(機械で日本語に翻訳)

The guide to Xillybus Lite

Xillybus Ltd.

www.xillybus.com

Version 3.0

この文書はコンピューターによって英語から自動的に翻訳されているため、言語が不明瞭になる可能性があります。このドキュメントは、元のドキュメントに比べて少し古くなっている可能性もあります。

可能であれば、英語のドキュメントを参照してください。

This document has been automatically translated from English by a computer, which may result in unclear language. This document may also be slightly outdated in relation to the original.

If possible, please refer to the document in English.

1

序章

1.1 全般的

Xillybus Lite は、Linux の下で実行される user space プログラムによって、logic fabric (PL) 内の registers に簡単にアクセスするための単純なキットです。これは、ソフトウェアに bare-metal 環境の錯覚を与え、アドレス、データ、および read/write-enable 信号の簡単なインターフェイスを logic design に提供します。

このキットを使用することで、開発チームは AXI bus インターフェイスや Linux kernel プログラミングを扱う必要がなくなり、オペレーティング システムや bus プロトコルに関する知識がなくても、周辺機器をメモリのように簡単に制御できるようになります。

このキットは、IP core と Linux driver で構成されています。これらは、Zedboard (バージョン 1.1 以降) の Xillinux ディストリビューションに含まれており、プロジェクトに含めるために個別にダウンロードすることもできます。

Xillybus Lite には、DMA 機能は含まれません。最大データレートは 28 MB/s 前後です (processor clock が 666 MHz の場合、1 秒あたり 700 万回の 32 ビット読み取りまたは書き込みアクセス)。

Xillybus Lite IP core は、無料であらゆる用途にリリースされます。特に、商用目的で使用および販売されるバイナリにダウンロード、コピー、および含めることができます。追加の同意や特定のライセンスは必要ありません。

Linux 用の Xillybus Lite driver は GPLv2 の下でリリースされ、Linux kernel 自体と同じ条件で無料で配布できます。

1.2 Xillybus Liteの入手

Xillybus Lite について学び、試してみるには、Xillinux (バージョン 1.1 以降) をダウンロードしてインストールすることをお勧めします。

ディストリビューションには、xillydemo.v/vhd で簡単に変更できるサンプル logic で Xillybus Lite を試すためのすべてがセットアップされています。 Linux 側には、driver が既にインストールされており、いくつかのサンプル プログラムから始めることができます。

Xillinux は http://xillybus.com/xillinux でダウンロードできます

既存のインストールが最新のものであることを確認するには、Xillinux 上の shell prompt で次のチェックを実行できます。

```
# uname -r
3.3.0-xillinux-1.1+
```

接尾辞 (上記の例では “1.1”) は Xillinux のバージョンを示します (この場合はこれで問題ありません)。

Xillybus Lite は、Xillinux に関係なく、どの Zynq-7000 design にも含めることができます。これには、セクション 3 で説明されているように、XPS プロジェクトに IP core を含め、いくつかの配線、compilation、および driver を取り付ける必要があります。

Xillybus Lite バンドルは http://xillybus.com/xillybus-lite でダウンロードできます

2

使用法

2.1 サンプル design

Xillinux (バージョン 1.1 以降) には、接続された IP core、プレインストールされた Linux driver、およびいくつかの単純なデモ user space プログラムで構成されるサンプル design が含まれています。

logic 側では、xillydemo.v(hd) モジュール ソース ファイルに、配列から推論される 32x32 bit RAM の実装が含まれています。この RAM は、host のサンプル プログラムでアクセスされます。その compilation と実行は、次のように Xillinux で直接行うことができます。

```
# make
gcc -g -Wall -I. -O3 -c -o uiotest.o uiotest.c
gcc  -g -Wall -I. -O3  uiotest.o -o uiotest
gcc -g -Wall -I. -O3 -c -o intdemo.o intdemo.c
gcc  -g -Wall -I. -O3  intdemo.o -o intdemo
# ./uiotest /dev/uio0 4096
0 1 2 3
```

C のソースは、/usr/src/xillinux/xillybus-lite/ (バージョン 1.1 以降) の Xillinux のファイル システムにあります。

“uiotest” プログラムは、register 配列の最初の 32 ビット要素に 4 つの値を書き込んでから、それらの値を読み込んで出力するだけですが、より便利なものに簡単に変更できます。

“intdemo” プログラムは、interrupts がどのように処理されるかを示しています。サンプル logic は interrupts をトリガーしないため、そのまま実行しても意味がありません。それにもかかわらず、interrupts がどのように待機されているかを示していま

す。

2.2 Host application とのインターフェース

Xillybus Lite は Linux のユーザー I/O インターフェイス (UIO) に基づいており、主にそのメモリ マッピングによってアクセスされる device file として周辺機器を表します。アクセスを取得するには、次のコードが適用されます。

```
#include <sys/mman.h>

int fd;
void *map_addr;
int size = ...;

fd = open("/dev/uio0", O_RDWR);

if (fd < 0) {
  perror("Failed to open devfile");
  exit(1);
}

map_addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED,
                fd, 0);

if (map_addr == MAP_FAILED) {
  perror("Failed to mmap");
  exit(1);
}
```

