(기계로 한국어 번역)

The guide to Xillybus Lite

Xillybus Ltd.

www.xillybus.com

Version 3.0

이 문서는 영어에서 컴퓨터에 의해 자동으로 번역되었으므로 언어가 불분명할 수 있습니다. 이 문서는 원본에 비해 약간 오래되었을 수 있습니다.

가능하면 영문 문서를 참고하시기 바랍니다.

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 This document may also be slightly outdated in relation to the original.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

소개

1.1 일반적인

Xillybus Lite는 Linux에서 실행되는 user space 프로그램에 의해 logic fabric(PL)의 registers에 쉽게 액세스할 수 있는 간단한 키트입니다. 소프트웨어에 bare-metal 환경의 환상을 제공하고 logic design에 주소, 데이터 및 read/write-enable 신호의 사소한 인터페이스를 제공합니다.

이 키트를 사용하면 개발 팀이 AXI bus 인터페이스와 Linux kernel 프로그래밍을 처리할 필요가 없으며 운영 체제나 bus 프로토콜에 대한 지식 없이도 주변 장치를 메모리와 같이 직접 제어할 수 있습니다.

키트는 IP core와 Linux driver로 구성됩니다. 이들은 Zedboard용 Xillinux 배포판(버전 1.1 이상)에 포함되어 있으며 프로젝트에 포함하기 위해 별도로 다운로드할 수도 있습니다.

Xillybus Lite는 DMA 기능을 포함하지 않습니다. 최대 데이터 속도는 약 28 MB/s입니다(초당 7백만 32비트 읽기 또는 쓰기 액세스, processor clock은 666 MHz).

Xillybus Lite IP core는 모든 용도로 무료로 출시됩니다. 특히, 추가 동의나 특정 라이선스 없이 상업적 목적으로 사용 및 판매되는 바이너리에 다운로드, 복사 및 포함될 수 있습니다.

Linux용 Xillybus Lite driver는 GPLv2로 출시되어 Linux kernel 자체와 동일한 조건으로 무료 배포됩니다.

1.2 Xillybus Lite 얻기

Xillybus Lite에 대해 배우고 사용해 보려면 Xillinux(버전 1.1 이상)를 다운로드하여 설치하는 것이 좋습니다.

배포판에는 xillydemo.v/vhd에서 쉽게 수정할 수 있는 샘플 logic에서 Xillybus Lite를 시험해 볼 수 있도록 모든 것이 설정되어 있습니다. Linux 쪽에는 driver가 이미 설치되어 있고

시작할 몇 가지 샘플 프로그램이 있습니다.

Xillinux는 http://xillybus.com/xillinux에서 다운로드할 수 있습니다.

기존 설치가 충분히 최신인지 확인하기 위해 Xillinux의 shell prompt에서 다음 검사를 실행할 수 있습니다.

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

접미사(위의 예에서 “1.1”)는 Xillinux 버전(이 경우 정상)을 나타냅니다.

Xillybus Lite는 Xillinux에 관계없이 모든 Zynq-7000 design에 포함될 수 있습니다. 이를 위해서는 3 섹션에 설명된 대로 XPS 프로젝트, 일부 배선, compilation 및 driver 설치에 IP core를 포함해야 합니다.

Xillybus Lite 번들은 http://xillybus.com/xillybus-lite에서 다운로드할 수 있습니다.

2

용법

2.1 샘플 design

연결된 IP core, 사전 설치된 Linux driver 및 몇 가지 간단한 데모 user space 프로그램으로 구성된 샘플 design이 Xillinux(버전 1.1 이상)에 포함되어 있습니다.

logic 측에서 xillydemo.v(hd) 모듈 소스 파일에는 어레이에서 추론된 32x32 bit RAM 구현이 포함되어 있습니다. 이 RAM은 host의 샘플 프로그램에서 액세스됩니다. compilation 및 실행은 다음과 같이 Xillinux에서 직접 수행할 수 있습니다.

```
# 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 소스는 /usr/src/xillinux/xillybus-lite/(버전 1.1 이상)의 Xillinux 파일 시스템에서 찾을 수 있습니다.

“uiotest” 프로그램은 register 배열의 처음 32비트 요소에 4개의 값을 쓴 다음 해당 값을 다시 읽고 인쇄하지만 더 유용한 것으로 쉽게 변경됩니다.

“intdemo” 프로그램은 interrupts가 어떻게 처리되는지 보여줍니다. 샘플 logic은 interrupts를 트리거하지 않으므로 그대로 실행하는 것은 의미가 없습니다. 그럼에도 불구하고 interrupts가 어떻게 기다리고 있는지 보여줍니다.

2.2 Host application와의 인터페이스

Xillybus Lite는 Linux의 사용자 I/O 인터페이스(UIO)를 기반으로 하며, 이는 주변기기를 주로 메모리 매핑에 의해 액세스되는 device file로 나타냅니다. 액세스 권한을 얻으려면 다음 코드가 적용됩니다.

