■ PowerPointの謎

私は研究会の発表などではMS PowerPointというソフトを使うことが多いのですが、図はほとんどTgifというソフトで作成してPNG形式に変換してからPowerPointに貼り付けています。今までこれで困ったことはほとんどなかったのですが、先日ある研究会で発表するときに直前に確認しようと思って見てみると、貼り付けた絵の何枚かが×印となっていて表示されていませんでした。焦っていろいろと試していたのですが、図を動かすと一瞬だけ元の図が表示されたりするだけで一向に解決できません。途方にくれていたところ、PCを再起動してみたらというアドバイスをいただき、半信半疑で再起動したところ無事図が表示できるようになりました。こんな思いをしてまでこのソフトを使い続ける必要もないとは思うのですが、同じような経験をしたらあせらず再起動してみてください（ちなみにPowerPointだけでの再立ち上げでは直りませんでした）。

■ PCI I/FとLinux

コンパイラの話が続いてソフトのことばかりだと思っている方も多いと思いますので、このあたりでハードウェアの話を1回挟んでおきます。ハードウェアといってもいろいろありますが、取り組みやすいところからと思い、PCI I/FとLinuxでPCIを利用する場合のデバイスドライバの話です。

PCIはPCのインターフェイスとしてはスタンダードといっても良いくらいに普及が進んでいます。PCIを使うことで割り込みやボードのIOアドレス設定などを自動化できるため、ユーザーから見てもハードウェアベンダーから見ても優れたものになっています。さすがに今となっては転送速度に不満が出始め、もっと高速なインターフェイスが模索されていますが、ほとんどの用途にはこの程度の速度でも十分すぎるくらい高速なインターフェイスです。

ところが、このインターフェイスは多くの設定を自動的に行うため、拡張ボードを手軽に作ることが難しくなっており、アマチュア的には敷居が高くなってしまいました。この部分の専用のLSIを購入することもできますが、アマチュアが電子工作をしたいときにはFPGAやCPLDを利用することが多いため、わざわざ専用のLSIをもう1つ購入するのももったいないところです。PCIのIPコアを購入する方法もありますが、量産向けのコアは値段が高く、かつ、仕様がデラックスになりすぎて使いにくいものがあります。

■機能を絞ったPCIコア

私の研究室では簡単に誰でも利用できるように仕様を絞ったPCIのコントローラIPコアを設計していますので、今回はこのPCIインターフェイスIPコア（PCIコア）を紹介したいと思います。このPCIコアは研究室の学部4年生の早坂晴康君が設計したもので、その原形は同じく研究室の大学院1年の孕石裕昭君が2001年度のパルテノン研究会ASIC設計コンテストで最優秀賞を受賞したPCIコアです。2人にはこの場を借りて感謝します。

本PCIコアは、論理合成した結果をALTERA社のCPLDを実装する基板に搭載することでLinuxから簡単にPCI拡張ボードを利用できます。また、このPCIインターフェイスと例題は付録CD-ROMに添付します。PCIコアは表1の条件でご使用いただきたいと思います。

条件が厳しくなっていますが、IPコアはソフトウェアではなく、コアでビジネスをしている方々の邪魔をしたくないのでご了承いただきたいと思います。なお、サンプルとして添付するドライバのライセンスはGPLといたします。

【表1】配布条件など

1	本PCIインターフェイスIPコア（以下PCIコア）は以下の目的だけに用いることができる。 ・他からの資金・機材の援助を受けない個人の興味による私的な実験 ・大学院、大学、短大、高校、高専等の教育機関における授業利用 ・評価を目的とする1ヵ月間以内の試用 ・商用／研究等上記条件以外の目的の利用には別途書面による利用契約ならびに利用許可を要する。
2	本PCIコアは無保証であり、PCIコア自体ならびに本PCIコアを組み込んだ機器の動作上の不具合に対する責任は負わない。またサポートも行わない。
3	本PCIコアを組み込んだ回路もしくは回路データの無許可配布はいかなる形態であれ禁止する。
4	本パッケージはすべてのドキュメントならびにコード・データを改変しない場合には再配布を許可する。ただし、パッケージのアーカイブ形態ならびに文字コード／媒体の変更は改変にはあたらないものとする。

