(机器翻译成中文)

The guide to Xillybus Lite

Xillybus Ltd.

www.xillybus.com

Version 3.0

本文档已由计算机自动翻译，可能会导致语言不清晰。 与原始文件相比，该文件也可能略微过时。

如果可能，请参阅英文文档。

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 This document may also be slightly outdated in relation to the original.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

介绍

1.1 一般的

Xillybus Lite 是一个简单的套件，用于通过在 Linux 下运行的 user space 程序轻松访问 logic fabric (PL) 中的 registers。它向软件呈现 bare-metal 环境的错觉，向 logic design 呈现一个简单的地址、数据和 read/write-enable 信号接口。

使用该套件使开发团队无需处理 AXI bus 接口和 Linux kernel 编程，并且无需任何操作系统或 bus 协议知识即可对外围设备进行直接的类似存储器的控制。

该套件由 IP core 和 Linux driver 组成。这些包含在 Zedboard（1.1 及更高版本）的 Xillinux 发行版中，也可单独下载以包含在项目中。

Xillybus Lite 不涉及任何 DMA 功能。最大数据速率约为 28 MB/s（每秒 700 万次 32 位读取或写入访问，processor clock 为 666 MHz）。

Xillybus Lite IP core 可免费用于任何用途。特别是，它可以被下载、复制和包含在用于商业目的的二进制文件中，没有任何限制，也没有任何额外的同意或特定许可。

Linux 的 Xillybus Lite driver 是在 GPLv2 下发布的，这使得它可以按照与 Linux kernel 本身相同的条款免费分发。

1.2 获取 Xillybus Lite

学习Xillybus Lite并试用，建议下载安装Xillinux（1.1及以上版本）。

该发行版已为在 logic 样本上试用 Xillybus Lite 进行了所有设置，可以在 xilly-demo.v/vhd 中轻松修改。 Linux 端已经安装了 driver 和几个示例程序开始使用。

Xillinux可以在http://xillybus.com/xillinux下载

要验证现有安装是否足够新，可以在 Xillinux 上的 shell prompt 上运行以下检查：

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

后缀（上例中的 “1.1”）表示 Xillinux 版本（在这种情况下是可以的）。

Xillybus Lite 可以包含在任何 Zynq-7000 design 中，与 Xillinux 无关。这需要在 XPS 项目中包含 IP core、一些接线、compilation 和 driver 的安装，如 3 部分所述。

Xillybus Lite 捆绑包可以在 http://xillybus.com/xillybus-lite 下载

2

用法

2.1 样品 design

Xillinux（1.1 及更高版本）中包含一个示例 design，包括一个连接的 IP core、一个预安装的 Linux driver 和几个简单的演示 user space 程序。

在 logic 端，xillydemo.v(hd) 模块源文件包含一个 32x32 bit RAM 的实现，它是从一个数组中推断出来的。在 host 的示例程序中访问此 RAM。它的 compilation 和执行可以直接在 Xillinux 上完成，如下：

```
# 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 源代码可以在 /usr/src/xillinux/xillybus-lite/（1.1 及更高版本）的 Xillinux 文件系统中找到。

“uiotest” 程序仅将四个值写入 register 数组中的前 32 位元素，然后读回并打印它们的值，但它很容易变成更有用的东西。

“intdemo” 程序显示了如何处理 interrupts。由于样本 logic 不会触发任何 interrupts，因此按原样运行它是没有意义的。尽管如此，它显示了 interrupts 是如何被等待的。

2.2 与 Host application 的接口

Xillybus Lite 基于 Linux 的用户 I/O 接口（UIO），它将外围设备表示为主要通过其内

存映射访问的 device file。要获得访问权限，以下代码适用：

```
#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);
}
```

除错误检查外，此代码段执行两个操作：

- 调用函数 open() 打开 device file（获取文件句柄）。
- 调用函数 mmap() 获取访问设备的地址。第二个参数 (“size”) 是映射的字节数。根据 device tree（未修改的 Xillinux 上的 4096），它不能超过为外设分配的字节数。

map_addr 是进程的 virtual memory space 中的一个地址，但出于所有目的，它可以被视为在 bare-metal 环境（即没有操作系统）中外设映射到的 physical address。

允许的访问范围从 mem_addr 到 mem_addr + size - 1，其中 “size” 是给 mmap() 的第二个参数。尝试访问超出此范围的内存可能会导致 segmentation fault。

有了手头的地址，在外设的基地址（偏移量零）处向 register 写入和读取 32 位字只是：