```
#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);
}
```

오류 검사를 제외하고 이 코드 조각은 두 가지 작업을 수행합니다.

- open() 함수를 호출하여 device file을 엽니다(파일 핸들 가져오기).
- mmap() 함수를 호출하여 장치에 액세스하기 위한 주소를 얻습니다. 두 번째 매개변수(“size”)는 매핑되는 바이트 수입니다. device tree(수정되지 않은 Xillinux 의 4096)에 따라 주변 장치에 할당된 바이트 수를 초과해서는 안 됩니다.

map_addr는 프로세스의 virtual memory space에 있는 주소이지만 모든 목적을 위해 주변 장치가 bare-metal 환경(즉, 운영 체제 없음)에서 매핑되는 physical address인 것처럼 취급될 수 있습니다.

허용된 액세스 범위는 mem_addr에서 mem_addr + size - 1까지이며, 여기서 “size”는 mmap()에 지정된 두 번째 인수입니다. 이 범위를 넘어 메모리에 액세스하려고 하면 segmentation fault가 발생할 수 있습니다.

가까운 주소에서 주변 장치의 기본 주소(오프셋 0)에서 register에 32비트 워드를 쓰고 읽는 것은 다음과 같습니다.

```
volatile unsigned int *pointer = map_addr;

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

특정 메모리 영역에서 memory caching은 Linux driver에 의해 비활성화되고 pointer는 volatile에 플래그가 지정됩니다. 따라서 프로그램의 각 읽기 및 쓰기 작업은 bus 작업을 트리거하고 결과적으로 Xillybus Lite의 logic 인터페이스 신호에 대한 액세스 주기를 트리거합니다.

중요한:

위의 예와 같이 pointer는 "volatile" 키워드를 사용하여 volatile로 플래그가 지정되어야 합니다. 이 플래그가 없으면 C compiler가 I/O 작업을 재정렬하고 최적화할 수 있습니다.

logic이 바이트 단위 액세스를 지원한다면 8비트 volatile char pointer 또는 16비트 volatile short int pointer로 주변 장치에 액세스하는 것도 괜찮습니다.

위의 예에서는 Xillybus Lite 주변 장치가 하나만 있다고 가정합니다. 따라서 첫 번째 인스턴스인 "/dev/uio0"이 열립니다. 추가 UIO 장치가 있는 경우(예: Xillybus Lite 인스턴스가 두 개 이상인 경우) /dev/uio1, /dev/uio2 등으로 표시됩니다.

어떤 device file이 어떤 logic 요소에 속하는지 알기 위해 응용 프로그램은 /sys/class/uio/(예: /sys/class/uio/uio0/name 또는 /sys/class/uio/uio0/maps/map0/addr)에서 정보를 얻어야 합니다. UIO 장치가 여러 개 생성될 때 device files의 일관된 이름을 지정하려면 udev 프레임워크를 사용하는 것이 좋습니다.

2.3 logic design와의 인터페이스

2.3.1 Register 관련 신호

Xillybus Lite IP core는 Verilog 형식으로 제공되는 application logic에 7개의 신호를 제공합니다.

```
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;
```

인터페이스는 동기식이며 Xillybus Lite에서 제공하는 user_clk을 기반으로 합니다(processor의 AXI Lite clock에 연결됨).

위의 신호 이름은 Xillydemo 모듈(Xillinux 번들의 일부)에 나타나는 이름입니다. processor 모듈의 신호 이름은 약간 다릅니다. 예를 들어 user_wren은 xillybus_lite_0_user_wren_pin로 나타날 수 있습니다.

이러한 신호는 표준 block RAM에 직접 연결할 수 있으며, 이 경우 host는 해당 RAM에 직접 액세스할 수 있습니다(dual-port RAM이 선택된 경우 “mailbox”로 사용할 수 있음). 또한 아래에 설명된 대로 logic에 정의된 registers에 연결할 수도 있습니다.

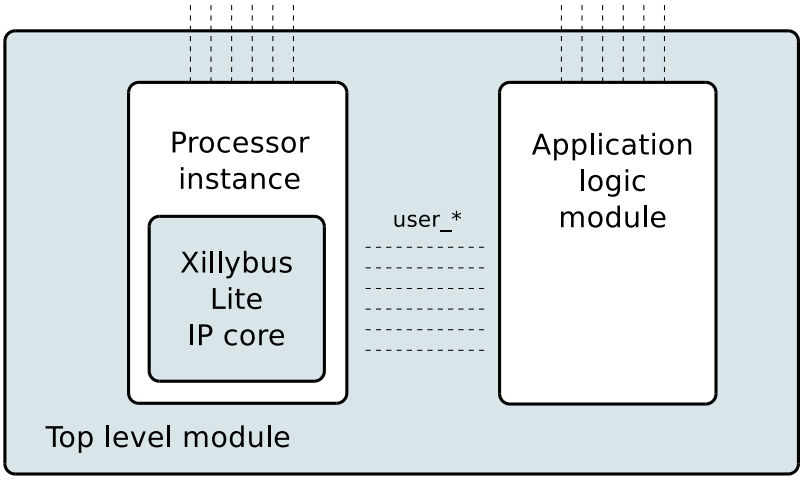
2.3.2 모듈 계층

Xilinx logic design이 embedded processor를 포함하는 경우 일반적으로 top level module에서 인스턴스화되는 이를 나타내는 모듈이 있습니다. 일반적으로 processor가 사물의 중심이고 주변의 모든 logic이 일종의 주변 장치라는 패러다임에 따라 이 모듈에 의해 노출된 포트는 모두 물리적 핀에 직접 연결됩니다.