【表2】PCIコアの機能

ターゲット専用	イニシエータ（データ転送主体）の機能は利用者にとって簡単に扱えるようにするためあえて削除しました。研究室にはイニシエータ機能のあるPCIコアもありますので、必要な方はご相談ください。
メモリデバイスとして構成	PCIの規格はさまざまなインターフェイスを想定していますが、本PCIコアではメモリデバイスとしてデバイスを扱います。コア側にどんな回路を付けてもいいのですが、プロセッサから見るとメモリマップされて見えることになるので、扱いは普通のロード・ストア命令でコアのレジスタを読み書きするだけで簡単になります。
バースト転送に対応	DMAコントローラなどバースト転送を行うイニシエータにバースト転送で応答します。
パリティエラー・システムエラー対応	PCIバス上のエラー検出に対応しています。
割り込み	レベルセンシティブな割り込みをサポートします。
20bitコアアドレス空間	実験用には十分広いアドレス空間となっています。

●PCIコアの機能

今回公開するPCIコアは表2に挙げた機能を有しています。PCIコアの概要と信号線を図1に示します。PCI IFと書いた箱の左側はPCIバスの信号です。バスの対応するピンに接続してください。箱の右側がユーザーが利用する信号です。COREというのがユーザーの回路（コア側）になります。

v_idとd_idはユーザー回路のベンダIDとデバイスIDです。ベンダIDには0xffff以外の値を設定してください。PCIのコンフィギュレーションはPCのPCI BIOSとPCIコアの間で行われます。そこで、アドレスの割り当てなど複雑な処理はすべて隠蔽されますのでユーザーは意識する必要はありません(デバイスドライバを作るためには、ベンダIDとデバイスIDは分かっている必要があります)。

コア側から見ると、32ビットのデータ幅の読み出し要求か書き込み要求がコマンド線とともに発生するように見えます。書き込みの時にはバイト単位でどのバイトに書き込むかを指定するバイトイネーブル(nBE[3..0])を用います。書き込み、読み出しどちらとも、終了を信号(tDone)でインターフェイスに連絡します。終了信号が来ない場合、PCI I/Fは16クロックまでは待ちますが、それ以上来ないときにはPCIバスをディスコネクトします。この点にだけ注意すれば、コア側のインターフェイスはISAバスよりも簡単に利用できます。

●読み書きのサイクル

コア側のデータ転送のタイムチャートを図2と図3に示します。プロセッサがコアのデータを読み出す場合、読み出しサイクルが実行されます。このときPCIコアは、コア側インターフェイスにtReadMem信号と読み出しアドレスをクロック同期で同時に発行します。コア側では、データの準備が出きたらクロックに同期してtDoneを発行し、データをPCIコアに転送します。プロセッサがコアにデータを書き込む場合、書き込みサイクルが実行されます。PCIコアはtWriteMem信号と書き込みアドレス、書き込みデータをクロックに同期して同時に発行します。コア側ではデータの書き込み完了とともにtDoneをPCIコアに発行します。