```
volatile unsigned int *pointer = map_addr;

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

在特定的内存区域，memory caching 被 Linux driver 禁用，pointer 被标记为 volatile。因此，程序中的每个读写操作都会触发一次 bus 操作，从而触发 Xillybus Lite 的 logic 接口信号的访问周期。

重要的:

pointer 必须使用 *"volatile"* 关键字标记为 *volatile*，如上例所示。缺少此标志将允许 *C compiler* 重新排序并可能优化 *I/O* 操作。

如果 logic 支持字节粒度访问，也可以使用 8 位 volatile char pointer 或 16 位 volatile short int pointer 访问外设。

在上面的示例中，假设只有一个 Xillybus Lite 外围设备存在。因此打开第一个实例 "/dev/uio0"。如果存在其他 UIO 设备（例如，存在多个 Xillybus Lite 实例），则它们表示为 /dev/uio1、/dev/uio2 等。

为了知道哪个 device file 属于哪个 logic 元素，应用程序应该获取 /sys/class/uio/ 中的信息（例如 /sys/class/uio/uio0/name 或 /sys/class/uio/uio0/maps/map0/addr ）。当创建多个 UIO 设备时，建议使用 udev 框架来统一命名 device files。

2.3 与 logic design 的接口

2.3.1 Register相关信号

Xillybus Lite IP core 向 application logic 提供七个信号，此处以 Verilog 格式给出：

```
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;
```

该接口是同步的，并且基于由 Xillybus Lite 提供的 user_clk（它连接到 processor 的 AXI Lite clock）。

上面的信号名称是出现在 Xillydemo 模块（Xillinux 捆绑包的一部分）中的名称。 processor 模块中的信号名称略有不同，例如user_wren 可能显示为xillybus_lite_0_user_wren_pin。

这些信号可以直接连接到标准 block RAM，在这种情况下，host 可以直接访问该 RAM（如果选择 dual-port RAM，则可以用作 “mailbox”）。它们也可以连接到 logic 中定义的 registers，如下所述。

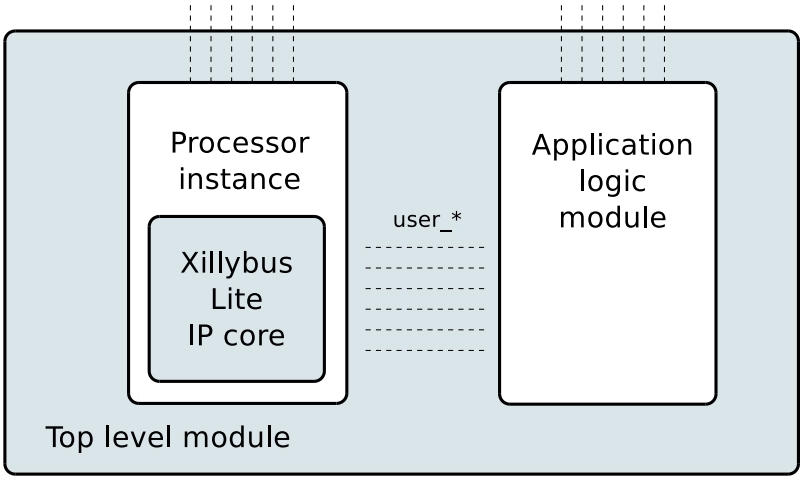
2.3.2 模块层次结构

当 Xilinx logic design 涉及 embedded processor 时，会有一个代表它的模块，通常在 top level module 中实例化。通常，该模块暴露的端口都直接连接到物理引脚，遵循 processor 是事物中心的范式，并且它周围的任何 logic 都是某种外围设备。

Xillybus Lite 旨在与大部分 application logic 接口，因此在某种程度上打破了这种通用结构：它的 user_* 信号旨在路由到 top level module，因此自定义 logic 也在 top level module 中实例化。整个项目的结构最终分为两大块：一个包含 processor 及其 IP cores（包括 Xillybus Lite IP core）的 instantiated module 和一个包含 application logic 的第二个模块。 user_* 信号连接在两者之间。