Xillybus Lite는 application logic의 상당 부분과 인터페이스하기 위한 것이므로 이 공통 구조를 다소 깨뜨립니다. user_* 신호는 top level module로 라우팅하기 위한 것이므로 사용자 지정 logic도 해당 top level module에서 인스턴스화됩니다. 전체 프로젝트의 구조는 processor와 IP cores(Xillybus Lite IP core 포함)를 포함하는 instantiated module와 application logic이 포함된 두 번째 모듈이라는 두 개의 큰 덩어리로 끝납니다. user_* 신호는 둘 사이를 연결합니다.

따라서 Xillybus Lite IP core 자체가 processor의 계층 내부 깊숙한 곳에서 Xilinx의 도구에 의해 인스턴스화되더라도 top level module에서 인터페이스됩니다.

이것은 Xillinux용 demo bundle에서 선택한 레이아웃이며(아래 그림 참조) 이 가이드에서 가정하는 것이기도 합니다. processor의 계층 구조 내에서 Xillybus Lite의 신호를 내부적으로 연결하는 것이 가능하지만 반드시 일을 더 단순하게 만들지는 않습니다.



2.3.3 32비트 정렬 register 액세스

logic(아래 “litearray”)에서 32x32 bit 어레이에 액세스하려면 다음과 같은 코드를 사용할 수 있습니다. 이것은 host가 32비트 워드 액세스를 고수하는 경우에만 잘 작동합니다(예: unsigned int에만 pointers 사용):

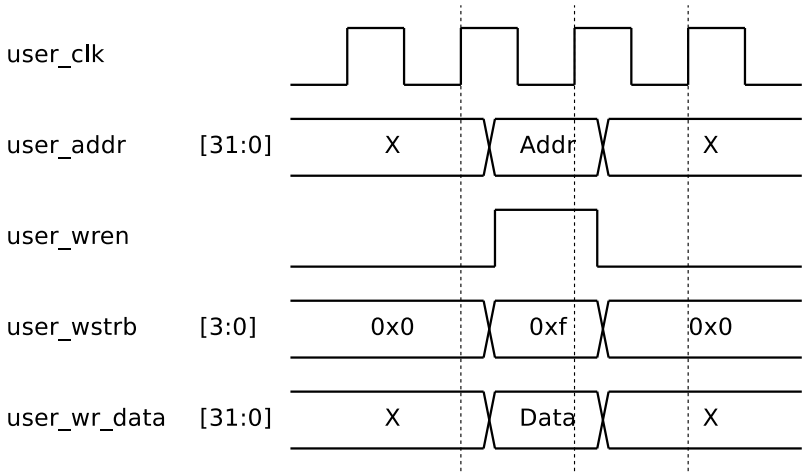
Verilog에서:

```
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
```

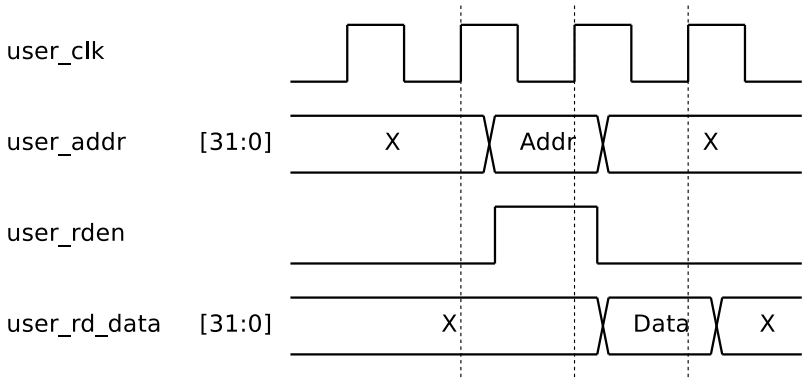
또는 VHDL:

```
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;
```

정렬된 쓰기 주기 및 읽기 주기의 파형은 다음과 같습니다.



Waveform 1: Write cycle for aligned access



Waveform 2: Read cycle

참고:

- XPS에서 Xillybus Lite 주변기기에 할당된 주소 영역에 대한 bus 작업은 항상 정확히 하나의 clock cycle에 대해 user_wren 또는 user_rden이 하이가 됩니다.
- user_rd_data는 user_rden이 하이 후에 clock cycle 하나만 Xillybus Lite core에 의해 감지됩니다. 따라서 user_rden을 모니터링할 실제적인 필요가 없습니다. 예를 들어, user_addr(하나의 clock의 latency 포함)에 따라 항상 user_rd_data를 업데이트하는 것도 좋습니다.

