

# GPCIe 開発キット ユーザガイド

## for GPCIe DevKit version 0.8

株式会社 K & F Computing Research

E-mail: support@kfcr.jp

### 概要

この文書では PCI Express IP コア GPCIe 開発キットの使用方法を説明します。

## 目次

<b>1</b>	<b>パッケージの内容</b>	<b>2</b>
<b>2</b>	<b>機能概要</b>	<b>3</b>
2.1	対応する PCI Express デバイスタ입	3
2.2	対応する FPGA デバイス	3
2.3	回路構成	4
<b>3</b>	<b>基礎的な使用方法</b>	<b>5</b>
3.1	回路の合成	6
3.2	ホスト計算機からの制御	7
3.3	ソフトウェアのインストール	7
3.4	デバイスドライバの設定	8
3.5	動作テスト	9
3.6	MTRR の設定	12
3.7	ユーザライブラリの使用方法	14
<b>4</b>	<b>より高度な使用方法</b>	<b>16</b>
4.1	ソースコードのアップデート	16
4.2	より高速に動作する回路の生成	16
4.3	PHY チップの使用方法	16
4.4	GPCIe エンジンを直接制御する使用方法	17
<b>5</b>	<b>エンティティ詳細</b>	<b>17</b>
5.1	エンティティ hib 詳細	18
5.2	エンティティ gpcie 詳細	20
5.3	エンティティ phy125 詳細	26

# 1 パッケージの内容

GPCIe は株式会社 K&F Computing Research 開発の PCI Express IP コアです。ユーザ自身の設計した回路へ GPCIe を組み込むことで、PCI Express プロトコルの詳細に立ち入らずに他の PCI Express デバイスとのインタフェースを実現できます。このパッケージには以下の 3 つの回路が含まれます (すべての回路は VHDL ソースコードで提供されます):

1. GPCIe 最上位階層に位置し、ユーザ回路に対して簡便なインタフェーシングを行う Host Interface Bridge (HIB)。
2. PCI Express 規格で定められたトランザクション層、データリンク層、PHY MAC 層と、それらの上位に位置するアプリケーション層 (PCI Configuration Register や DMA コントローラ) を実現する GPCIe エンジン。
3. Altera 社 FPGA の内蔵トランシーバ向け PHY PCS 層、PMA 層 を実現する PHY。

またこのパッケージには、HIB を利用したインタフェース回路のリファレンスデザイン (サンプル回路) と、それを Linux OS から制御するためのソフトウェアも含まれています。このパッケージのファイル構成は以下の通りです:

```
00readme-j          -- 本ファイル。
00readme             -- 本ファイルの英語版。
00license-j          -- パッケージの利用許諾。
00license            -- 00license-j の英語版。
doc/                 -- ユーザガイド、その他の文書。
hib.vhd              -- GPCIe 最上位層 Host Interface Bridge (HIB)。
gpcie.vhd            -- GPCIe のアプリケーション層と PCI Express
                    -- トランザクション層、データリンク層、PHY MAC 層。
phy.vhd              -- Altera 社 FPGA 内蔵トランシーバ用の
                    -- PHY PCS 層と PHY PMA 層。
ifpga_{agx,s2gx}{8,4}.vhd -- HIB を用いたインタフェース回路のサンプル。
synth/              -- サンプル回路の合成に使用するファイル
                    -- (.qpf .qsf .sdc)。
Makefile             -- テンプレートから hib.vhd, gpcie.vhd, phy.vhd
                    -- を生成するための makefile。
templates/          -- VHDL ソーステンプレート
scripts/            -- HIB 制御用ソフトウェアの管理ユーティリティ。
include/            -- HIB 制御ライブラリヘッダファイル (Linux 用)。
lib/                -- HIB 制御ライブラリ (Linux 用)。
driver/             -- HIB デバイスドライバソースコード (Linux 用)。
hibutil/            -- HIB 制御ライブラリソースコード (Linux 用)。
sample/            -- HIB 制御ライブラリを使用したアプリケーション
                    -- プログラムの例。
```

本ユーザガイド第 2 節では GPCIe の機能概要を説明します。第 3 節では GPCIe の基礎的な使用方法について説明します。この節では GPCIe の HIB 階層と Altera 社 FPGA

内蔵トランシーバの利用を前提とした説明を行います。第 4 節では GPCIE のより高度な使用方法について説明します。内蔵トランシーバの代わりに外付けの PHY チップを使用する方法や、HIB 階層を介さずに GPCIE エンジンに直接制御する使用方法の説明を行います。第 5 節では主要な VHDL エンティティの I/O ポートについて説明します。