因此，即使 Xillybus Lite IP core 本身是由 Xilinx 的工具在 processor 层次结构深处的某个地方实例化的，它也与 top level module 接口。

这是 demo bundle 中为 Xillinux 选择的布局（如下图所示），也是本指南的假设。可以在 processor 的层次结构内部连接 Xillybus Lite 的信号，但这并不一定会使事情变得更简单。



2.3.3 32 位对齐 register 访问

要访问 logic（下面的 “litearray”）中的 32x32 bit 数组，可以使用如下代码。仅当 host 坚持 32 位字访问（仅使用 pointers 到例如 unsigned int）时，这才能正常工作：

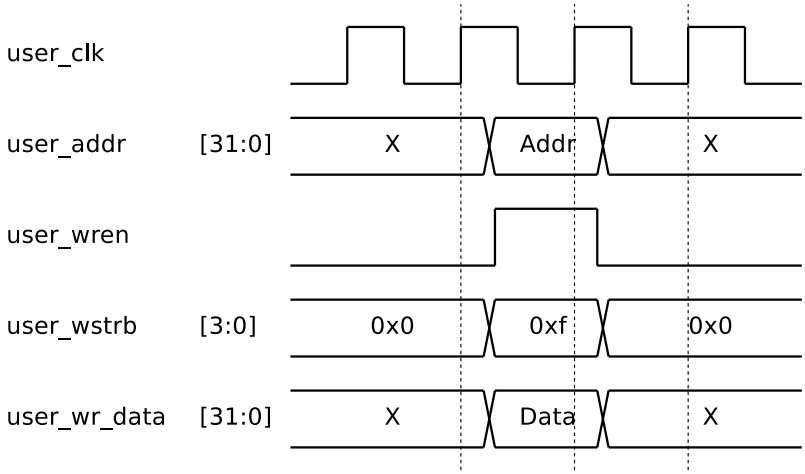
在 Verilog 中：

```
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
```

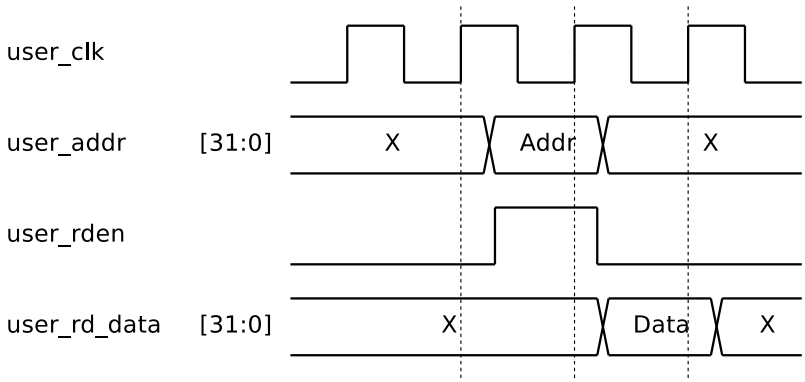
或在 VHDL 中：

```
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;
```

对齐的写周期和任何读周期的波形是：



Waveform 1: Write cycle for aligned access



Waveform 2: Read cycle

备注:

- 在 XPS 中分配给 Xillybus Lite 外设的地址区域上的任何 bus 操作总是导致 user_wren 或 user_rden 恰好为一个 clock cycle 为高电平。
- 在 user_rden 为高电平后，user_rd_data 只被 Xillybus Lite core 感应到一个 clock cycle。因此实际上没有必要监视 user_rden：根据 user_addr （使用一个 clock 的 latency ）始终更新 user_rd_data 也是可以的，例如，