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

- 위의 코드는 32개 요소의 32비트 너비 배열에 대한 액세스를 보여줍니다. 더 일반적인 설정은 주소 오프셋 0x14에서 “myregister”를 매핑하기 위해 Verilog

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

  에서 registers에 액세스하는 것입니다.

- 마찬가지로 user_addr에 의존하는 case statement는 다음과 같은 user_rd_data 값 할당의 일반적인 구현입니다.

```
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_addr는 32비트 너비이며 액세스되는 전체 물리적 주소를 보유합니다. enable 신호는 주소가 할당된 범위 내에 있을 때만 High이므로 주소의 MSBs를 확인할 필요가 없습니다.
- 항상 user_addr[1:0]를 무시합니다. 이 두 LSBs는 32비트 정렬 bus 액세스에서 항상 0이며 아래 설명된 대로 정렬되지 않은 액세스의 경우에도 무시해야 합니다.

2.3.4 정렬되지 않은 register 액세스

host가 32비트 정렬되지 않은 방식으로 register 공간에 액세스할 가능성이 있는 경우 logic에서 각 바이트를 별도로 처리해야 합니다.

bus에서 바이트와 32비트 워드에 액세스하는 데는 동일한 시간이 걸리므로 정렬되지 않은 액세스는 대역폭이 4배 비효율적입니다.

litearray3, litearray2, litearray1 및 litearray0이 각각 8비트를 갖는 32개 요소의 메모리 어레이라고 가정합니다. 다음 코드 조각은 정렬되지 않은 액세스를 지원하기 위해 2.3.3의 예

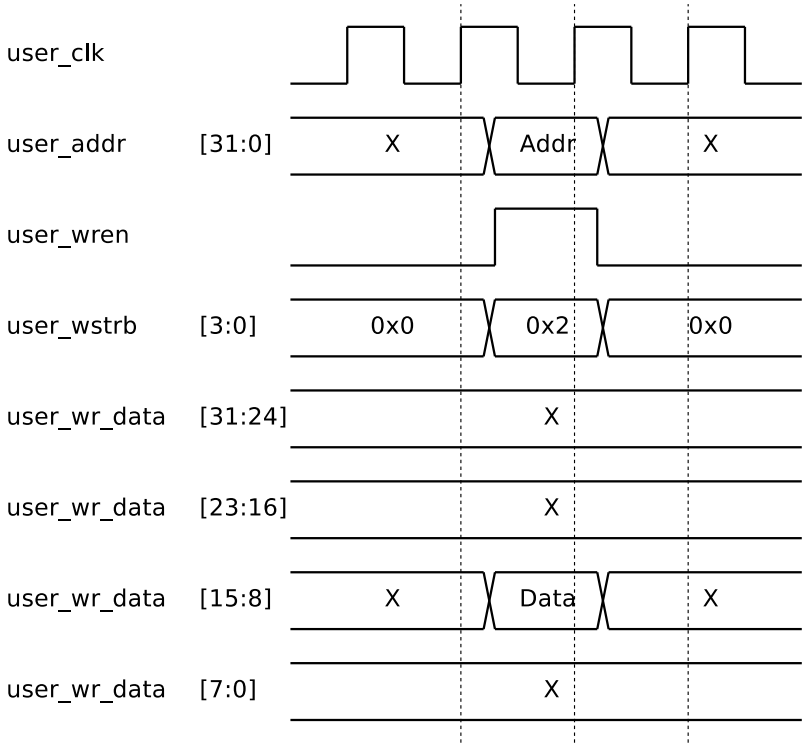
제를 다시 작성하는 방법을 보여줍니다. Verilog에서:

```
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
```

또는 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;
```

기본 주소에서 0x01 오프셋이 있는 단일 바이트의 정렬되지 않은 쓰기 주기에 대한 파형은 다음과 같습니다.



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

참고:

- 할당된 주소 영역에 대한 쓰기 bus 작업은 항상 하나의 clock cycle에 대해 user_wren와 user_wstrb의 비트 중 하나 이상이 동시에 하이가 되도록 합니다. 위와 같이 값 할당이 user_wstrb에 의존한다면 user_wren을 확인할 필요가 없습니다.
- 정렬되지 않은 읽기 액세스는 logic에서 정렬된 액세스와 동일하게 처리됩니다. 예를 들어, processor에서 실행되는 프로그램이 바이트를 읽을 때 전체 32비트 워드를 bus에서 읽고 processor는 워드에서 필요한 부분을 선택합니다.
- user_addr[1:0]은 processor에 필요한 주소가 정렬되지 않은 경우 0이 아닐 수 있습니다. 쓰기 주기에 대한 logic의 올바른 동작은 user_wstrb에만 의존하기 때문에 이것은 의미가 없습니다. 따라서 이 두 비트는 정렬되지 않은 액세스의 경우에도 가장 잘 무시됩니다.