エラー チェックを除いて、このコード スニペットは 2 つの操作を実行します。

- 関数 open() を呼び出して device file を開きます (ファイル ハンドルを取得します)。
- 関数 mmap() を呼び出して、デバイスにアクセスするためのアドレスを取得します。 2 番目のパラメーター (“size”) は、マップされるバイト数です。 device tree (変更されていない Xillinux の 4096) に従って、周辺機器に割り当てられたバイト数を超えてはなりません。

map_addr はプロセスの virtual memory space 内のアドレスですが、すべての目的において、bare-metal 環境 (つまり、オペレーティング システムなし) で周辺機器がマップされている physical address であるかのように扱うことができます。

許可されるアクセス範囲は mem_addr から mem_addr + size - 1 までです。ここで、“size” は mmap() に指定された 2 番目の引数です。この範囲を超えてメモリにアクセスしようとすると、segmentation fault が発生する可能性があります。

アドレスが手元にあれば、ペリフェラルのベース アドレス (オフセット ゼロ) で register に 32 ビット ワードを読み書きするのは次のとおりです。

```
volatile unsigned int *pointer = map_addr;

*pointer = the_value_to_write;
the_value_read_from_register = *pointer;
```

特定のメモリ領域では、memory caching は Linux driver によって無効にされ、pointer には volatile のフラグが立てられます。したがって、プログラム内の各読み取りおよび書き込み操作は、bus 操作をトリガーし、その結果、Xillybus Lite の logic インターフェイス信号でアクセス サイクルをトリガーします。

重要:

上記の例に示すように、*pointer* は、*“volatile”* キーワードを使用して *volatile* としてフラグを立てる必要があります。このフラグがないと、*C compiler* は *I/O* 操作を並べ替えたり、最適化したりすることができます。

logic がバイト粒度アクセスをサポートしている場合、8 ビット volatile char pointer または 16 ビット volatile short int pointer を使用してペリフェラルにアクセスすることも問題ありません。

上記の例では、Xillybus Lite 周辺機器が 1 つだけ存在すると想定されています。したがって、最初のインスタンスである “/dev/uio0” が開かれます。追加の UIO デバイスが存在する場合 (たとえば、複数の Xillybus Lite インスタンスがある場合)、それらは /dev/uio1、/dev/uio2 などとして表されます。

どの device file がどの logic 要素に属しているかを知るために、アプリケーションは /sys/class/uio/ で情報を取得する必要があります (例: /sys/class/uio/uio0/name または /sys/class/uio/uio0/maps/map0/addr)。 udev フレームワークは、複数の UIO デバイスが作成されたときに device files の一貫した名前を付けるために推奨されます。

2.3 logic design とのインターフェース

2.3.1 Register 関連の信号

Xillybus Lite IP core は application logic に 7 つの信号を提供します。以下に Verilog 形式で示します。

```
output        user_clk;

output [31:0] user_addr;

output        user_wren;
output  [3:0] user_wstrb;
output [31:0] user_wr_data;

output        user_rden;
input  [31:0] user_rd_data;
```

インターフェイスは同期的で、Xillybus Lite によって提供される user_clk に基づいています (processor の AXI Lite clock に配線されています)。

上記の信号名は、Xillydemo モジュール (Xillinux バンドルの一部) に表示されるものです。 processor のモジュール内の信号の名前は若干異なります。たとえば、user_wren は xillybus_lite_0_user_wren_pin と表示される場合があります。

これらの信号は、標準の block RAM に直接接続できます。その場合、host はその RAM に直接アクセスできます (dual-port RAM を選択した場合、“mailbox” として使用できます)。以下に詳述するように、logic で定義されている registers に接続することもできます。

2.3.2 モジュール階層

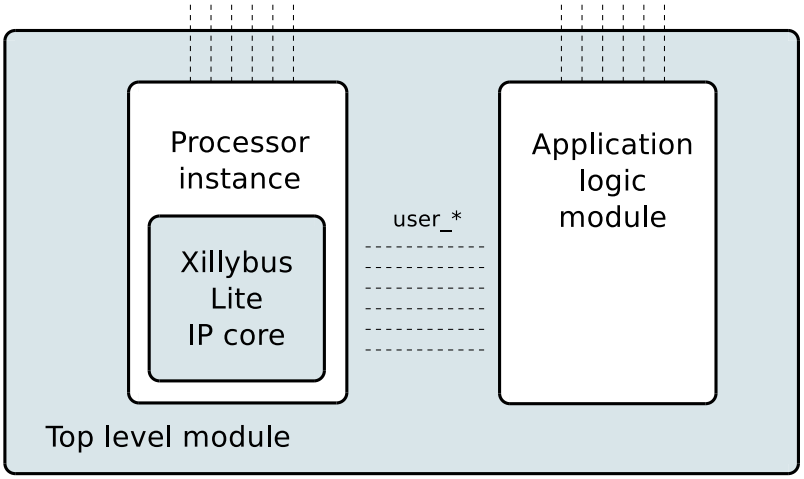
Xilinx logic design が embedded processor を含む場合、それを表すモジュールがあり、通常は top level module でインスタンス化されます。通常、このモジュールによって公開されるポートはすべて、processor が物事の中心であり、その周りの logic は何らかの周辺機器であるというパラダイムに従って、すべて物理ピンに直接接続されます。

Xillybus Lite は、application logic のかなりの部分とのインターフェイスを意図しているため、この共通の構造を多少壊しています。その user_* 信号は、top level module へのルーティングを目的としているため、カスタム logic はその top level module でもインスタンス化されます。プロジェクト全体の構造は、最終的に 2 つ

の大きな塊になります。processor とその IP cores (Xillybus Lite IP core を含む) を含む instantiated module と、application logic を含む 2 番目のモジュールです。user_* 信号は 2 つの間を接続します。