なお以降では、ファイルのパス名はすべて本パッケージのルートディレクトリ `gpciepkg/` からの相対パスで表記します。

## 2 機能概要

本節では GPCIE のサポートしている機能と FPGA デバイス、回路構成を説明します。

### 2.1 対応する PCI Express デバイスタイプ

エンドポイントとして動作します。スイッチやルートコンプレクス内のルートポートとしては利用できません。

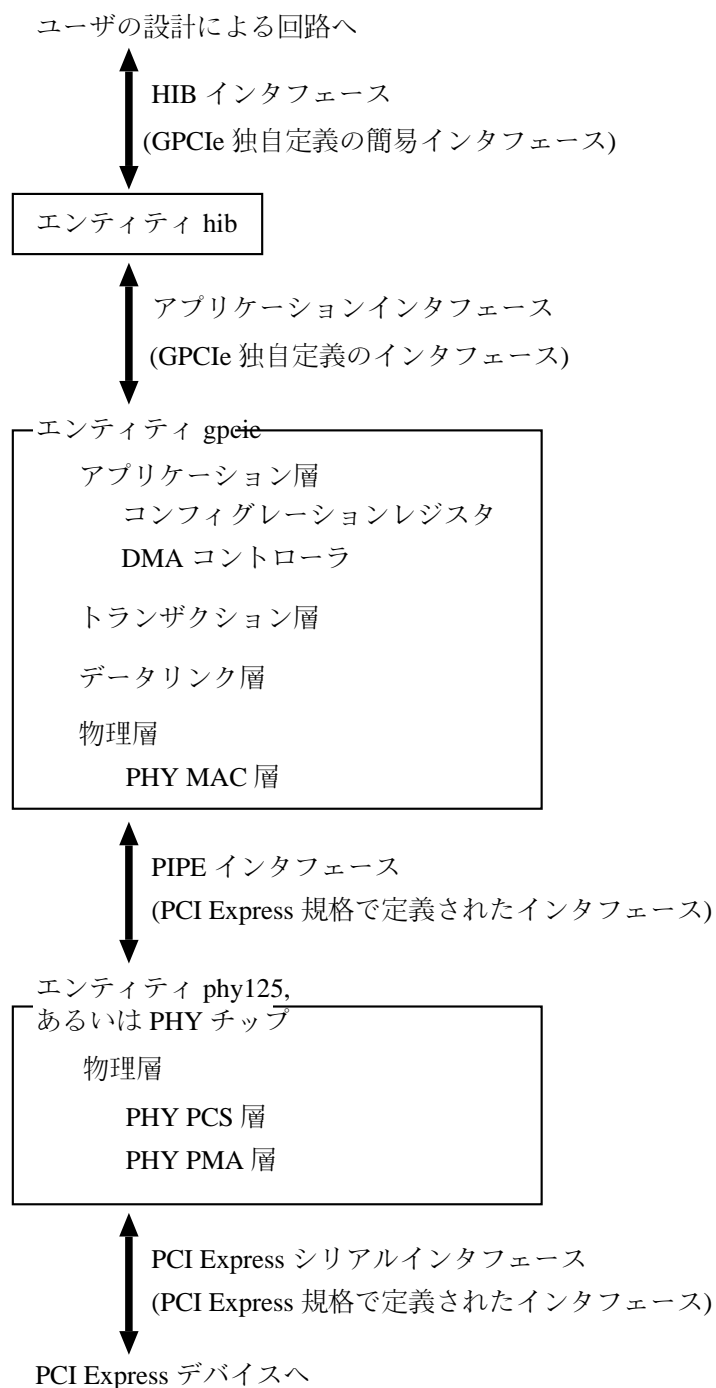
### 2.2 対応する FPGA デバイス

Altera 社の FPGA デバイス Arria GX および Stratix II GX に対応しています。内蔵 Gbit トランシーバを持たないデバイス (*e.g.* Cyclone II, III, Stratix II) の場合にも PIPE インタフェースを持つ外付け PHY チップと組み合わせて使用できるよう設計されていますが、動作確認はおこなっていません。

リビジョン	リンク幅	PIPE I/F	Arria GX	StratixII GX
Gen1.0				
(2.5Gbps)	x8	128b@125MHz	◎	◎
	x4	64b@125MHz	◎	◎
	x1	16b@125MHz	◎	◎
Gen2.0				
(5.0Gbps)	x8	128b@250MHz	-	○
	x4	64b@250MHz	-	○
	x1	16b@250MHz	-	○

◎:対応 ○:無償配布版では非対応。Gen2 対応版 GPCIE のみで対応。

## 2.3 回路構成



GPCIE の回路は hib、gpcie、phy125 の 3 つのエンティティから構成されています。

**エンティティ hib** は GPCIE 最上位階層に位置し、ユーザ回路に対して簡便なインタフェースを提供します。

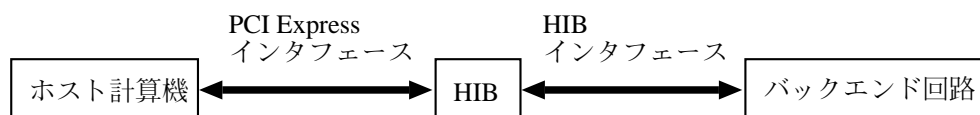
エンティティ **gpcie** は PCI Express 規格で定められた物理層 (PHY MAC 層)、データリンク層、トランザクション層と、その上位に位置するアプリケーション層を実装します。アプリケーション層には PCI Configuration Register と DMA コントローラが含まれます。

エンティティ **phy125** は FPGA 内蔵 Gbit トランシーバ向けの物理層 (PHY PCS 層、PMA 層) を実装します。外付けの PHY チップを用いる場合には GPCie の PIPE インタフェースに直接接続して使用するため、このエンティティは使用しません。

### 3 基礎的な使用方法

本節では GPCie の基礎的な使用方法について説明します。この節では HIB と FPGA 内蔵トランシーバの利用を前提として説明を行います。

ユーザの設計による回路 (以降バックエンド回路と呼びます) が GPCie を使用するには、エンティティ **hib** をインスタンス化します。バックエンド回路は HIB インタフェースを通じてホスト計算機 (より正確には PCI Express リンクの上流に位置するデバイス) との間でデータ転送を行います。HIB は PCI Express プロトコルによるデータ転送と、HIB インタフェースを介したデータ転送とをブリッジします。



ホスト計算機 から HIB への転送は Programm I/O (PIO) による write によって行います。HIB からのホスト計算機への転送は Direct Memory Access (DMA) による write によって行います。本パッケージにはこれらの転送を行うための Linux 用ソフトウェア (デバイスドライバとユーザライブラリ) が含まれています。ソフトウェアの使用方法については後述します。

バックエンドと HIB との転送は `hib_we`, `hib_data`, `backend_we`, `backend_data` の 4 種の信号と、125MHz のクロック信号 `clk_out` を用いて行います。バックエンドから HIB への転送は、バックエンドのドライブする write 信号 `backend_we` と、それに同期したデータ入力 `backend_data` で行います。HIB からバックエンドへの転送は、HIB のドライブする write 信号 `hib_we` と、それに同期したデータ出力 `hib_data` で行います。

バックエンドから HIB への書き込み:

```
clk_out      -- ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
backend_we    ----- ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
backend_data      <d0><d1><d2><d3>
```

HIB からバックエンドへの書き込み:

```
clk_out      -- ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
hib_we        ----- ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
hib_data      <d0><d1><d2><d3>
```