2.4 Interrupts

Xillybus Lite IP core는 입력 신호 user_irq를 노출하여 application logic이 hardware interrupts를 processor로 보낼 수 있도록 합니다. 이것은 동기식 positive edge-triggered interrupt request 신호로 처리됩니다. 즉, 이 신호가 한 clock cycle에서 다음 clock cycle로 로우에서 하이로 변경될 때 interrupt가 생성됩니다.

이 신호는 xillydemo.v(hd) 모듈에서 0으로 유지됩니다.

Xillybus Lite는 interrupts를 처리하는 UIO의 방법을 채택합니다. user space 프로그램은 device file에서 데이터 읽기를 시도할 때 잠자기 상태입니다. interrupt가 도착하면 4바이트의 데이터를 읽고 프로세스를 깨웁니다. 이 4바이트는 driver가 로드된 이후 트리거된 interrupts의 총 수 값을 갖는 unsigned int로 처리되어야 합니다. 프로그램은 이 값을 무시하거나 값이 이전에 읽은 값에 1을 더한 값인지 확인하여 interrupts가 누락되었는지 확인하는 데 사용할 수 있습니다.

정상적인 시스템 작동에서 이 interrupt counter는 절대로 0이 되지 않습니다.

예를 들어 “fd”가 /dev/uio0에 대한 파일 핸들이라고 가정합니다.

```
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() 함수 호출에는 4바이트가 필요합니다. 다른 길이 인수는 오류를 반환합니다. interrupt 파일 디스크립터는 select() 함수 호출에서 사용될 수 있습니다.

또한 EINTR에 대한 부품 검사는 software interrupts를 올바르게 처리하며(예: 프로세스가 중지되고 다시 시작됨) hardware interrupt와 아무 관련이 없습니다.