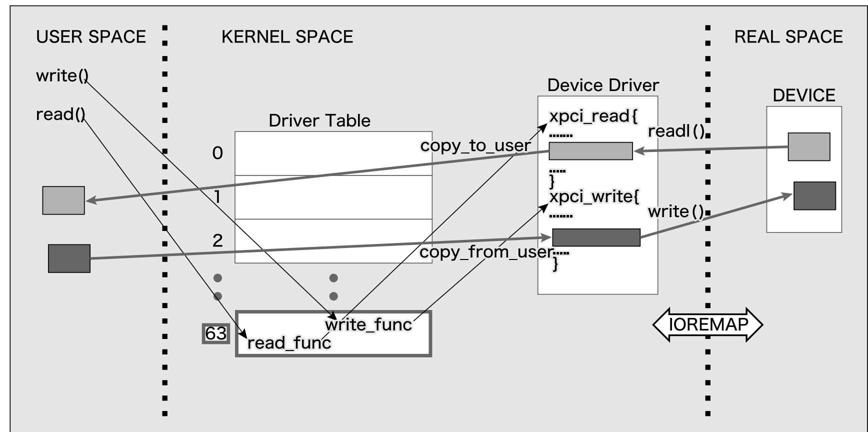
次にコア側が割り込みを起す場合にはPCIコアのINTA線に1を与えます。これは、レベルセンシティブな信号でPCI I/Fに割り込み信号INTAを発生させます。IBM PCのオリジナルな設計では、割り込みはエッジトリガとなっていて、割り込み信号の共用ができないようになっていたのですが、PCIは最近の設計のためさすがにレベルセンスで共用可能な割り込み信号となっています。対応する割り込みは、デバイスドライバが発生元をクリアしない限り信号を上げ続けるようにします。これをコア側で実現するには、割り込み要因が発生したら1をセットするレジスタを用意し、このレジスタはデバイスドライバで読み書き可能なように、コアのアドレスのどこかに割り当てておきます。そして、デバイスドライバは割り込み発生時に要因を解析して対象レジスタを判定した後、レジスタをクリアします。

●コンフィギュレーションレジスタ

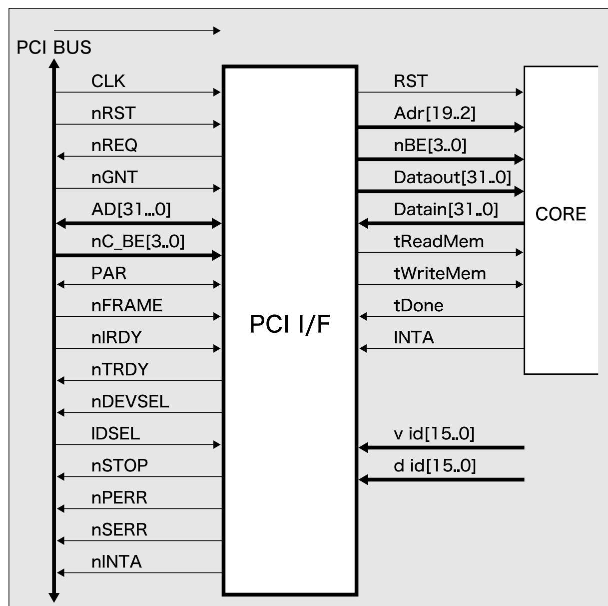
コンフィギュレーションレジスタの話とか、PnP（プラグ&プレイ）であるとか、PCIにまつわるハードウェアの複雑さはすべてPCIコアが対応しますので、ユーザーはこれ以上の知識を必要としません。しかし、それだけでは物足りない人もいるかと思いますから、このPCIコアに用意したコンフィギュレーションレジスタについて説明します。

図4が本PCIコアのコンフィギュレーションレジスタのマップになります。網かけの部分がレジスタとなっていて書き換え可能です。その他の部分は読み出し専用ですが、実際にレジスタとして用意したものは太枠で囲ったレジスタだけです。ベンダIDやデバイスIDはPCIコアのピンに与えた値がそのまま読み出されます。

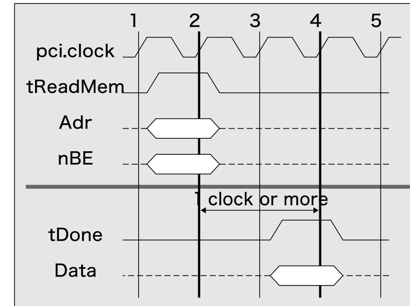
【図4】



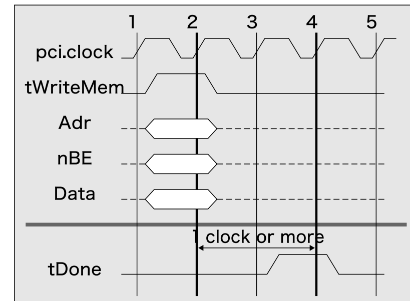
【図1】



【図2】



【図3】



ベースアドレスレジスタは、コア側のアドレス空間がマップされたPCI側のアドレスの先頭を差しています。PCIの仕様では6個のアドレスを別々にPnPで取得できるようになっていますが、本PCIコアでは1つの1Mbyteの空間だけをサポートしています。コンフィギュレーションレジスタを読み出してこのアドレスを取得すれば、後は普通のメモリと同じようにロード／ストア命令でコアを参照できますので、mmapしてあげればユーザープログラムが直接デバイスを触るプログラムを書くことだってできます（もちろん、Linux的には推奨できませんけれど、IO空間でなくわざわざメモリ空間にした理由の1つです）。

