Direction :	in
Function :	A clock input used to generate timing for power on reset signals. The clock frequency can be any value in the range of 10MHz-125MHz.

Port name :	clk100
Type :	std_logic
Direction :	in
Function :	A 100MHz differential input used as a reference clock of the Gigabit transceivers.

Port name :	clk125out
Type :	std_logic
Direction :	out
Function :	A 125MHz clock output generated based on `clk100` input. All parallel signals of PHY are synchronized to this clock.

Port name :	clk125plllock
Type :	std_logic
Direction :	out
Function :	Asserted when internal PLL is locked and clock output from `clk125out` becomes stable.

Port name :	rstn
Type :	std_logic
Direction :	int
Function :	An active low reset signal.

Port name :	rx_in
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	in
Function :	Input from the PCI Express high-speed serial receiver port.

Port name :	tx_out
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	out
Function :	Output to the PCI Express high-speed serial transmitter port.

Port name :	wake
Type :	std_logic
Direction :	out
Function :	Not used.

Port name :	phystatus
Type :	std_logic
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	powerdown
Type :	std_logic_vector(1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	txdetectrx
Type :	std_logic
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	txdata
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	txdatak
Type :	std_logic_vector(NLANE*2-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	txelecidle
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	txcompl
Type :	std_logic_vector(NLANE-1 downto 0);
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxpolarity
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	in
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxdata
Type :	std_logic_vector(NLANE*16-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxdatak
Type :	std_logic_vector(NLANE*2-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxvalid
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxelecidle
Type :	std_logic_vector(NLANE-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

Port name :	rxstatus
Type :	std_logic_vector(NLANE*3-1 downto 0)
Direction :	out
Function :	Refer to the specification of the PIPE Interface.

7 License

Basically, GPCIe can freely be used under the following restriction (See `00license` for the details).

- The value (1B1Ah) of vendor ID initially set to address 00h of PCI configuration register should not be modified.
- Any products obtained using GPCIe must explicitly describe the fact "the product is obtained using PCI Express IP Core GPCIe developed and distributed by K&F Computiong Research Co." in a substantial part of the product.
- No activity related to GPCIe should be performed inside the United States.

8 Modification History

version	date	description	author(s)
0.8.1	03-Jan-2009	User's Guide PDF version created.	AK
0.8	17-Nov-2008	Logic optimized.	AK
0.7	09-Oct-2008	Support for x1.	AK
0.6	28-Sep-2008	Support for DMA read.	AK
0.5	28-Jul-2008	DMA write performance improved. User's Guide created.	A. Kawai

Contact address for questions and bug reports:
K&F Computing Research Co. (`support@kfcr.jp`)