3

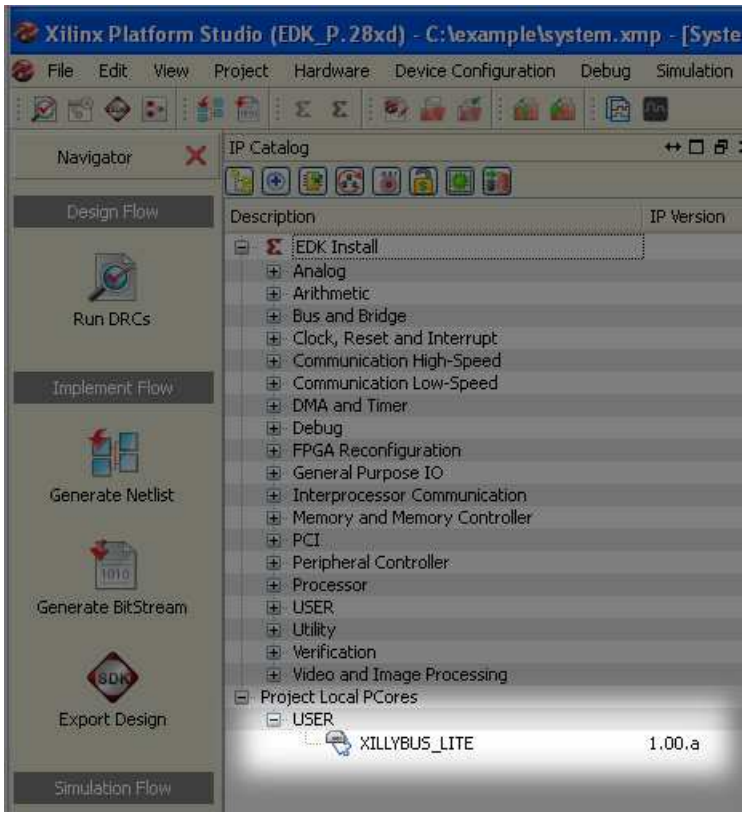
Xillinux를 사용하지 않는 프로젝트의 Xillybus Lite

3.1 IP core 적용

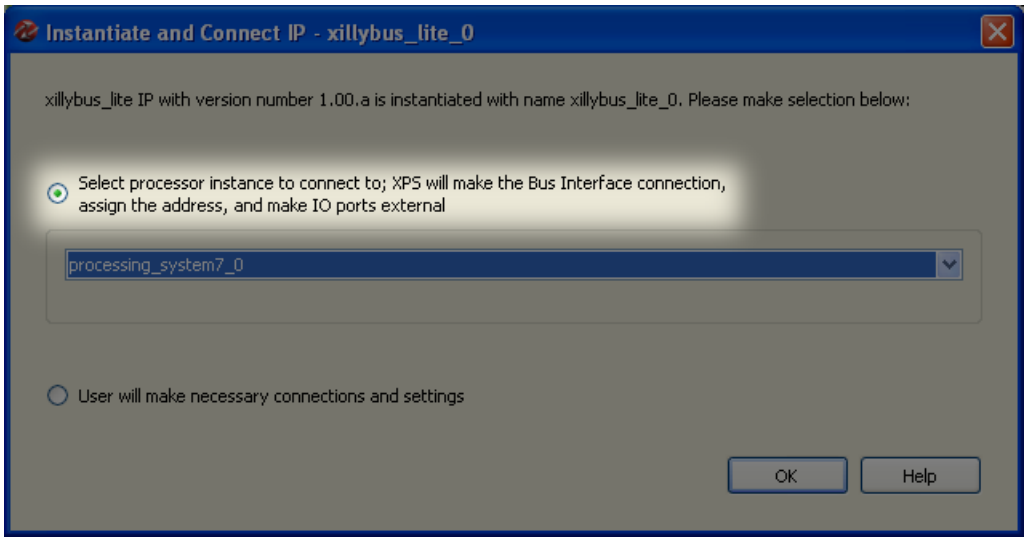
다음에 설명하는 절차는 Xilinx Platform Studio 14.2 (XPS)를 기반으로 하지만 이후 버전에서도 거의 동일하게 작동할 것으로 예상됩니다.

기존 프로젝트에 Xillybus Lite를 추가하려면 다음 단계를 따르십시오.

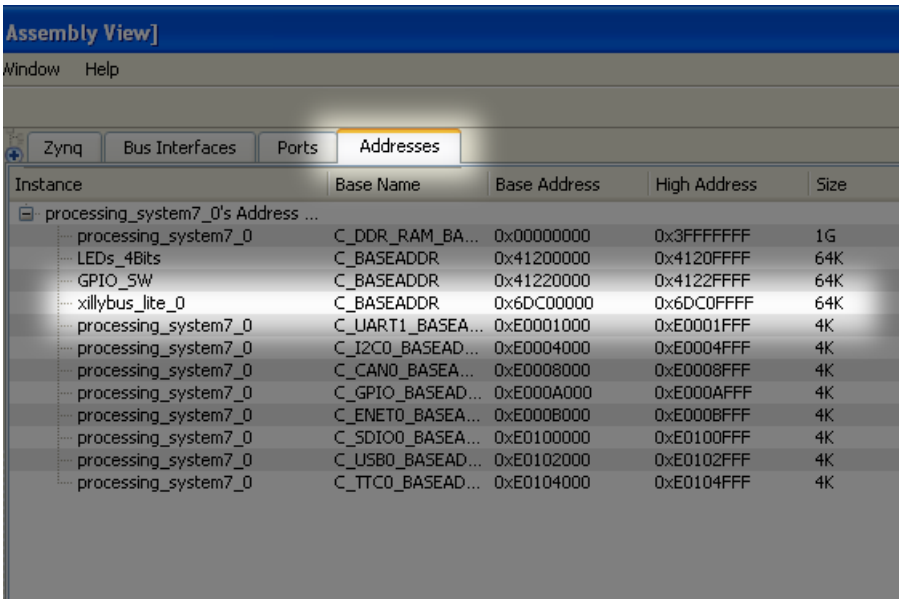
- http://xillybus.com/xillybus-lite에서 Xillybus Lite 번들을 다운로드합니다.
- Xillybus Lite 번들의 “pcores” 폴더에서 xillybus_lite_v1_00_a 폴더를 XPS 프로젝트의 “pcores” 폴더로 복사합니다. XPS 프로젝트가 방금 생성된 경우 후자는 비어 있을 수 있습니다.
- XPS에서 해당 프로젝트를 연 상태에서 Project > Rescan User Repositories를 클릭합니다. 결과적으로 “USER” 항목은 IP 카탈로그 왼쪽의 “Project Local PCores” 아래에 나타납니다. 이 항목을 확장하고 XILLYBUS_LITE core를 찾으십시오.



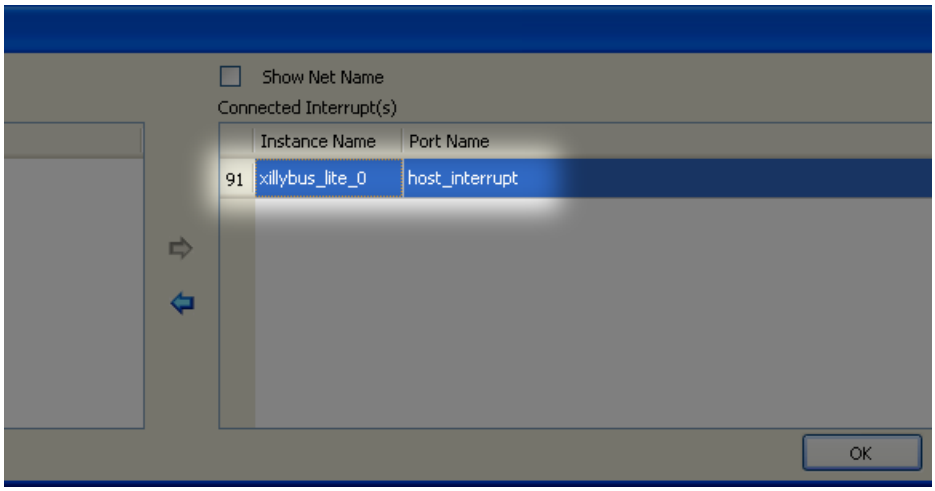
- XILLYBUS_LITE를 두 번 클릭합니다. IP core를 design에 추가해야 하는지 묻는 팝업 창을 확인합니다.
- XPS Core Config 창이 다음에 나타납니다. “OK”를 클릭하기만 하면 됩니다. 변경할 필요가 없습니다.