■デバイスドライバ

入出力デバイスをLinuxで使う場合、避けて通れないものにデバイスドライバがあります。デバイスドライバは、カーネル空間で動作するデバイス専用のルーチンの集りになります。カーネル空間で動作することで入出力命令も実アドレスも利用できるため、自由度が高く、ユーザーが利用できないさまざまなサービスを提供するために用意されます（実アドレスはioremap()で変換テーブルを作って間接的に利用します。Alphaではioremapは何もしないのですが、i386では多少複雑な処理をしています。カーネルのソースツリーから、arch/i386/mm/ioremap.cをご覧ください）。

UNIX系列のOSでは、デバイスにはメジャー番号とマイナー番号という2つの番号が割り振られます。そのうちのメジャー番号がデバイスドライバの選択のために用いられます。ドライバはカーネルにあるテーブルに登録して利用するのですが、メジャー番号によってこのテーブルを索引します。テーブルのエントリにはデバイスドライバの各ルーチンへのジャンプテーブルが置かれ、デバイスへのサービスが要求された場合カーネルはこのジャンプテーブルからサービスを提供する関数のアドレスを求めてサービスの実行を行います。

サンプルとして提供しているものは、キャラクタ型のデバイスドライバになります。割り込みを使わない場合の使い方は簡単で、

- ・デバイスをオープンする
- ・lseekでアドレス設定する
- ・read／writeで読み書きする
- ・デバイスをクロースする

の順で利用します。

複雑なことをしたい場合にはそれなりに高度なドライバが必要になりますが、この記事で対象としている簡単な電子工作レベルのアプリケーションでは、とにかくソフトウェアからボードのIOレジスタの読み書きができれば十分な用途が多いと思います。割り込みを使う場合でも割り込みハンドラの処理が増えるだけで、基本的にはIOレジスタの読み書きで処理が進められます。

図5を参考に実際のデータの流れを追い掛けてみましょう。read()システムコールがデバイスに対して発行されると、デバイスドライバテーブルの対応するエントリから、read()関数を実行するドライバ本体の関数を間接呼び出しします。read()はデータの読み出しを指示する関数で、デバイスからデータを読み出しユーザーの指定するバッファに書き込みます。

そこで、ドライバはreadb()関数を利用してデバイスコアをアクセスします。readb()の代りに、readw()やreadl()も利用可能です。bが8bit、wが16bit、lが32bitの読み出しをそれぞれ行う関数になります。

ここではlseekで指示したアドレスからデータを読み込むようにします。先ほど書いたようにデバイスドライバはカーネル空間上で動作します。ということは、アドレス変換の異なるユーザー空間のアドレスはそのままでは利用できないことになり、ユーザー側のメモリに値を戻すために専用の関数が用意されています。それがcopy_to_user()で、この関数はユーザー側のアドレス変換に従ってデータをコピーする関数となっています。これを使うことで、readbなどで読み出したデバイスのデータを、ユーザーが利用可能な形でユーザーに引き渡すことができるようになります。

最初からアドレスが決っていれば、デバイスドライバのやることはこれだけなのですが、PCIのPnPの機能により、PCIデバイスのアドレスがメモリ空間上のどこに配置されるのかはコンフィギュレーションが終了してみないと分かりません。

そこで、デバイスドライバはサービス対象のデバイスがどのアドレスに割り当てられたのかをドライバの初期化時に決定しておかなくてはなりません。PCIのデバイスドライバはデバイスを見つけるためにベンダIDとデバイスIDを用います。同じボードを何枚も差している場合があるため、複数のボードをサポートするドライバはシステム中の全ボードを見つける必要があります。