バックエンドは HIB からの書き込みを待たせることはできません。hib\_we が assert されている間は、常にデータを受け取り続けなくてはなりません。

HIB はバックエンドからの書き込みを受け取るためのバッファを内部に持っています。ホスト計算機から DMA write 要求があると、HIB はこのバッファの内容をホスト計算機へ write します。HIB 自身はバッファのオーバーフローチェックを行いませんので、バックエンドがバッファサイズを超えるデータを write する場合には、バックエンドが書き込みタイミングの制御を行う必要があります。バッファのサイズはデフォルトでは 8k byte に設定されています。

HIB 内部には DMA 制御回路や PIO write 制御回路、それらへホスト計算機からアクセスするためのローカルレジスタ群などが実装されています。ローカルレジスタ群へは BAR0 空間を介してアクセスします。PIO write は、BAR2 空間に対して行います。PIO write は BAR2 を write combined に設定して使用すると高い転送速度を得られるよう設計されています。詳細についてはソースコード templates/hibctl.vhd を参照してください。

HIB を実装したユーザ回路のサンプルが ./ifpga\_agx,s2gx8,4.vhd にあります。次節以降ではこのサンプル回路を例に、回路の合成方法とホスト計算機からの制御方法について説明します。

### 3.1 回路の合成

合成には QuartusII を使用します。それ以外のツールでの合成は確認していません。QuartusII のプロジェクトファイル (.qpf) と設定ファイル (.qsf) のサンプルが

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

にあります。これらを使用して前述のサンプル回路

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

を合成すると、KFCR 社製の評価ボード GPCIE-Eval-AGX8 および GPCIE-Eval-S2GX8 で動作する回路が生成されます。なお合成に必要な VHDL ソースファイルは

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

の 2 ファイルのみです。HIB は内部的に GPCIE および FPGA 内蔵トランシーバの制御回路を用いており、それらは `gpcie.vhd`, `phy.vhd` で記述されていますが、これらのファイルは `hib.vhd` に包含されているため、合成には `gpcie.vhd`, `phy.vhd` は必要ありません。

## 3.2 ホスト 計算機からの制御

本パッケージには、HIB をホスト計算機から制御するためのソフトウェアが含まれています。ソフトウェアはデバイスドライバとユーザライブラリから構成されています。これらのソフトウェアは Linux OS 上で動作します。

Linux 以外の OS 向けのソフトウェアは現在のところパッケージには含まれていません。ただしこれは、HIB 回路の設計が Linux OS に依存していることを意味するものではありません。適切なソフトウェアを用意すれば、Linux 以外の OS から HIB を制御することも可能です。

本節ではソフトウェアのインストール、設定、使用方法について説明します。

## 3.3 ソフトウェアのインストール

ソフトウェアのインストールは `./scripts/install.csh` を実行し、その指示に従って下さい。

```
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...
-----
```

(中略)

```
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

なおソフトウェアのインストールには Linux カーネルの完全なソースツリーが必要です。

### 3.4 デバイスドライバの設定

デバイスドライバを使用するには、デバイスドライバを Linux カーネルへリンクする必要があります。そのためには `./driver/` ディレクトリへ移動し、ルート権限下で `make installmodule` を実行します。

```
[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 --
```

この操作によって HIB のデバイスドライバ `hibdrv` が Linux カーネルにリンクされます。この操作はホスト計算機を起動するたびに必要です。`/sbin/lsmmod` コマンドを実行し、その出力に `hibdrv` という語を含む行が含まれていれば、デバイスドライバは正常にリンクされています。

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

デバイスドライバの設定を終えると、ホスト計算機上のプログラムから HIB へアクセスできるようになります。

### 3.5 動作テスト

`./hibutil/hibtest` コマンドを用いると、HIB の動作テストを行えます。例えば PCI configuration register への read/write, local register への read/write、転送速度の測定などが可能です。以下に使用例を示します。

`hibtest` を引数無しで実行すると、使用法が表示されます。

```
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]
```

`hibtest 0` を実行すると、HIB の PCI configuration register の内容が出力されます。

```
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
```

hibtest 10 10 1 を実行すると、ループバック転送のテストを行えます。10 \* 8 byte のデータを PIO write によってホスト計算機から HIB へ送信し、そのデータをそのまま DMA write によって回収します。送信したデータと回収したデータが完全に一致した場合には OK を、一致しない場合には NG を出力します。

```
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
```

hibtest 12 を実行すると DMA write 転送 (HIB からホスト計算機への write) の速度を測定できます。

```
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
```

hibtest 13 1 を実行すると PIO write 転送 (ホスト計算機から HIB への write) の速度を測定できます。

```
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
```

hibtest を用いたその他のテストについてはソースコード `./hibutil/hibtest.c` を参照してください。

### 3.6 MTRR の設定

ホスト計算機から HIB へのデータ転送は、PCI アドレス空間 BAR2 に対する PIO write によって行われます。ピーク性能に近い転送速度を得るには、BAR2 空間を write combining モードに設定する必要があります。設定しない場合でも転送は行えますが、ピーク性能の 20% あるいはそれ以下の転送速度しか得られません。

BAR2 空間を write combining モードに設定するには、ルート権限下でスクリプト `./scripts/setmtrr.csh` を実行します。

```
[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
```

スクリプトの出力に `"base=0xAAAAAAAA (XXXXMB), size = 4kB: write-combining"` という文字列が含まれていれば、正しく設定を行えています。ただし `AAAAAAAA` は HIB の BAR2 空間の先頭アドレスです。この値は `hibtest` コマンドを 2 つの引数 4 18 を与えて実行することで確認できます。

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

ホスト計算機の設定によっては MTRR を設定できないことがあります (メモリを 4GB 以上搭載している場合や、CPU の持つ 8 個すべての MTRR が他の PCI デバイスによって既に使用されている場合など)。BIOS の設定変更によってこの問題を回避できる場合があります。設定方法はチップセットやマザーボードに依存しますので、それらに関する資料を参照してください。

MTRR 設定の前後で hibtest 13 1 を用いた PIO write 転送速度の測定を行うと、速度の向上を確認できます。

#### MTRR 設定前 (8 レーン)