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

- 上面的代码演示了对 32 个元素的 32 位宽数组的访问。更常见的设置是访问 registers，例如在 Verilog

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

  中，以在地址偏移 0x14 处映射 “myregister”。

- 同样，一个依赖于 user_addr 的 case statement 是 user_rd_data 赋值的常见实现，比如

```
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 为 32 位宽，并保存正在访问的完整物理地址。由于 enable 信号仅在地址在分配范围内时为高电平，因此无需验证地址’MSBs。

- 始终忽略 user_addr[1:0]。这两个 LSBs 在 32 位对齐的 bus 访问中始终为零，如下所述，即使对于未对齐的访问，它们也应该被忽略。

2.3.4 未对齐的 register 访问

当 host 有可能以 32 位非对齐方式访问 register 空间时，需要在 logic 中单独处理每个字节。

请注意，在 bus 上访问一个字节和一个 32 位字需要相同的时间，因此未对齐访问的带宽效率低了四倍。

假设 litearray3、litearray2、litearray1 和 litearray0 是 32 个元素的内存数组，每个元素 8 位。以下代码片段演示了如何重写 2.3.3 中的示例以支持非对齐访问。在 Verilog

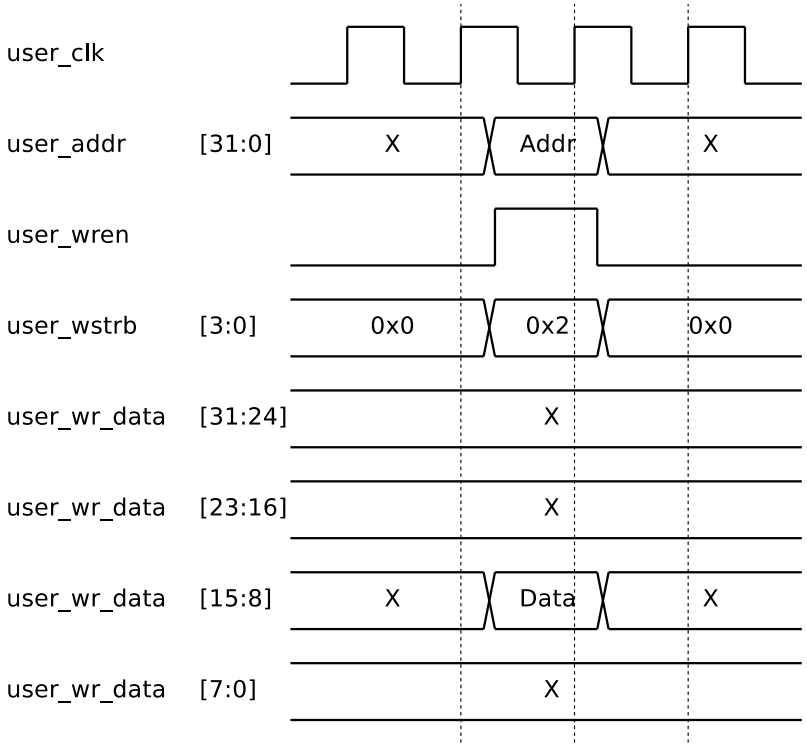
中：

```
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
```

或在 VHDL 中：

```
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;
```

以下是在单个字节上的未对齐写入周期的波形，其中 0x01 偏移从基地址开始。



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

备注:

- 对分配的地址区域执行写入 bus 操作始终会导致 user_wren 和 user_wstrb 的至少一个位同时为一个 clock cycle 为高。如上图，如果赋值依赖于user_wstrb，则无需勾选user_wren。
- logic 处理未对齐的读取访问与对齐的读取访问相同。例如，当在 processor 上运行的程序读取一个字节时，在 bus 上读取整个 32 位字，然后 processor 从字中挑选所需的部分。
- 当 processor 所需的地址未对齐时，user_addr[1:0] 可能为非零。这没有任何意义，因为 logic 在写周期上的正确行为仅取决于 user_wstrb。因此，即使对于未对齐的访问，这两位也最好忽略。

2.4 Interrupts

Xillybus Lite IP core 公开一个输入信号 user_irq，它允许 application logic 将 hardware interrupts 发送到 processor。它被视为同步positive edge-triggered interrupt request 信号，即当该信号从一个clock cycle 到下一个clock cycle 由低变高时，产生一个interrupt。

该信号在 xillydemo.v(hd) 模块中保持为零。

Xillybus Lite 采用 UIO 处理 interrupts 的方法： user space 程序在尝试从 device file 读取数据时休眠。当 interrupt 到达时，读取四个字节的数据，唤醒进程。这四个字节应被视为 unsigned int，其值为自加载 driver 以来已触发的 interrupts 总数。程序可能会忽略此值，或使用它来检查是否错过了 interrupts，方法是验证该值是否为先前读取的值的 1 加。

请注意，在正常系统操作中，此 interrupt counter 永远不会归零。

例如，假设 “fd” 是 /dev/uio0 的文件句柄：