したがって、Xillybus Lite IP core 自体は processor の階層の奥深くにある Xilinx のツールによってインスタンス化されますが、top level module からインターフェイスされます。

これは、Xillinux 用に demo bundle で選択されたレイアウト (下の図を参照) であり、このガイドが想定しているものでもあります。 processor の階層内で Xillybus Lite の信号を内部的に接続することは可能ですが、必ずしも物事が単純になるわけではありません。



2.3.3 32 ビット アラインされた register アクセス

logic (以下の “litearray”) で 32x32 bit 配列にアクセスするには、次のようなコードを使用できます。これは、host が 32 ビット ワード アクセスに固執している場合にのみ正常に機能します (pointers を使用して、たとえば unsigned int のみ):

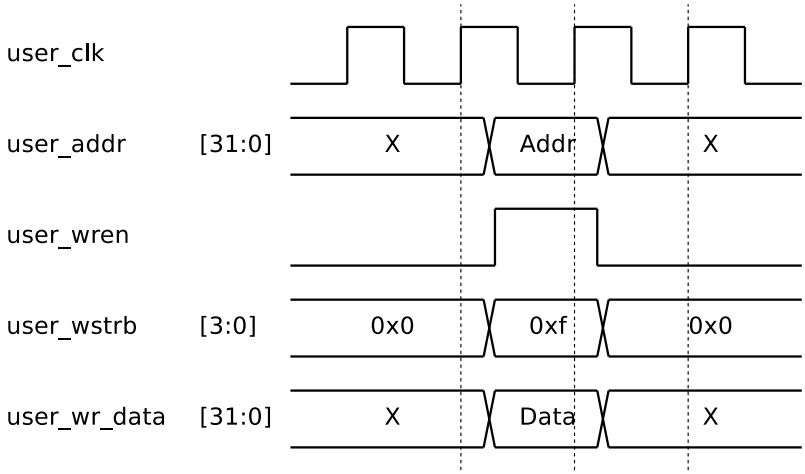
Verilog:

```
always @(posedge user_clk)
  begin
    if (user_wren)
      litearray[user_addr[6:2]] <= user_wr_data;

    if (user_rden)
      user_rd_data <= litearray[user_addr[6:2]];
  end
```

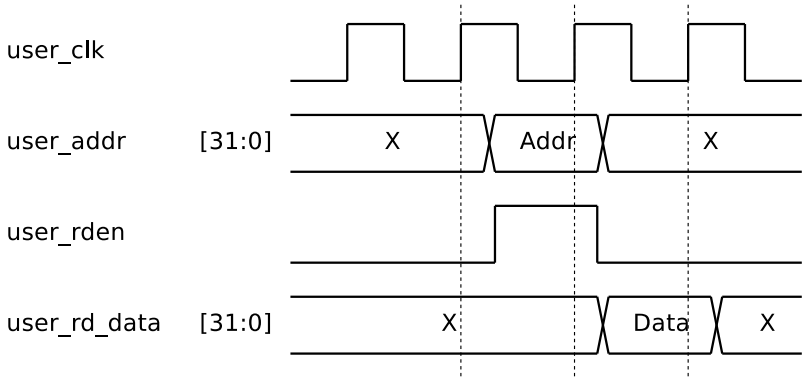
または VHDL:

```
lite_addr <= conv_integer(user_addr(6 DOWNTO 2));

process (user_clk)
begin
  if (user_clk'event and user_clk = '1') then
    if (user_wren = '1') then
      litearray(lite_addr) <= user_wr_data;
    end if;

    if (user_rden = '1') then
      user_rd_data <= litearray(lite_addr);
    end if;
  end if;
end process;
```

アライメントされた書き込みサイクルと任意の読み取りサイクルの波形は次のとおりです。



Waveform 1: Write cycle for aligned access



Waveform 2: Read cycle

備考:

- XPS で Xillybus Lite ペリフェラルに割り当てられたアドレス領域で bus 操作を行うと、必ず 1 つの clock cycle に対して user_wren または user_rden が High になります。
- user_rd_data は、user_rden が High になった後、1 つの clock cycle だけが Xillybus Lite core によって検出されます。したがって、実際には user_rden を監視する必要はありません: user_addr に応じて (clock の latency を 1 つ使用して) user_rd_data を常に更新することも問題ありません。たとえば、

```
always @(posedge user_clk)
  user_rd_data <= litearray[user_addr[6:2]];
```

- 上記のコードは、32 個の要素からなる 32 ビット幅の配列へのアクセスを示しています。より一般的な設定は、アドレス オフセット 0x14 で “myregister” をマッピングするために Verilog

```
always @(posedge user_clk)
  if ((user_wren) && (user_addr[6:2] == 5))
    myregister <= user_wr_data;
```

などで registers にアクセスすることです。

- 同様に、user_addr に依存する case statement は、user_rd_data の値割り当ての一般的な実装です。

```
always @(posedge user_clk)
  case (user_addr[6:2])
    5: user_rd_data <= myregister;
    6: user_rd_data <= hisregister;
    7: user_rd_data <= herregister;
    default: user_rd_data <= 0;
  endcase
```

- user_addr は 32 ビット幅で、アクセスされる完全な物理アドレスを保持します。 enable 信号は、アドレスが割り当てられた範囲内にある場合にのみハイになるため、アドレスの MSBs を確認する必要はありません。
- user_addr[1:0] は常に無視してください。これら 2 つの LSBs は、32 ビット アラインされた bus アクセスでは常にゼロであり、以下で説明するように、アラインされていないアクセスでも無視する必要があります。