- 다음 창에서 “Instantiate and Connect IP”는 상단 라디오 버튼을 선택하여 XPS가 bus 인터페이스 연결을 만들 수 있도록 합니다.



- “Addresses” 탭을 선택하고 나중에 참조할 수 있도록 xillybus_lite_0에 할당된 주소 범위(기본 주소 및 크기)를 기록해 둡니다. 필요에 따라 이 범위를 변경할 수도 있습니다.

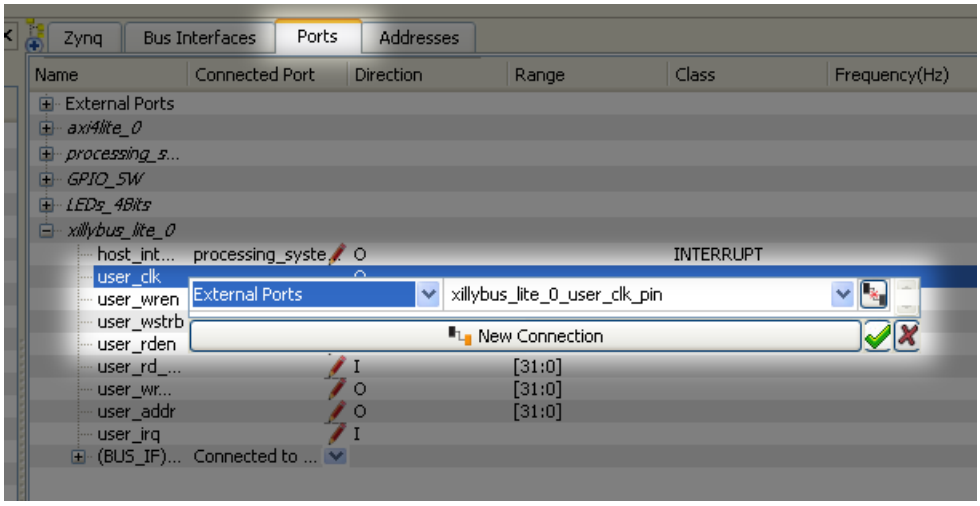


- “Zynq” 탭을 선택하고 IRQ 상자를 클릭합니다. 연결되지 않은 interrupts 목록에서 “host_interrupt” 포트를 선택하고 가운데 화살표를 클릭하여 연결된 interrupts 목록으로 이동합니다. interrupt number(샘플 스크린샷의 91)를 기록해 둡니다.



- “Ports” 탭을 선택하고 “xillybus_lite_0” 항목을 확장합니다. “user_” 접두사가 있는 8개 신호 각각에 대해 포트 이름 오른쪽의 빈 공간을 클릭하고 포트를 외부로 만듭니다.

  drop-down 선택 상자에서 “External Port”를 선택하고 주어진 기본 와이어 이름을 수락합니다(또는 변경 가능). 확인 표시를 클릭하거나 각 포트에 대해 ENTER를 눌러 확인합니다.



중요한:

host_interrupt 포트는 EDGE_RISING로 설정됩니다. level triggered로 변경하지 마십시오. 그렇지 않으면 interrupt가 시스템을 잠글 수 있습니다.

이제 XPS 프로젝트를 빌드할 준비가 되었습니다(예: “Create Netlist”).

processor의 모듈(일반적으로 “system” 라고 함)에는 8개의 추가 포트가 있습니다. 이것들은 이 모듈의 인스턴스화(및 VHDL의 아키텍처 설명)에 추가되어야 합니다.

예를 들어 Verilog 에서