```
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
```

#### MTRR 設定後 (8 レーン)

```
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
```

### 3.7 ユーザライブラリの実装方法

ユーザライブラリはユーザプログラムに対して HIB 制御用の API を提供します。ユーザライブラリを使用するには、ユーザプログラムに `./include/hibutil.h` をインクルードし、実効ファイル生成時にライブラリ `./lib/libhib.a` をリンクしてください。

ユーザライブラリの提供する API のうち主要なものを以下で説明します。説明の無い API については `./hibutil/hibutil.[hc]` を参照してください。

**Hib\* hib\_openMC(int devid)** は識別子 `devid` を持つ HIB の使用権限を取得します。使用権限が他のプロセスによって取得されている場合には、権限を得られるまでブロックします。`devid` は各 HIB 回路ごとにユニークに与えられる小さな整数です。ホスト計算機に  $n$  個の HIB がインストールされている場合には、それぞれの HIB に識別子 0 から  $n - 1$  が割り当てられます。

**hib\_openMC()** は成功すると Hib 型構造体へのポインタを返します。この構造体には HIB に関する情報がまとめられており、必要に応じて他の API が利用します (cf. `hib_dmawMC`)。

**void hib\_closeMC(int devid)** は識別子 `devid` を持つ HIB の使用権限を破棄します。使用権限を破棄した HIB は、他のプロセスから使用できるようになります。

**void hib\_piowMC(int devid, int size, UINT64 \*buf)** は識別子 `devid` を持つ HIB に対して、`buf` でポイントされるアドレスから始まる (`size * 8`) byte に格納されているデータを書き込みます。`buf` にはユーザ空間上で静的に確保した配列や、`malloc()` を用いて動的に確保した領域など、通常のメモリ領域を指定できます。

**void hib\_start\_dmawMC(int devid, int size, UINT64 \*buf)** は識別子 `devid` を持つ HIB に対して、`buf` でポイントされるアドレスから始まる (`size * 8`) byte ヘデータを書き込むよう要求します。

ここで、`buf` には任意のメモリ領域を指定することはできない点に注意してください。`buf` として指定できるのは `h->dmaw_buf`、あるいは `h->dmaw_buf` にオフセットを加えた `h->dmaw_buf + offset` のみです。また `offset + size` の値は 32k byte を超えることはできません。ただし `h` は `hib_openMC()` が返す Hib 型構造体へのポインタです。`h->dmaw_buf` は Linux のカーネル空間内に確保された 32k byte の連続領域を、ユーザ空間へマップしたものです。

通常のメモリ領域 (ユーザ空間上で静的に確保した配列や、`malloc()` など用いて動的に確保した領域) へ HIB から受け取ったデータを格納するためには、いったん HIB から `h->dmaw_buf` ヘデータを受け取り、それをコピーしてください。

**int hib\_finish\_dmawMC(int devid)** は **hib\_start\_dmawMC()** で発行した書き込み要求の終了を待ちます。

**UINT32 hib\_config\_readMC(int devid, UINT32 addr)** は識別子 **devid** を持つ HIB の、PCI Configuration Register アドレス **addr** 番地の値を読み出します。

**void hib\_config\_writeMC(int devid, UINT32 addr, UINT32 value)** は識別子 **devid** を持つ HIB の、PCI Configuration Register アドレス **addr** 番地へ値 **value** を書き込みます。

**UINT32 hib\_mem\_readMC(int devid, UINT32 addr)** は識別子 **devid** を持つ HIB の、Local Register アドレス **addr** 番地の値を読み出します。Local Register のアドレスマップについては `./templates/hibctl.vhd` を参照してください。

**void hib\_mem\_writeMC(int devid, UINT32 addr, UINT32 value)** は識別子 **devid** を持つ HIB の、Local Register アドレス **addr** 番地へ値 **value** を書き込みます。

## ユーザライブラリの使用例

ユーザライブラリを使用したアプリケーションプログラムの例が `./sample/loopback.c` にあります。このプログラムはループバック転送のテストを行います。10 \* 8 byte のデータを PIO write によってホスト計算機から HIB へ送信し、そのデータをそのまま DMA write によって回収します。送信したデータと回収したデータが完全に一致した場合には OK を、一致しない場合には NG を出力します。

```
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

## 4 より高度な使用方法

本節では GPCIE のより高度な使用方法について説明します。4.1 節では回路の変更方法について説明します。4.2 節ではより高速に動作する回路の生成方法を、4.3 節では内蔵トランシーバの代わりに外付けの PHY チップを使用する方法を説明します。4.4 節では HIB を介さずに GPCIE エンジンに直接制御する方法について説明します。

### 4.1 ソースコードのアップデート

GPCIE のソースコードは `./templates/` ディレクトリ内の多数のファイルに分割されて記述されています。これらのうちエンティティ `hib` の依存しているファイルを使用上の便宜からひとつのファイルにまとめたものが `hib.vhd` です。同様に、エンティティ `gpcie` と `phy125` の依存しているファイルをそれぞれひとつにまとめたものが、`gpcie.vhd` と `phy.vhd` です。

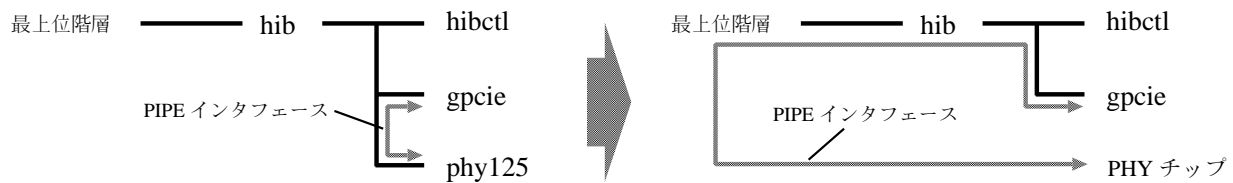
ソースコードに変更を加えた場合には、`gpciepkg` のルートディレクトリで `make` を実行してください。ソースコードの変更が `hib.vhd`、`gpcie.vhd`、`phy.vhd` へ反映されます。