2.3.4 アラインされていない register アクセス

host が register 空間に 32 ビット アラインされていない方法でアクセスする可能性がある場合、logic で各バイトを個別に処理する必要があります。

bus ではバイトと 32 ビット ワードへのアクセスに同じ時間がかかるため、アラインされていないアクセスは帯域幅の効率が 4 倍低いことに注意してください。

litearray3、litearray2、litearray1、および litearray0 は、それぞれ 8 ビットの 32 要素のメモリ配列であるとします。次のコード スニペットは、アラインされていないア

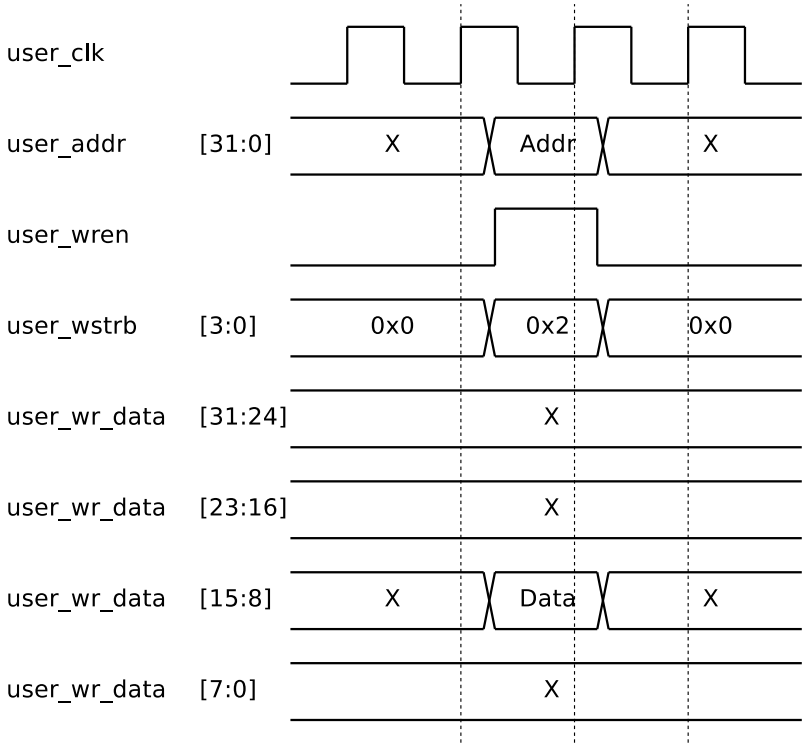
クセスをサポートするために 2.3.3 の例を書き直す方法を示しています。 Verilog:

```
always @(posedge user_clk)
  begin
    if (user_wstrb[0])
      litearray0[user_addr[6:2]] <= user_wr_data[7:0];

    if (user_wstrb[1])
      litearray1[user_addr[6:2]] <= user_wr_data[15:8];

    if (user_wstrb[2])
      litearray2[user_addr[6:2]] <= user_wr_data[23:16];

    if (user_wstrb[3])
      litearray3[user_addr[6:2]] <= user_wr_data[31:24];

    if (user_rden)
      user_rd_data <= { litearray3[user_addr[6:2]],
                        litearray2[user_addr[6:2]],
                        litearray1[user_addr[6:2]],
                        litearray0[user_addr[6:2]] };
  end
```

または VHDL:

```
lite_addr <= conv_integer(user_addr(6 DOWNTO 2));

process (user_clk)
begin
  if (user_clk'event and user_clk = '1') then
    if (user_wstrb(0) = '1') then
      litearray0(lite_addr) <= user_wr_data(7 DOWNTO 0);
    end if;

    if (user_wstrb(1) = '1') then
      litearray1(lite_addr) <= user_wr_data(15 DOWNTO 8);
    end if;

    if (user_wstrb(2) = '1') then
      litearray2(lite_addr) <= user_wr_data(23 DOWNTO 16);
    end if;

    if (user_wstrb(3) = '1') then
      litearray3(lite_addr) <= user_wr_data(31 DOWNTO 24);
    end if;

    if (user_rden = '1') then
      user_rd_data <= litearray3(lite_addr) & litearray2(lite_addr) &
                      litearray1(lite_addr) & litearray0(lite_addr);
    end if;
  end if;
end process;
```

ベース アドレスからの 0x01 オフセットを使用した 1 バイトのアラインされていない書き込みサイクルの波形は次のとおりです。



Waveform 3: Write cycle for unaligned access (byte offset 0x01 shown)

備考:

- 割り当てられたアドレス領域で bus 書き込み操作を行うと、常に user_wren と、1 つの clock cycle に対して user_wstrb のビットの少なくとも 1 つが同時に High になります。上記のように、値の割り当てが user_wstrb に依存する場合、user_wren を確認する必要はありません。
- アラインされていない読み取りアクセスは、logic によってアラインされたアクセスと同じように処理されます。たとえば、processor で実行されているプログラムが 1 バイトを読み取る場合、bus では 32 ビット ワード全体が読み取られ、processor はワードから必要な部分を選択します。
- processor が必要とするアドレスがアラインされていない場合、user_addr[1:0] は非ゼロになることがあります。書き込みサイクルでの logic の正しい動作は user_wstrb のみに依存するため、これは重要ではありません。したがって、これらの 2 つのビットは、アラインされていないアクセスの場合でも無視するのが最善です。