Linuxでこれらのことをサポートする関数は表3に挙げたものになります。pcibios_presentは、PCI機能がある場合に真を返します。

pcibios_find_deviceはベンダID（vendor）と、デバイスID（id）、ボードのインデックス（index）を与えてデバイスを探す関数です。インデックスというのは通し番号です。同じIDを持つ複数のボードや、多機能ボードの各機能に通し番号を振って区別しているので、この番号を0から1つずつ上げてボードの存在を確認していきます。見つかったところで、PCIBIOS_SUCCESSFULと結果が返ります。見つからない場合には、PCIBIOS_DEVICE_NOT_FOUNDが返りますので、その場合にはドライバの登録を諦めます。このときに引数で与えたバス番号（bus）と機能番号（function）にボードを特定する情報が入ります。この情報は次のpcibios_read_config_dwordで用います。システムに1枚しかボードがなく、なおかつ単機能の場合にはインデックスは0だけを与えておけば大丈夫です。

【図5】

31　　24	23　　16	15　　8	7　　0	Adress
Device ID		Vendor ID		00h
Device Status		Device Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	CacheSise	0Ch
Base Address Registers				10h
Base Address Registers				14h
Base Address Registers				18h
Base Address Registers				1Ch
Base Address Registers				20h
Base Address Registers				24h
CardBus Card Infomation Structure Pointer				28h
Subsystem ID		Subsystem Vender ID		2Ch
Expansion ROM Base Address				30h
Reserved				34h
Reserved				38h
Max Latency	Min Grant	Interrrupt Pin	Interrpt Line	3Ch
Device Original Register Space				40h–FCh

（白：Read Only　灰色：Register）

【表3】

```
int pcibios_present(void);

int pcibios_find_device(unsigned short vendor, unsigned short id,
                        unsigned short index, unsigned char *bus,
                        unsigned char *function);

int pcibios_read_config_dword(unsigned char bus, unsigned char function,
                        unsigned char where, unsigned char *ptr);
```

ボードのベースアドレスはコンフィギュレーションレジスタから読み出します。busとfunctionは、pcibios_find_deviceで返された値を用います。whereはコンフィギュレーションレジスタのアドレスを与えますが、ベースアドレスはPCI_BASE_ADDRESS_0という定義がされているので、これを用いることができます。読み出し先はptrで指定します。ここで読み出したベースアドレスは後でioremapでカーネル空間にPCIの空間をマップするために用いますから、staticな変数に保存してください。

デバイスドライバはモジュールとして実装しておくとデバッグに便利です。モジュールの初期化の部分でこれらのPCI関係の情報を取得しておきます。そこで、モジュールの構成は次のようになります。

- pcibios_presentでPCI機能の有無を調べる。
- pcibios_find_deviceでターゲットのボードが実装されているか調べる。
- pcibios_read_config_dwordでターゲットボードのベースアドレスを調べる。
- register_chrdevでキャラクタデバイスのドライバを登録する。

デバイスのメジャー番号はカーネルソースのDocumentationディレクトリにあるdevices.txtを参照して、空いている番号を使ってください。ここでは実験的に63を用いています。

モジュールを使わなくなったときにはcleanup_module関数が呼ばれます。そこでは、unregister_chrdevを呼出しドライバを解放します。

●ファイル操作関数

登録するデバイスドライバのファイル操作関数は

- lseek
- read
- write
- open
- release

になります。

デバイスをオープンするときには

- ioremapでPCIのベースアドレスをカーネル空間にマッピング
- MOD_INC_USE_COUNTマクロで利用カウントを上げる(使用中に削除されないため)。