```
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);
}
```

请注意，read() 函数调用必须需要 4 个字节。任何其他长度参数都将返回错误。interrupt 文件描述符可用于 select() 函数调用。

另请注意，检查 EINTR 的部分可以正确处理 software interrupts（例如，进程被停止和重新启动）并且与 hardware interrupt 无关。

3

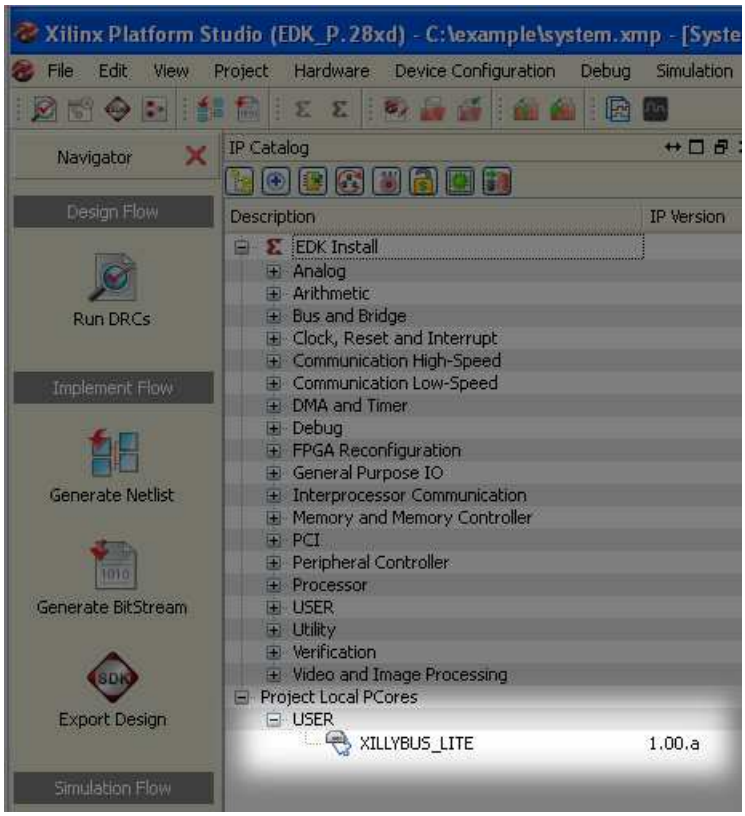
不使用 Xillinux 的项目上的 Xillybus Lite

3.1 应用 IP core

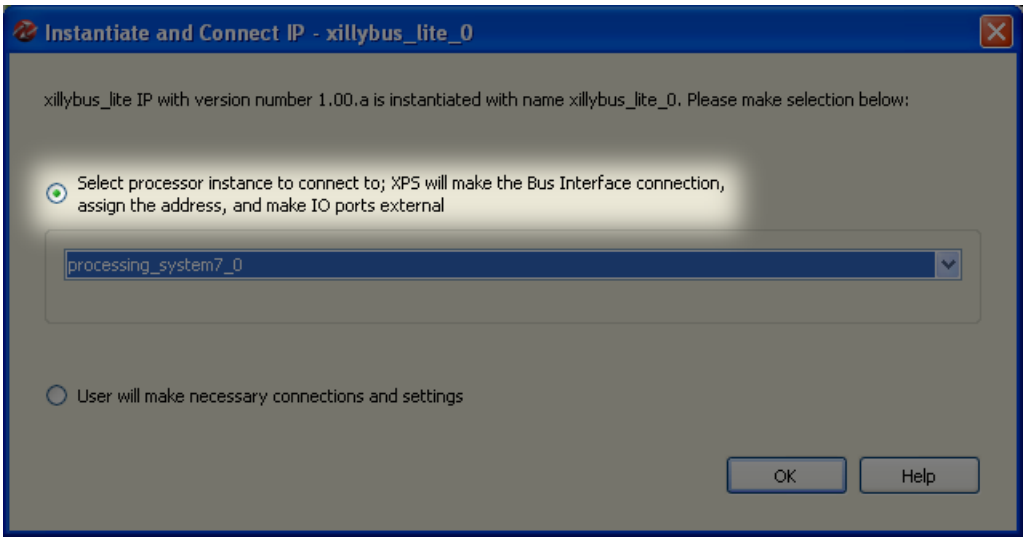
接下来描述的过程基于 Xilinx Platform Studio 14.2 (XPS)，但预计后续版本的行为大致相同。

要将 Xillybus Lite 添加到现有项目，请执行以下步骤：

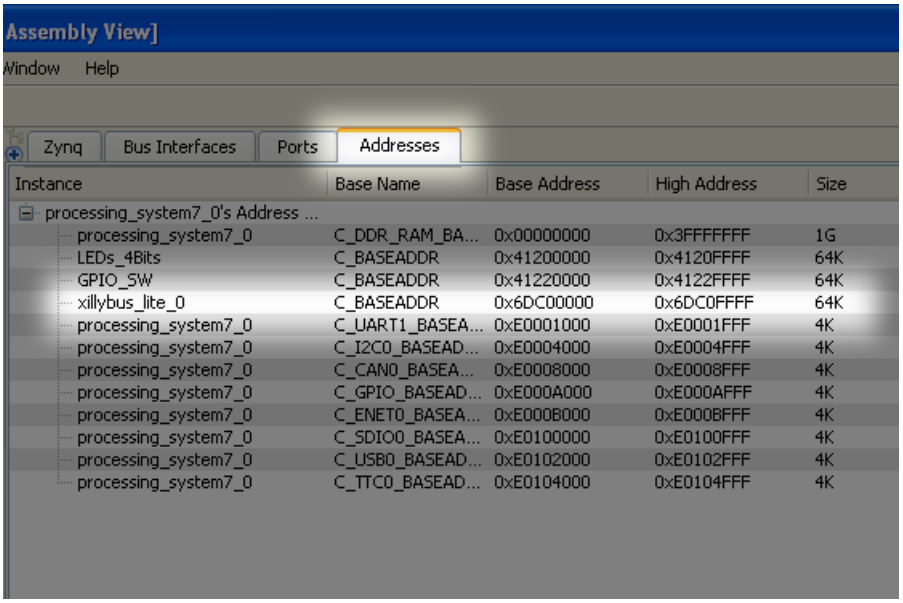
- 从 http://xillybus.com/xillybus-lite 下载 Xillybus Lite 包。
- 将 xillybus_lite_v1_00_a 文件夹从 Xillybus Lite 捆绑包的 “pcores” 文件夹复制到 XPS 项目的 “pcores” 文件夹中。如果 XPS 项目刚刚生成，则后者可能为空。
- 在 XPS 中打开相关项目后，单击 Project > Rescan User Repositories。因此，“USER” 条目将出现在左侧 “Project Local PCores” 下的 IP 目录中。展开此条目，找到 XILLYBUS_LITE core。



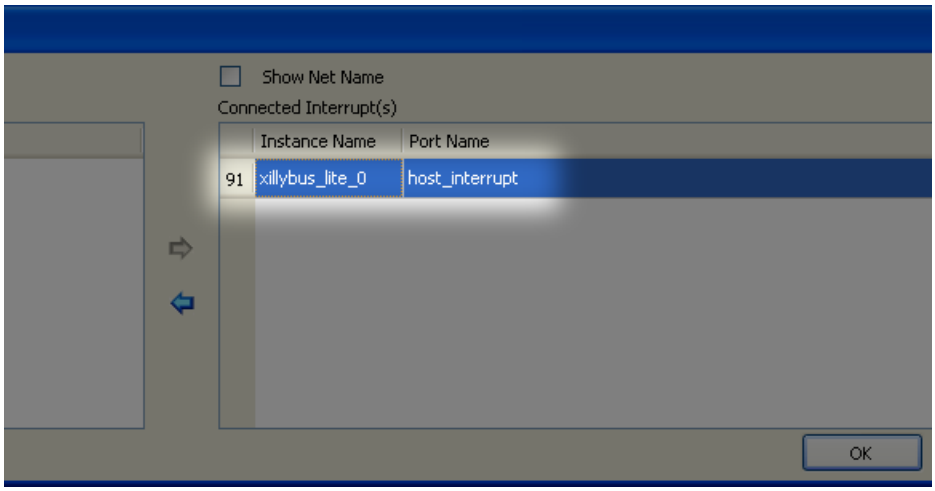