2.4 Interrupts

Xillybus Lite IP core は、application logic が hardware interrupts を processor に送信できるようにする入力信号 user_irq を公開します。これは、同期 positive edge-triggered interrupt request 信号として処理されます。つまり、この信号が 1 つの clock cycle から次の clock cycle へと Low から High に変化すると、interrupt が生成されます。

この信号は、xillydemo.v(hd) モジュールではゼロに保持されます。

Xillybus Lite は、UIO の interrupts の処理方法を採用しています。user space プログラムは、device file からデータを読み取ろうとしてスリープします。 interrupt が到着すると、4 バイトのデータが読み取られ、プロセスが起動します。これらの 4 バイトは、driver がロードされてからトリガーされた interrupts の総数の値を持つ unsigned int として扱われるべきです。プログラムはこの値を無視するか、値が以前に読み取った値に 1 を加えた値であることを確認することにより、interrupts が欠落しているかどうかを確認するために使用できます。

通常のシステム操作では、この interrupt counter は決してゼロにされないことに注意してください。

たとえば、“fd” が /dev/uio0 へのファイル ハンドルであると仮定すると、次のようになります。

```
unsigned int interrupt_count;
int rc;

while (1) {
  rc = read(fd, &interrupt_count, sizeof(interrupt_count));

  if ((rc < 0) && (errno == EINTR))
    continue;

  if (rc < 0) {
    perror("read");
    exit(1);
  }

  printf("Received interrupt, count is %d\n", interrupt_count);
}
```

read() 関数呼び出しには 4 バイトが必要であることに注意してください。それ以外の長さの引数はエラーを返します。 interrupt ファイル記述子は、select() 関数呼び出しで使用できます。

また、EINTR の部分チェックは software interrupts を適切に処理し (プロセスの停止と再起動など)、hardware interrupt とは何の関係もないことに注意してください。

3

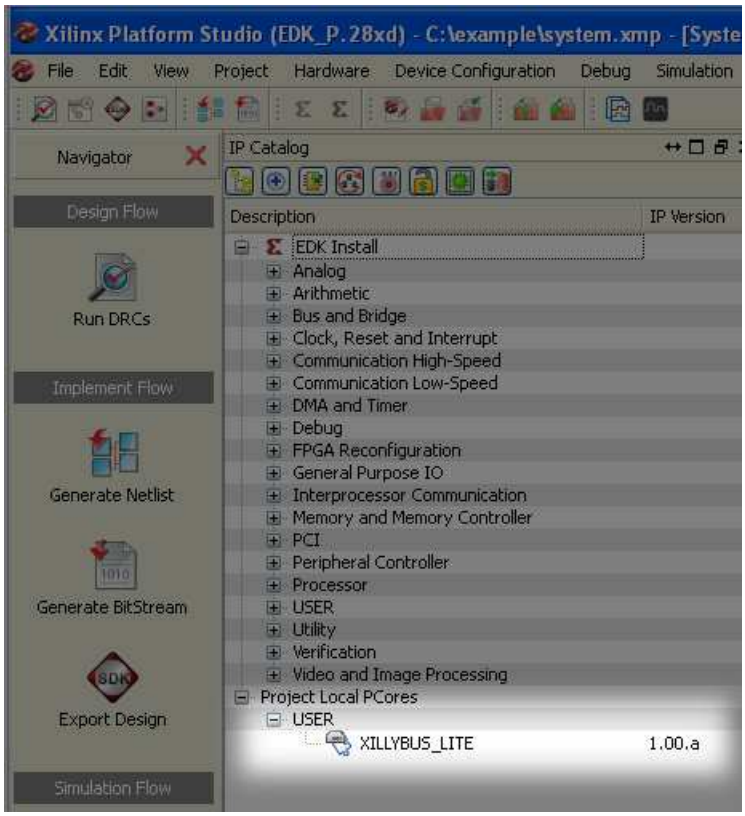
Xillinux を使用しないプロジェクトの Xillybus Lite

3.1 IP core の適用

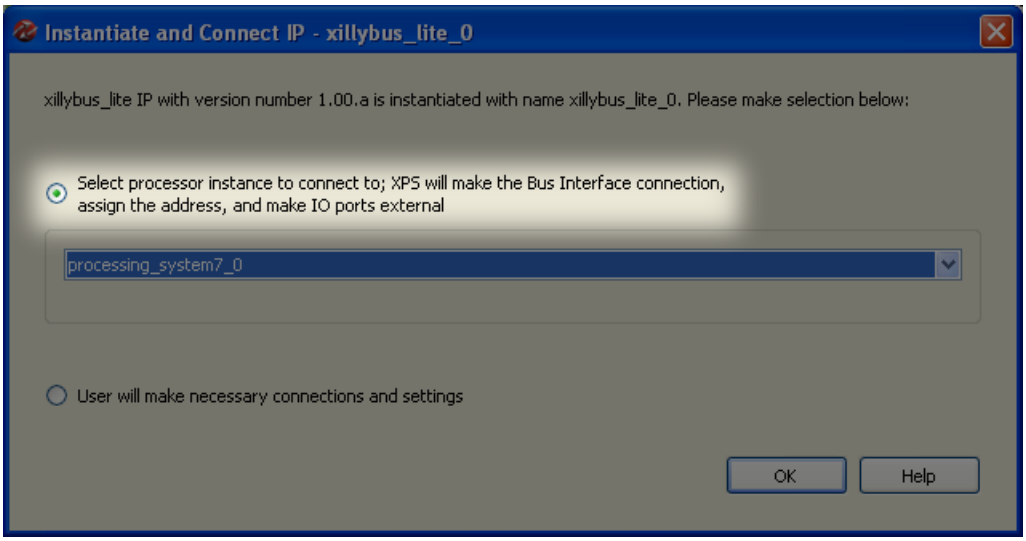
次に説明する手順は Xilinx Platform Studio 14.2 (XPS) に基づいていますが、それ以降のバージョンでもほぼ同じように動作すると予想されます。