の処理が必要になります。

また、終了(release)時には

- MOD_DEC_USE_COUNTマクロで利用カウントを下げる。
- iounmapでカーネル空間のマッピングを解除

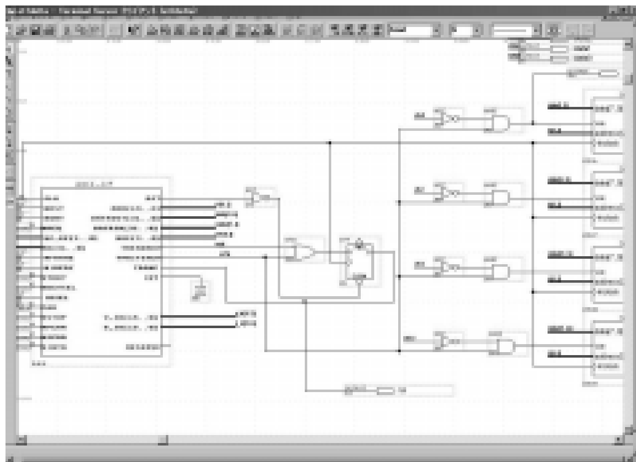
の処理を行います。

デバイスの読み書きはlseekでアドレスを指定しread/writeで実行します。アドレス指定と実行が異なる関数で実現されるため、lseekで指定するアドレスは、staticな変数に格納しておく必要があります。UNIX流の考え方ではlseekはバイト単位に位置決めする関数なのですが、デバイスドライバは比較的自由に自分の好きなように定義しても構いません。例題と提示するドライバはダブルワードを単位として位置決めを行います。また、それに合わせてread/writeもダブルワード境界から4の倍数の長さの転送しか許していません。

read時にはPCIデバイスからデータを読み出すためにreadlを用いています。readl関数はマクロ定義されている関数で、バイトオーダの問題などをこの中に隠蔽してくれます。何度も出てきますが、readlはデバイスからカーネル空間にデータのコピーはするのですが、ユーザーの空間は異なるアドレス変換が必要になるため、カーネル空間から直接データを読み書きすることはできません。そこで、copy_to_user関数を用いて、空間を跨いだデータコピーを行います。

write時はPCIデバイスにデータを書き込むためにwritelを用います。readlと同様にwritelもマクロ定義されていて、バイトオーダなどをこのマクロ内で解決してくれます。こちらもユーザーのアドレス空間からの直接コピーはできませんから、copy_from_user関数を用いてデータコピーを行います。

【画面1】



記事では詳細には書きませんが、ドライバのサンプルファイルを付録CD-ROMに収録していますので、サンプルとオンラインマニュアルを参考にご理解いただければと思います。

最後にこのPCIコアを使った例題としてALTERA社のCPLDの内部メモリ(EAB)をアクセスする回路の回路図を示します(画面1)。PCIコアはデバイス側に32bitの入出力ピンを持っていますので、8bit幅のEABで作成した同期RAMを4個利用しています。EABとPCIコアが簡単に接続できることが分ると思います。後はボードに合わせてピン番号を入力し論理合成するだけで実験ボードが完成します。

■もっと知りたい方は?

ソフトウェアもそうだけれど、ハードウェアのプロジェクトも一発で動くことは少なく、動かなければ試行錯誤で動くまでデバッグを集中的に行うことになります。今回も、回路図を一部使ったのですが、この接続の配線が一部ミスがあって動かないという問題でしばらく悩んでいました。言語で設計するときには名前の突き合わせができるし、クロスリファレンスを作ることもそれほど難しくはないのですが、配線を追い掛けるのは大変です。古い人の中には配線図がないとダメという人もいるようです。しかし、私の15年以上のHDLとの付き合いの経験からいっても配線図は始末に悪い。ブロック図の代替として論理ブロックを並べて、信号名で間の接続を行うのが分りやすい(すなわち誤りにくい)設計への道ではないかと常々思っています。

というわけで、今回はPCIコアの内部を隠蔽して、いかに簡単に使うかを目標に作ったコアの利用方法を中心にまとめました。PCI自体についてもっと知りたい方には物足りないものかもしれませんが、そういう人はぜひご自分でPCIコアを作ってみてください。百聞は一見にしかず、論より証拠、作ってみれば理解が必ず深まること請け負いです。

Resource

[1] OPEN DESIGN No.7(PCIバスの詳細と応用へのステップ)
1835円/ISBN4-7898-3530-8/インターフェース編集部編/CQ出版社

[2] LINUXデバイスドライバ
4800円/ISBN4-900900-73-7/オライリー・ジャパン/Alessandro Rubini著/山崎康宏、山崎邦子共訳