### 4.2 より高速に動作する回路の生成

開発キットのソースコード中には、より高速に動作する回路が用意されています。この回路は、特に Arria GX を用いる場合には、タイミング制約を満たすことが困難なので、初期設定では無効化されています。この回路を生成するには `gpciepkg` のルートディレクトリで `make fast` を実行します。`hib0.vhd`、`gpcie0.vhd` が生成されるので、これらを従来の `hib.vhd`、`gpcie.vhd` と置き換えて使用してください。この回路を用いると、DMA write 転送の実効性能が従来版に比べて 20% 程度向上する可能性があります。PIO write 転送や DMA read 転送の性能は変わりません。

### 4.3 PHY チップの使用方法

FPGA に内蔵のトランシーバの代わりに外付けの PHY チップを使用するには、エンティティ `hib` を変更する必要があります (エンティティ `hib` は `./templates/hibtop.vhd` 内で定義されています)。



エンティティ `hib` はその内部で 3 つのエンティティ `hibctl`、`gpcie`、`phy125` のインスタンスを使用しています。これらに対して 2 つの変更が必要です。まずインスタンス `phy125` を削除します。インスタンス `phy125` は内蔵トランシーバの PHY PCS 層と PHY PMA 層を実装しています。これらの層は PHY チップによって実装されるため、`phy125` は不要になります。この変更にともない、`phy125` と `gpcie` との間の PIPE インタフェースによる接続も削除されます。

次にインスタンス `gpcie` の PIPE インタフェースを PHY チップの PIPE インタフェースと接続します。このためには PHY チップの PIPE インタフェースを FPGA の I/O ピンから回路の最上位階層 (サンプル回路を使用する場合には `./ifpga_agx,s2gx8,4.vhd` 内で定義されているエンティティ `ifpga`) の port に接続し、さらにそれをエンティティ `hib` へと引き入れる必要があります。

## 4.4 GPCIE エンジンを直接制御する方法

HIB はユーザ回路に対して簡便なインタフェースを提供しますが、HIB インタフェースから GPCIE のサポートする機能すべてを利用することはできません。例えば以下の機能を利用するためには、ユーザ回路が HIB を介さずに GPCIE エンジンを直接制御する必要があります。

- アドレスやバイト イネーブル、ウエイト付きの PIO read/write 転送。
- ユーザ独自の BAR0～BAR5 空間定義。
- DMA チャネルの追加・削除。

ユーザ回路から GPCIE エンジンを直接制御するには、`gpci.vhd` 内で定義されているエンティティ `gpcie` をユーザの回路内にインスタンス化します。FPGA 内蔵 Gbit トランシーバを用いる場合には、`phy.vhd` 内で定義されているエンティティ `phy125` もインスタンス化し、両者の PIPE インタフェースを接続します。

## 5 エンティティ詳細

VHDL エンティティ `hib`、`gpcie`、`phy125` は多くの generic パラメタと port を持ちます。以下では特に重要な generic パラメタと、すべての port について説明します。

なおエンティティ `gpcie` の generic パラメタには、default 値として HIB 用に最適化された値が設定されています。用途に応じて設定を変更してください。

パラメタの中には設定を誤ると PCI Express デバイスとして正しく動作しなくなるものや、転送速度に影響を与えるもの、回路資源の消費量に影響を与えるもの、回路の動作周波数に影響を与えるものがあります。パラメタの意味を理解せずに変更することはお薦めできません。

## 5.1 エンティティ `hib` 詳細

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.gpciepkg.all;  -- gpciepkg を use する必要があります。
                        -- gpciepkg は gpcie.vhd 内で定義されています。

entity hib is
  generic (
    DEVICE          : string := "Arria GX";
    -- ターゲットとする FPGA デバイスを指定します。
    -- "Arria GX" もしくは "Stratix II GX" を指定してください。
    NLANE           : integer
    -- PCI Express のリンク幅を指定します。
    -- 1, 4, 8 のいずれかの値を指定してください。
    PIOBUF_DEPTH    : integer := 8;
    -- PIO write バッファの深さ。default は 256 (=2^8) word
    -- (リンク幅 1, 4, 8 レーンの場合 512, 2k, 4k byte) です。
    -- この値は PIO write 性能に影響を与えますが、変更の必要はまれでしょう。
    TXBUF_DEPTH     : integer := 10;
    -- backend_data 回収用バッファの深さ。default は 1024 word (=2^10)
    -- (リンク幅 1, 4, 8 レーンの場合 2k, 8k, 16k byte) です。
    -- 1 回の DMA write 転送でこのサイズを越えるデータ転送を行う必要がある
    -- 場合には、バッファのオーバフローを防ぐための機構をバックエンド回路
    -- 側に実装してください。
    USE_CLK32       : integer := 1
    -- 0 を設定すると clk32 を使わない動作を試みます。0 を設定した場合には
    -- 動作が不安定になる可能性があるため、通常は 1 (default 値) を設定して
    -- ください。何らかの制限により clk32 を供給できない場合にのみ 0 を
    -- 設定してください。
  );
  port (
```

```

phy_linkup      : out std_logic;
  -- PCIe 物理層で link が確立すると assert されます。
  -- LED などの外部出力へ接続し、リンクのインジケータとして使用できます。
dl_linkup       : out std_logic;
  -- PCIe datalink 層の初期化が完了すると assert されます。
  -- LED などの外部出力へ接続し、リンクのインジケータとして使用できます。
clk32           : in  std_logic;
  -- トランシーバの power on reset のタイミングを生成するために使用する
  -- クロック信号です。トランシーバのキャリブレーションにも使用します。
  -- 10MHz-125MHz の任意の周波数の信号を入力してください。
clk100_ext      : in  std_logic;
  -- Gbit トランシーバ用のリファレンスクロックです。
  -- 100MHz のディファレンシャル信号を入力してください。
mperst          : in  std_logic;
  -- リセット線です (active low)。