Xillybus Lite を既存のプロジェクトに追加するには、次の手順に従います。

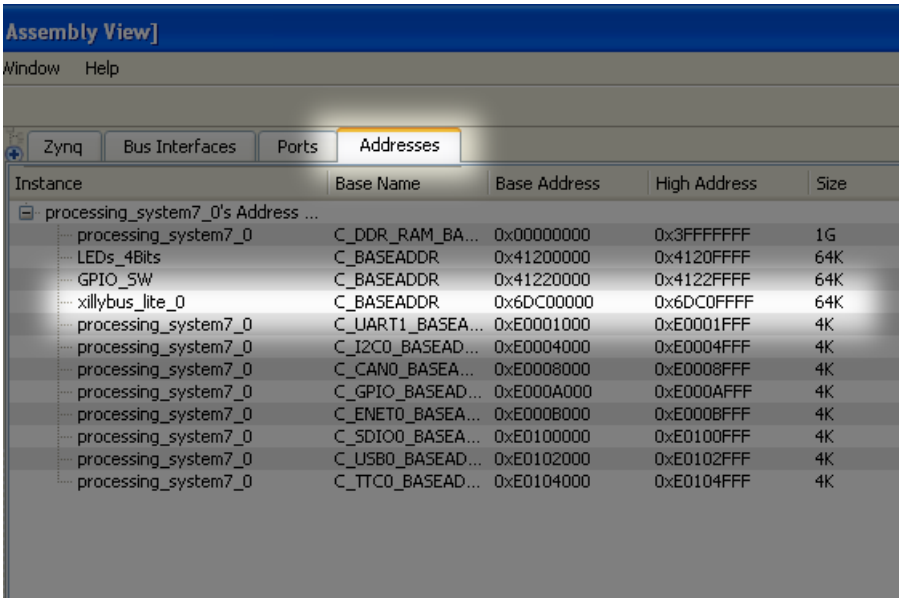
- http://xillybus.com/xillybus-lite から Xillybus Lite バンドルをダウンロードします。
- xillybus_lite_v1_00_a フォルダーを Xillybus Lite のバンドルの “pcores” フォルダーから XPS プロジェクトの “pcores” フォルダーにコピーします。 XPS プロジェクトが生成されたばかりの場合、後者は空である可能性があります。
- 関連するプロジェクトを XPS で開いた状態で、Project > Rescan User Repositories をクリックします。その結果、“USER” エントリが IP カタログの左側の “Project Local PCores” の下に表示されます。このエントリを展開し、XILLYBUS_LITE core を見つけます。



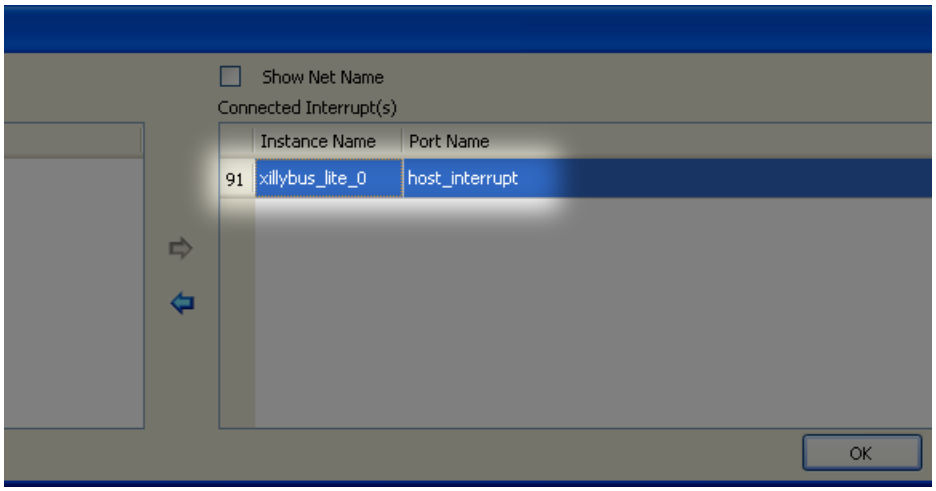
- XILLYBUS_LITE をダブルクリックします。 IP core を design に追加するかどうかを尋ねるポップアップ ウィンドウを確認します。
- XPS Core Config ウィンドウが次に表示されます。 “OK”をクリックするだけです。変更の必要はありません。
- 次のウィンドウで、“Instantiate and Connect IP” は、上部のラジオ ボタンを選択して、XPS が bus インターフェイス接続を確立できるようにします。



- “Addresses” タブを選択し、後で参照できるように xillybus_lite_0 に割り当てられたアドレス範囲をメモします (ベース アドレスとサイズ)。この範囲は必要に応じて変更することもできます。