- 双击 XILLYBUS_LITE。确认弹出窗口询问是否应将 IP core 添加到 design。
- 接下来会出现一个 XPS Core Config 窗口。只需单击 “OK”。没有必要进行更改。
- 在以下窗口中，“Instantiate and Connect IP” 允许 XPS 通过选择上方的单选按钮进行 bus 接口连接。



- 选择 “Addresses” 选项卡并记下为 xillybus_lite_0 分配的地址范围以供将来参考（基地址和大小）。也可以根据需要更改此范围。

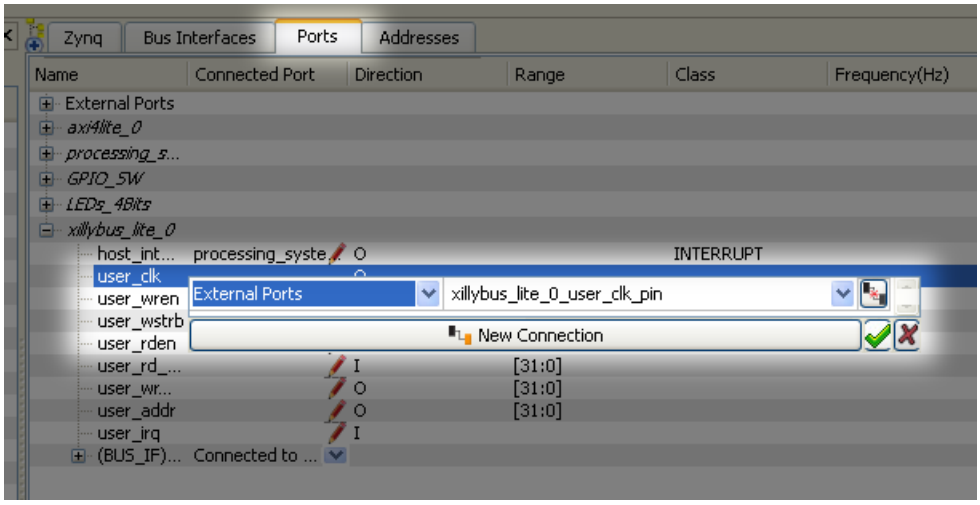


- 选择 “Zynq” 选项卡并单击 IRQ 框。在未连接的 interrupts 列表中选择 “host_interrupt” 端口，然后单击中间的箭头将其移动到已连接的 interrupts 列表中。记下 interrupt number（示例屏幕截图中的 91）。



- 选择 “Ports” 选项卡并展开 “xillybus_lite_0” 条目。对于带有 “user_” 前缀的八个信号中的每一个，单击端口名称右侧的空白区域，并将端口设为外部：

  在 drop-down 选择框中选择 “External Port”，并接受给定的默认电线名称（或可能更改它）。通过单击复选标记或按每个端口的 ENTER 进行确认。



重要的:

host_interrupt 端口设置为 *EDGE_RISING*。请勿将其更改为 *level triggered*，否则 *interrupt* 可能会锁定系统。

XPS 项目现在可以构建了（例如 “Create Netlist”）。

processor 的模块（通常命名为 “system”）将有 8 个额外的端口。这些应该添加到此模块的实例化中（以及 VHDL 中的架构描述）。

例如，在 Verilog