--
-- PCI Express シリアルインタフェース
--
rx_in           : in  std_logic_vector(NLANE-1 downto 0);
  -- PCI Express 高速シリアル信号の受信ポートです。
tx_out          : out std_logic_vector(NLANE-1 downto 0);
  -- PCI Express 高速シリアル信号の送出ポートです。
wake            : out std_logic;
  -- 未使用。
clk_out         : out std_logic;
  -- PHY PCS 層で clk100 から生成される 125MHz の
  -- パラレルインタフェースクロックです。
  -- HIB のすべてのパラレル信号はこのクロックに同期します。

--
-- バックエンド回路へのインタフェース
--
hib_we          : out std_logic;
  -- HIB からバックエンド回路への write 信号です。この信号に同期して
  -- データが hib_data からバックエンド回路へ出力されます。
hib_data        : out std_logic_vector(NLANE*16-1 downto 0);
  -- HIB からバックエンド回路へのデータ出力です。
backend_we      : in  std_logic;
  -- バックエンド回路から HIB への write 信号です。この信号に同期して

```

```

    -- データが backend_data から HIB へ出力されます。
backend_data : in std_logic_vector(NLANE*16-1 downto 0);
    -- バックエンド回路から HIB へのデータ出力です。
reset_backend : out std_logic;
    -- バックエンド回路へのリセット信号です (active high)。
board_info    : in std_logic_vector(31 downto 0)
    -- ホスト計算機から読み書き可能なメールボックスレジスタ board_info
    -- の初期値を与えます。このレジスタはバックエンド回路が任意の
    -- 目的に使用できます。
);
end hib;

```

## 5.2 エンティティ gpcie 詳細

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.gpciepkg.all; -- gpciepkg を use する必要があります。
                        -- gpciepkg は gpcie.vhd 内で定義されています。

entity gpcie is
generic (
    NLANE                : integer range 1 to 8 := 8;
    -- PCI Express のリンク幅を指定します。
    -- 1, 4, 8 のいずれかの値を指定してください。
    NDMACH               : integer range 0 to 7 := 2;
    -- 実装する DMA チャンネル数を指定します。
    MAX_READ_REQ_SIZE    : integer := 256;
    -- max read request size を指定します。単位は byte です。
    MAX_PAYLOAD          : integer := 256;
    -- この回路でサポートする max payload size の最大値を指定します。
    -- 単位は byte です。実際の通信に使用される max payload size は
    -- 上流デバイスとのネゴシエーションによって決定されます。

    --
    -- Flow Control バッファの深さを指定します。
    --
    CA_PH_VCO_INIT       : integer := 16;
    -- Rx Flow Control バッファ (posted, header) の深さを指定します。

```

```

-- 単位はパケット数です。
CA_PD_VCO_INIT      : integer := 64;
-- Rx Flow Control バッファ (posted, data) の深さを指定します。
-- 単位は 16byte です。
CA_NPH_VCO_INIT     : integer := 2;
-- Rx Flow Control バッファ (non-posted, header) の深さを指定します。
-- 単位はパケット数です。
CA_NPD_VCO_INIT     : integer := 16;
-- Rx Flow Control バッファ (non-posted, data) の深さを指定します。
-- 単位は 16byte です。
CA_CH_VCO_INIT      : integer := 2;
-- Rx Flow Control バッファ (completion, header) の深さを指定します。
-- 単位はパケット数です。
CA_CD_VCO_INIT      : integer := 16;
-- Rx Flow Control バッファ (completion, data) の深さを指定します。
-- 単位は 16byte です。
CL_PH_VCO_INIT      : integer := 16;
-- Tx Flow Control バッファ (posted, header) の深さを指定します。
-- 単位はパケット数です。
CL_PD_VCO_INIT      : integer := 64;
-- Tx Flow Control バッファ (posted, data) の深さを指定します。
-- 単位は 16byte です。
CL_NPH_VCO_INIT     : integer := 2;
-- Tx Flow Control バッファ (non-posted, header) の深さを指定します。
-- 単位はパケット数です。
CL_NPD_VCO_INIT     : integer := 16;
-- Tx Flow Control バッファ (non-posted, data) の深さを指定します。
-- 単位は 16byte です。
CL_CH_VCO_INIT      : integer := 2;
-- Tx Flow Control バッファ (completion, header) の深さを指定します。
-- 単位はパケット数です。
CL_CD_VCO_INIT      : integer := 16;
-- Tx Flow Control バッファ (completion, data) の深さを指定します。
-- 単位は 16byte です。

CFG_VENDOR_ID_INIT   : std_logic_vector(15 downto 0) := x"1b1a";
-- KFCR 社のベンダ ID です。変更しないでください (利用許諾を参照)。
CFG_DEVICE_ID_INIT   : std_logic_vector(15 downto 0) := x"0e70";
-- デバイス ID です。0e70h は KFCR 社が HIB に対して割り当てた値です。
CFG_REVISION_ID_INIT : std_logic_vector( 7 downto 0) := x"01";

```

```

-- リビジョン ID です。
CFG_CLASS_CODE_INIT      : std_logic_vector(23 downto 0) := x"ff0000";
-- PCI クラスコードです。

--
-- PCI Base Address Register0-5 と拡張 ROM の初期値を指定します。
--
CFG_BAR0_INIT             : std_logic_vector(31 downto 0) := x"ffff8008";
-- 32kB, prefetchable, 32-bit address, memory space.
CFG_BAR1_INIT             : std_logic_vector(31 downto 0) := x"fffff008";
-- 4kB, prefetchable, 32-bit address, memory space.
CFG_BAR2_INIT             : std_logic_vector(31 downto 0) := x"ffff8008";
-- 32kB, prefetchable, 32-bit address, memory space.
CFG_BAR3_INIT             : std_logic_vector(31 downto 0) := x"00000000";
-- 未使用
CFG_BAR4_INIT             : std_logic_vector(31 downto 0) := x"00000000";
-- 未使用
CFG_BAR5_INIT             : std_logic_vector(31 downto 0) := x"00000000";
-- 未使用
CFG_BAR_ROM_INIT          : std_logic_vector(31 downto 0) := x"00000000";
-- 未使用
CFG_SUB_VENDOR_ID_INIT    : std_logic_vector(15 downto 0) := x"1b1a";
-- サブベンダ ID です。
CFG_SUB_DEVICE_ID_INIT    : std_logic_vector(15 downto 0) := x"0e70";
-- サブデバイス ID です。
CFG_INT_PIN_INIT          : std_logic_vector( 7 downto 0) := x"00"
-- 使用する割り込み線を指定します。0 を設定してください。
-- 現在のところ GPCIE は割り込みをサポートしていません。
);

port (
  phy_linkup      : out std_logic;
  -- 物理層でリンクトレーニングが終了し、リンクが確立されると
  -- assert されます。
  dl_linkup       : out std_logic;
  -- データリンク層の初期化が終了し、通常の通信を行える状態に
  -- 遷移すると assert されます。
  clk             : in  std_logic;
  -- PHY PCS 層から供給される 125MHz のパラレルインタフェース
  -- クロックを入力します。PIPE インタフェースを含むすべての