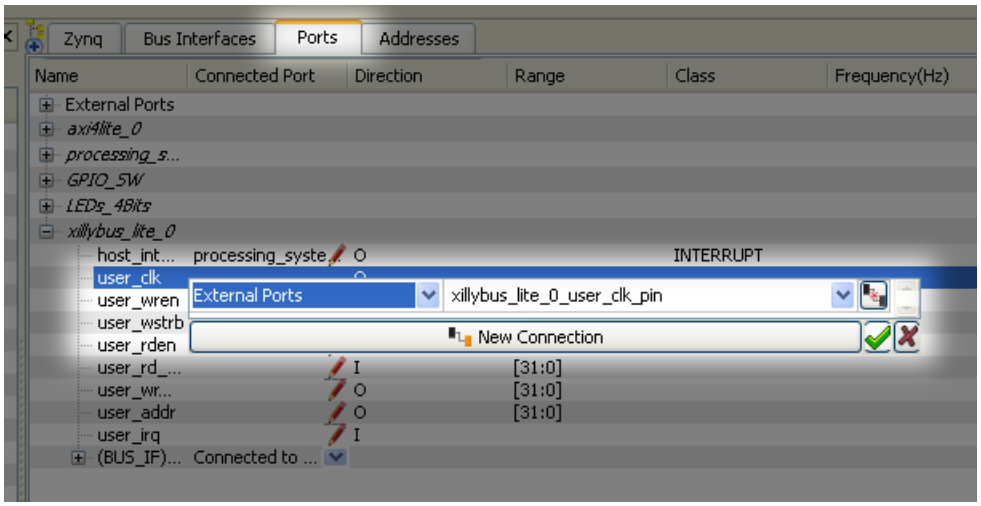


- “Zynq” タブを選択し、IRQ ボックスをクリックします。未接続の interrupts のリストで “host_interrupt” ポートを選択し、中央の矢印をクリックして接続済みの interrupts のリストに移動します。 interrupt number (サンプル スクリーンショットの 91) をメモします。



- “Ports” タブを選択し、“xillybus_lite_0” エントリを展開します。 “user_” プレフィックスを持つ 8 つの信号のそれぞれについて、ポート名の右側にある空のスペースをクリックし、ポートを外部にします。

  drop-down 選択ボックスで “External Port” を選択し、既定のワイヤ名を受け入れます (または変更することもできます)。チェックマークをクリックするか、各ポートの ENTER を押して確認します。



重要:

host_interrupt ポートは *EDGE_RISING* に設定されています。 *level triggered* に変更しないでください。変更すると、*interrupt* がシステムをロックする可能性があります。

XPS プロジェクトをビルドする準備が整いました (例: “Create Netlist”)。

processor のモジュール (通常は “system” という名前) には、8 つの追加ポートがあります。これらは、このモジュールのインスタンス化 (および VHDL のアーキテクチャーの説明) に追加する必要があります。

たとえば、Verilog では

```
...
  wire        user_clk;
  wire        user_wren;
  wire [3:0]  user_wstrb;
  wire        user_rden;
  wire [31:0] user_rd_data;
  wire [31:0] user_wr_data;
  wire [31:0] user_addr;
  wire        user_irq;

...

  system
    system_i (

    ...

      .xillybus_lite_0_user_clk_pin ( user_clk ),
      .xillybus_lite_0_user_wren_pin ( user_wren ),
      .xillybus_lite_0_user_wstrb_pin ( user_wstrb ),
      .xillybus_lite_0_user_rden_pin ( user_rden ),
      .xillybus_lite_0_user_rd_data_pin ( user_rd_data ),
      .xillybus_lite_0_user_wr_data_pin ( user_wr_data ),
      .xillybus_lite_0_user_addr_pin ( user_addr ),
      .xillybus_lite_0_user_irq_pin ( user_irq )
    );
```

user_rd_data と user_irq を除くすべての信号は、processor からの出力です。

3.2 device tree の改造

Xillybus Lite のエントリを追加できるように、既存のシステムの device tree を取得する必要があります。有効な device tree から開始することが重要です。そうしないと、システムの構成が変更されるか、boot プロセスで失敗する可能性さえあります。

Xillinux 2.0 以降では、使用中の device tree ソースは kernel ソースの一部であり、Github からダウンロードできます。これを取得する方法については、Getting started with Xillinux for Zynq-7000 のセクション 6 を参照してください。

Xillinux の以前のリビジョンでは、device tree ソースは devicetree-3.3.0-xillinux-1.1.dts などの /boot ディレクトリにあります。

device tree ソースが利用できない場合は、電源投入時に boot.bin がロードされるのと同じディレクトリにあるバイナリから再構築できます。この問題 (および device tree に関連するその他の問題) を説明するチュートリアルは、次の場所にあります。

http://xillybus.com/tutorials/device-tree-zynq-1

DTS ファイルの bus ペリフェラルを含むセグメント (`axi@0` を囲む中括弧内) に、次のようなエントリを追加する必要があります。

```
xillybus_lite@6dc00000 {
  compatible = "xillybus_lite_of-1.00.a";
  reg = < 0x6dc00000 0x10000 >;
  interrupts = < 0 59 1 >;
  interrupt-parent = <&gic>;
} ;
```

重要:

このサンプル エントリは、*Xillinux* で使用される設定ではなく、上記のスクリーンショットと一致します。

XPS プロジェクトの周辺機器のインスタンスと一致させるために、DTS エントリで次の変更が必要になる場合があります。

- XPS のアドレス マップから取得したベース アドレスを、“reg” の割り当てとノードの名前に設定します (上記の 0x6dc00000、ノードの名前に “0x” プレフィックスを付けない)。