```
...
  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 )
    );
```

除了 user_rd_data 和 user_irq 之外的所有信号都是 processor 的输出。

3.2 修改 device tree

必须获取现有系统的 device tree，以便添加 Xillybus Lite 的条目。重要的是从 device tree 开始生效，否则系统的配置可能会在 boot 过程中发生变化甚至可能失败。

对于 Xillinux 2.0 及更高版本，使用的 device tree 源是 kernel 源的一部分，可以从 Github 下载。请参阅 Getting started with Xillinux for Zynq-7000 中的第 6 节了解如何获取它。

在 Xillinux 的早期版本中，device tree 源代码位于 /boot 目录中，例如 devicetree-3.3.0-xillinux-1.1.dts。

如果 device tree 源不可用，则可以从其二进制文件中重建它，该二进制文件位于启动时加载 boot.bin 的同一目录中。可以在以下位置找到解释此问题（以及与 device tree 相关的其他问题）的教程

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

应将如下条目添加到 DTS 文件中，在包含 bus 外围设备的段中（在包含 `axi@0` 的大括号内）：

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

重要的:

此示例条目与上面显示的屏幕截图匹配，而不是 *Xillinux* 中使用的设置。

DTS 条目中可能需要进行以下更改，以匹配您在 XPS 项目中的外设实例：

- 在 “reg” 分配和节点名称中设置从 XPS 的地址映射中获取的基地址（上面的 0x6dc00000，节点名称中没有 “0x” 前缀）。
- 将 “reg” 的第二个参数设置为从基地址分配的字节数（上面的 0x10000）
- 将 “interrupts” 的第二个参数设置为 XPS 中给定的 interrupt number 减去 32。在示例中，XPS 将 interrupt 91 分配给外围设备，因此分配给 91 - 32 = 59。
- 如果 device tree 是通过二进制文件或 /proc/device-tree/ 中的 reverse compilation 获得的，则不会定义 “gic” 标签。由于系统中的所有设备都使用相同的 interrupt controller，因此可以从 device tree 中的其他 “interrupt-parent” 分配中复制数值。在几乎所有情况下，这仅仅意味着将 `&gic` 替换为值 0x1。

编辑 device tree 源文件后，运行 compilation 以使用 Device Tree Compiler (DTC) 将其转换为二进制 blob (DTB)。此 compiler 是 Linux kernel tree 的一部分。

在运行的 Xillinux 系统上，可以按如下方式完成。 kernel 树的目录可能会有所不同，具体取决于所使用的 Xillinux 发行版。

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

然后用刚刚生成的文件覆盖用于 boot 的媒体上现有的 devicetree.dtb 文件。

请注意，Xillinux 可用于 device tree 的 compilation，该 device tree 用于 Xillinux 以外的系统。

3.3 Linux driver 的 Compilation

driver 可以在 “linuxdriver” 目录中的 Xillybus Lite 包中找到。

compilation 有两种选择：

- cross compilation 用于 ARM processor 与预期的 kernel 的 source code （或至少其 headers ）。
- 如果预期的发行版安装了 GNU tools 和 kernel headers，则可以在 Zynq 板上直接使用 compilation。

Xillybus lite 依赖于几个 kernel 选项，特别是 UIO 选项 (CONFIG_UIO) 至少作为一个模块启用。

在下文中，假设板上有一个直接的 compilation。这可以在 Xillinux 发行版中完成，但对于在 Xillinux 上运行 Xillybus Lite 不是必需的（因为 driver 已经安装）。另一方面，如果 driver 的 compilation 已经在 Xillinux 上完成（因此针对 Xillinux 的 kernel headers ），则 binary 可能无法在其他 kernels 上工作。

首先更改目录：

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

并将 “make” 键入 driver 的 compilation。会话应如下所示：