```

```

-- パラレル信号はこのクロックに同期します。
rstn          : in  std_logic;
-- リセット線です (active low)。

--
-- PIPE インタフェース
--
phystatus      : in  std_logic;
powerdown      : out std_logic_vector(1 downto 0);
txdetectrx     : out std_logic;
txdata         : out std_logic_vector(NLANE*16-1 downto 0);
txdatak        : out std_logic_vector(NLANE*2-1 downto 0);
txelecidle     : out std_logic_vector(NLANE-1 downto 0);
txcompl        : out std_logic_vector(NLANE-1 downto 0);
rxpolarity     : out std_logic_vector(NLANE-1 downto 0);
rxdata         : in  std_logic_vector(NLANE*16-1 downto 0);
rxdatak        : in  std_logic_vector(NLANE*2-1 downto 0);
rxvalid        : in  std_logic_vector(NLANE-1 downto 0);
rxelecidle     : in  std_logic_vector(NLANE-1 downto 0);
rxstatus       : in  std_logic_vector(NLANE*3-1 downto 0);

--
-- アプリケーションインタフェース
--

-- ターゲット (スレーブ) read/write インタフェース
slv_readreq    : out std_logic;
-- read 要求信号です。要求に対して slv_accept を assert すると
-- read 転送が開始されます。
slv_writereq    : out std_logic;
-- write 要求信号です。要求に対して slv_accept を assert すると
-- write 転送が開始されます。
slv_accept     : in  std_logic;
-- アクセス許可信号です。
slv_read       : out std_logic;
-- read 信号です。ユーザ回路はこの信号の次のクロックで
-- slv_datain ヘデータを供給して下さい。
slv_write      : out std_logic;
-- write 信号です。この信号に同期してデータが slv_dataout から
-- ユーザ回路へ出力されます。

```

```

slv_bar      : out std_logic_vector(6 downto 0);
-- 現在アクセスを受けている base address 空間を示します。
slv_addr     : out std_logic_vector(63 downto 0);
-- 現在アクセスを受けているローカルアドレスを示します。
slv_bytevalid : out std_logic_vector(NLANE*2-1 downto 0);
-- slv_dataout の byte enable 信号です。write アクセス時のみ使用します。

slv_bytecount : out std_logic_vector(11 downto 0);
-- 現在のトランザクションの残りバイト数です。
slv_dataout   : out std_logic_vector(NLANE*16-1 downto 0);
-- GPCIE からのデータ出力です。
slv_datain    : in  std_logic_vector(NLANE*16-1 downto 0);
-- GPCIE へのデータ入力です。

-- イニシエータ (マスター) read/write インタフェース
ms_wrchannel  : out std_logic_vector(NDMACH-1 downto 0);
-- 現在 DMA write アクセスを受けている DMA チャンネルを示します。
ms_write     : out std_logic;
-- DMA write 信号です。ユーザ回路はこの信号の次のクロックで
-- ms_wrdata ヘデータを供給して下さい。
ms_wraddr    : out std_logic_vector(31 downto 0);
-- 現在 DMA write アクセスを受けているローカルアドレスを示します。
ms_wrdata    : in  std_logic_vector(NLANE*16-1 downto 0);
-- GPCIE への DMA write データ入力です。
ms_rdchannel  : out std_logic_vector(NDMACH-1 downto 0);
-- 現在 DMA read アクセスを受けている DMA チャンネルを示します。
ms_read      : out std_logic;
-- DMA read 信号です。この信号に同期してデータが ms_rddata から
-- ユーザ回路へ出力されます。
ms_rdaddr    : out std_logic_vector(31 downto 0);
-- 現在 DMA read アクセスを受けているローカルアドレスを示します。
ms_rddata    : out std_logic_vector(NLANE*16-1 downto 0);
-- GPCIE からの DMA read データ出力です。

--
-- DMA コントローラインタフェース
--
-- 第 0 チャンネルから第 NDMACH-1 チャンネルまでの各 DMA チャンネルに
-- 一組のインタフェースが提供されます。
--