- “reg” の 2 番目の引数を、ベース アドレス (上記の 0x10000) から割り当てられたバイト数に設定します。
- “interrupts” の 2 番目の引数を、XPS で指定された interrupt number から 32 を引いた値に設定します。この例では、XPS が interrupt 91 をペリフェラルに割り当て、結果として 91 - 32 = 59 を割り当てました。
- device tree がバイナリまたは /proc/device-tree/ から reverse compilation によって取得された場合、“gic” ラベルは定義されません。システム内のすべてのデバイスが同じ interrupt controller を使用するため、device tree の他の “interrupt-parent” 割り当てから数値をコピーしても問題ありません。ほとんどの場合、これは単純に `&gic` を値 0x1 に置き換えることを意味します。

device tree ソース ファイルを編集した後、compilation を実行して、Device Tree Compiler (DTC) を使用してバイナリ BLOB (DTB) に変換します。この compiler は Linux kernel tree の一部です。

実行中の Xillinux システムでは、これは次のように実行できます。 kernel ツリーのディレクトリは、使用されている Xillinux ディストリビューションによって異なります。

```
# cd /usr/src/kernels/3.3.0-xillinux-1.1+/
# scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb my_device_tree.dts
```

次に、boot に使用されるメディア上の既存の devicetree.dtb ファイルを、生成されたばかりのファイルで上書きします。

Xillinux は、Xillinux 以外のシステム向けの device tree の compilation に使用できることに注意してください。

3.3 Linux driverのCompilation

driver は、“linuxdriver” ディレクトリの Xillybus Lite バンドルにあります。

compilation には 2 つのオプションがあります。

- 意図した kernel の source code (または少なくともその headers) に対する ARM processor の cross compilation 。
- 対象のディストリビューションに GNU tools および kernel headers がインストールされている場合、Zynq ボード自体に compilation を直接接続することができます。

Xillybus lite は、いくつかの kernel オプション、特に UIO オプション (CONFIG_UIO) が少なくともモジュールとして有効になっていることに依存しています。

以下では、ボード上の直接 compilation を想定しています。これは Xillinux ディストリビューションで実行できますが、Xillinux で Xillybus Lite を実行する場合は必要ありません (driver が既にインストールされているため)。一方、driver の compilation が Xillinux で (したがって Xillinux の kernel headers に対して) 実行された場合、binary は他の kernels では動作しない可能性があります。

最初の変更ディレクトリ:

```
$ cd /path/to/linuxdriver
```

driver の compilation の場合は、“make” と入力します。セッションは次のようになります。

```
$ make
make -C /lib/modules/3.3.0/build SUBDIRS=/tmp/lite/linuxdriver modules
make[1]: Entering directory '/usr/src/kernels/3.3.0'
  CC [M]  /tmp/lite/linuxdriver/xillybus_lite_of.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /tmp/lite/linuxdriver/xillybus_lite_of.mod.o
  LD [M]  /tmp/lite/linuxdriver/xillybus_lite_of.ko
make[1]: Leaving directory '/usr/src/kernels/3.3.0'
```

kernel module の compilation は、compilation 中に実行されている kernel 用に特別に行われていることに注意してください。別の kernel を使用する場合は、“make TARGET=kernel-version” と入力します。ここで、“kernel-version” は、/lib/modules/ に表示されるように、目的の kernel バージョンです。

セッションの出力は多少異なる場合がありますが、エラーや警告は表示されません。

特に、これらの警告が表示された場合、

```
WARNING: "__uio_register_device" [xillybus_lite_of.ko] undefined!
WARNING: "uio_unregister_device" [xillybus_lite_of.ko] undefined!
```

は、対象の kernel に UIO オプションがなく、driver を kernel に挿入すると失敗する可能性が高いことを意味します。

3.4 driver の取り付け

xillybus_lite_of.ko ディレクトリを既存の driver サブディレクトリにコピーし、次のように depmod を実行します (目的の kernel が現在実行されていると仮定します)。

```
# cp xillybus_lite_of.ko /lib/modules/$(uname -r)/kernel/drivers/char/
# depmod -a
```

インストールは、driver を kernel にすぐにはロードしません。 Xillybus Lite 周辺機器が検出された場合、システムの次の boot でこれが行われます。 driver を手動でロードする方法を次に示します。

3.5 driver のロードとアンロード

driver をロードする (そして Xillybus Lite で作業を開始する) には、root と入力します。

```
# modprobe xillybus_lite_of
```

これにより、Xillybus Lite device file (/dev/uio0) が表示されます。

システムが boot を実行したときに Xillybus Lite 周辺機器が存在し、上記のように driver がすでにインストールされている場合、これは必要ないことに注意してください。

kernel のモジュールのリストを表示するには、“lsmod” と入力します。 kernel から driver を削除するには、次の手順を実行します。

```
# rmmod xillybus_lite_of
```

これにより、device file が消えます。

何か問題が発生したと思われる場合は、/var/log/syslog ログ ファイルで “xillybus” という単語を含むメッセージを確認してください。多くの場合、このログ ファイルには貴重な手がかりが見つかります。

/var/log/syslog ログ ファイルが存在しない場合は、代わりにおそらく /var/log/messages です。

__uio_register_device に関する “unknown symbol” エラーまたは同様のエラーがログ ファイルに表示される場合は、実行中の kernel に UIO 構成オプションがないことを示しています。