```
$ 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'
```

注意kernel module的compilation是专门为compilation期间运行的kernel做的。如果使用另一个 kernel，请键入 “make TARGET=kernel-version”，其中 “kernel-version” 是预期的 kernel 版本，如 /lib/modules/ 中所示。

会话的输出可能略有不同，但不会出现错误或警告。

特别是，如果出现这些警告，

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

则意味着预期的 kernel 缺少 UIO 选项，将 driver 插入 kernel 很可能会失败。

3.4 安装 driver

将 xillybus_lite_of.ko 目录复制到某个现有的 driver 子目录中，并按如下方式运行 depmod（假设当前正在运行的预期 kernel）：

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

安装不会立即将 driver 加载到 kernel 中。如果发现 Xillybus Lite 外围设备，它将在系统的下一个 boot 上执行此操作。接下来显示如何手动加载 driver。

3.5 装载和卸载 driver

为了加载 driver（并开始使用 Xillybus Lite），输入 root：

```
# modprobe xillybus_lite_of
```

这将使 Xillybus Lite device file (/dev/uio0) 出现。

请注意，如果在系统执行 boot 并且 driver 已按上述方式安装时存在 Xillybus Lite 外围设备，则不需要这样做。

要查看 kernel 中的模块列表，请键入 “lsmod”。要从 kernel 中移除 driver，请执行以下操作：

```
# rmmod xillybus_lite_of
```

这将使 device file 消失。

如果出现问题，请检查 /var/log/syslog 日志文件中包含 “xillybus” 字样的消息。在这个日志文件中经常可以找到有价值的线索。

如果不存在 /var/log/syslog 日志文件，则可能是 /var/log/messages。

如果日志文件中出现有关 __uio_register_device 或类似内容的 “unknown symbol” 错误，则表明正在运行的 kernel 缺少 UIO 配置选项。