```

```

-- 例) 第 n チャンネルの dma_control 信号は dma_control(n)(6 downto 0)
-- です。なお 2 次元配列型 each7b, each16b, each4b 等はパッケージ
-- gpciepkg 内で定義されています。
--
dma_control      : in  each7b(NDMACH-1 downto 0);
-- DMA レジスタ の制御信号です。
-- dma_control(n)(0) : dma_paddr_low_in(n) の write 信号です。
-- dma_control(n)(1) : dma_paddr_high_in(n) の write 信号です。
-- dma_control(n)(2) : dma_laddr_in(n) の write 信号です。
-- dma_control(n)(3) : dma_size_in(n) の write 信号です。
-- dma_control(n)(4) : dma_param_in(n) の write 信号です。
-- dma_control(n)(5) : 1 クロックのパルスを入力すると DMA 転送を
--                      開始します。
-- dma_control(n)(6) : 1 クロックのパルスを入力すると DMA 転送を
--                      中断します。
dma_param        : in  each16b(NDMACH-1 downto 0);
-- DMA 転送パラメタを設定します。
-- dma_param(n)(8) : 転送方向を設定します。
--                      0:read (ホスト計算機からの読み出し)
--                      1:write (ホスト計算機への書き込み)
dma_status       : out each4b(NDMACH-1 downto 0);
-- DMA 転送の状態を示します。
-- dma_status(n)(3) : 0:転送中 1:転送終了
dma_fifocnt      : in  each13b(NDMACH-1 downto 0);
-- DMA write 時 : 送出用バッファに溜ったデータの byte 数を入力します。
-- DMA read 時  : 受信用バッファの空き byte 数を入力します。
dma_paddr_low_in : in  each32b(NDMACH-1 downto 0);
-- 転送開始アドレスの下位 32-bit (PCI アドレス空間) を入力します。
dma_paddr_high_in : in  each32b(NDMACH-1 downto 0);
-- 転送開始アドレスの上位 32-bit (PCI アドレス空間) を入力します。
dma_laddr_in     : in  each32b(NDMACH-1 downto 0);
-- 転送開始アドレス (ローカルアドレス空間) を入力します。
dma_size_in      : in  each32b(NDMACH-1 downto 0);
-- 転送データサイズを入力します。単位は byte です。
dma_paddr_low_out : out each32b(NDMACH-1 downto 0);
-- 現在転送中のアドレスを示します (PCI アドレス空間、下位 32-bit)。
dma_paddr_high_out : out each32b(NDMACH-1 downto 0);
-- 現在転送中のアドレスを示します (PCI アドレス空間、上位 32-bit)。
dma_laddr_out    : out each32b(NDMACH-1 downto 0);
-- 現在転送中のアドレスを示します (ローカルアドレス空間)。

```

```

    dma_size_out      : out each32b(NDMACH-1 downto 0)
    -- 残り転送データサイズを示します。単位は byte です。
);

end gpcie;

```

### 5.3 エンティティ phy125 詳細

```

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

entity phy125 is
  generic (
    DEVICE      : string := "Arria GX";
    -- ターゲットとする FPGA デバイスを指定します。
    -- "Arria GX" もしくは "Stratix II GX" を指定してください。
    NLANE       : integer;
    -- PCI Express のリンク幅を指定します。
    -- 1, 4, 8 のいずれかの値を指定してください。
    USE_CLK32   : integer := 1;
    -- 0 を設定すると clk32 を使わない動作を試みます。0 を設定した場合には
    -- 動作が不安定になる可能性があるため、通常は 1 (default 値) を設定して
    -- ください。何らかの制限により clk32 を供給できない場合にのみ 0 を
    -- 設定してください。
    SIMULATION  : integer
  );
  port (
    cal_blk_clk      : in    std_logic;
    -- トランシーバのキャリブレーションに使用するクロック信号です。
    -- 10MHz-125MHz の任意の周波数の信号を入力してください。
    clk32            : in    std_logic;
    -- トランシーバの power on reset のタイミングを生成するために使用する
    -- クロック信号です。10MHz-125MHz の任意の周波数の信号を入力してくだ
    -- さい。
    clk100           : in    std_logic;
    -- Gbit トランシーバ用のリファレンスクロックです。
    -- 100MHz のディファレンシャル信号を入力してください。
    clk125out        : out   std_logic;

```

```

-- PHY PCS 層で clk100 から生成される 125MHz のパラレルインタフェース
-- クロックです。GPCIe の PIPE インタフェースを含むすべてのパラレル信号
-- はこのクロックに同期します。
clk125plllock          : out    std_logic;
-- PLL がロックされ、clk125out が安定して出力されている状態で
-- assert されます。
rstn                    : in     std_logic;
-- リセット線です (active low)。

--
-- PCI Express シリアルインタフェース
--
rx_in                   : in     std_logic_vector(NLANE-1 downto 0);
-- PCI Express 高速シリアル信号の受信ポートです。
tx_out                  : out    std_logic_vector(NLANE-1 downto 0);
-- PCI Express 高速シリアル信号の送出ポートです。

--
-- PIPE インタフェース
--
wake                    : out    std_logic;
phystatus               : out    std_logic;
powerdown               : in     std_logic_vector(1 downto 0);
txdetectrx              : in     std_logic;
txdata                  : in     std_logic_vector(NLANE*16-1 downto 0);
txdatak                 : in     std_logic_vector(NLANE*2-1 downto 0);
txelecidle              : in     std_logic_vector(NLANE-1 downto 0);
txcompl                 : in     std_logic_vector(NLANE-1 downto 0);
rxpolarity              : in     std_logic_vector(NLANE-1 downto 0);
rxdata                  : out    std_logic_vector(NLANE*16-1 downto 0);
rxdatak                 : out    std_logic_vector(NLANE*2-1 downto 0);
rxvalid                 : out    std_logic_vector(NLANE-1 downto 0);
rxelecidle              : out    std_logic_vector(NLANE-1 downto 0);
rxstatus                : out    std_logic_vector(NLANE*3-1 downto 0)
);
end phy125;

```

## 利用許諾

PCI コンフィグレーションレジスタに設定するベンダ ID として 1B1Ah (PCI-SIG により当社へ割り当てられた値) を使用すること、成果物の販売または公表時には GPCIE の使用を明らかにすること、が主要な利用条件です。詳細は 00license-j ファイルをご確認下さい。

GPCIE の利用地域は原則として日本国内に限ります。日本国外での利用、ベンダ ID を変更しての利用、未実装機能の実装、Linux 以外のプラットフォームへの対応、Gen2 対応版 GPCIE の入手方法などについては support@kfcr.jp までご相談ください。

## GPCIE 開発キット 更新履歴

version	date	description	author(s)
0.8	17-Nov-2008	回路を最適化。ユーザガイド PDF 版を作成。	AK
0.7	09-Oct-2008	x1 をサポート。	AK
0.6	28-Sep-2008	DMA read 機能を追加。	AK
0.5	28-Jul-2008	DMA write 性能を向上。英文書類を追加。	AK
0.1	10-Jul-2008	パッケージ構成を変更。ユーザガイドを改訂。	AK
0.0	09-Jul-2008	初版作成。	A. Kawai

お問い合わせおよびバグレポートは下記まで:

株式会社 K&F Computing Research (support@kfcr.jp)