```
...
  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 및 user_irq를 제외한 모든 신호는 processor에서 출력됩니다.

3.2 device tree 수정

기존 시스템에 대한 device tree를 가져와야 Xillybus Lite에 대한 항목을 추가할 수 있습니다. 실제로 device tree에서 시작하는 것이 중요합니다. 그렇지 않으면 시스템 구성이 변경되거나 boot 프로세스에서 실패할 수도 있습니다.

Xillinux 2.0 이상의 경우 사용 중인 device tree 소스는 Github에서 다운로드할 수 있는

kernel 소스의 일부입니다. 이것을 얻는 방법에 대해서는 Getting started with Xillinux for Zynq-7000의 섹션 6을 참조하십시오.

Xillinux의 이전 버전에서 device tree 소스는 devicetree-3.3.0-xillinux-1.1.dts와 같은 /boot 디렉토리에 있습니다.

device tree 소스를 사용할 수 없는 경우 전원을 켤 때 boot.bin이 로드되는 동일한 디렉토리에서 사용할 수 있는 바이너리에서 재구성할 수 있습니다. 이 문제(및 device tree와 관련된 다른 문제)를 설명하는 자습서는 다음에서 찾을 수 있습니다.

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

DTS 파일에 bus 주변 장치를 포함하는 세그먼트(`axi@0`을 둘러싸는 중괄호 내)에 다음 항목을 추가해야 합니다.

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

중요한:

이 샘플 항목은 위에 표시된 스크린샷과 일치하며 Xillinux에서 사용된 설정과 일치하지 않습니다.

XPS 프로젝트의 주변 장치 인스턴스와 일치시키기 위해 DTS 항목에서 다음 변경 사항이 필요할 수 있습니다.

- XPS의 주소 맵에서 가져온 기본 주소를 “reg” 할당과 노드 이름에 설정합니다(위의 0x6dc00000, 노드 이름에 “0x” 접두사 제외).
- “reg”의 두 번째 인수를 기본 주소(위의 0x10000)에서 할당된 바이트 수로 설정합니다.
- “interrupts”의 두 번째 인수를 XPS에서 32를 뺀 interrupt number로 설정합니다. 이 예에서 XPS는 interrupt 91을 주변 장치에 할당했으며 결과적으로 91 - 32 = 59를 할당했습니다.
- device tree가 바이너리 또는 /proc/device-tree/에서 reverse compilation을 통해 얻은 경우 “gic” 레이블이 정의되지 않습니다. 시스템의 모든 장치가 동일한 interrupt controller를 사용하기 때문에 device tree의 다른 “interrupt-parent” 할당에서 숫자

값을 복사하는 것이 좋습니다. 거의 모든 경우에 이것은 단순히 `&gic`를 값 0x1로 바꾸는 것을 의미합니다.

device tree 소스 파일을 편집한 후 compilation을 실행하여 Device Tree Compiler (DTC)를 사용하여 바이너리 블롭(DTB)으로 변환합니다. 이 compiler는 Linux kernel tree의 일부입니다.

실행 중인 Xillinux 시스템에서는 다음과 같이 수행할 수 있습니다. kernel 트리의 디렉토리는 사용된 Xillinux 배포판에 따라 다를 수 있습니다.

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

그런 다음 방금 생성된 파일로 boot에 사용된 미디어의 기존 devicetree.dtb 파일을 덮어씁니다.

Xillinux는 Xillinux 이외의 시스템용인 device tree의 compilation에 사용할 수 있습니다.

3.3 Linux driver의 Compilation

driver는 “linuxdriver” 디렉토리의 Xillybus Lite 번들에서 찾을 수 있습니다.

compilation에는 두 가지 옵션이 있습니다.

- 의도한 kernel의 source code(또는 최소한 headers)에 대한 ARM processor용 cross compilation.
- 의도한 배포판에 GNU tools 및 kernel headers가 설치되어 있는 경우 Zynq 보드 자체에서 직접 compilation이 가능합니다.

Xillybus lite는 여러 kernel 옵션에 의존하며, 특히 UIO 옵션(CONFIG_UIO)은 최소한 모듈로 활성화됩니다.

다음 내용에서는 보드에 직접 compilation이 있다고 가정합니다. 이것은 Xillinux 배포판에서 수행할 수 있지만 Xillinux에서 Xillybus Lite를 실행하는 데는 필요하지 않습니다(driver가 이미 설치되어 있기 때문에). 반면에 driver의 compilation이 Xillinux에서 수행된 경우(따라서 Xillinux의 kernel headers에 대해) binary는 다른 kernels에서 작동하지 않을 수 있습니다.

첫 번째 변경 디렉토리:

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

driver의 compilation에 대해 “make”을 입력합니다. 세션은 다음과 같아야 합니다.

```
$ 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 module의 compilation은 compilation 중에 실행되는 kernel을 위해 특별히 수행되었습니다. 다른 kernel이 사용되는 경우 “make TARGET=kernel-version”을 입력합니다. 여기서 “kernel-version”은 /lib/modules/에 표시되는 kernel 버전입니다.

세션의 출력은 약간 다를 수 있지만 오류나 경고가 나타나지 않아야 합니다.

특히 이러한 경고가 나타나면

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

는 의도한 kernel에 UIO 옵션이 없고 driver를 kernel에 삽입하면 실패할 가능성이 높다는 의미입니다.

3.4 driver 설치

xillybus_lite_of.ko 디렉터리를 기존 driver 하위 디렉터리에 복사하고 다음과 같이 depmod를 실행합니다(원하는 kernel이 현재 실행 중이라고 가정).

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

설치는 driver를 kernel로 즉시 로드하지 않습니다. Xillybus Lite 주변 장치가 검색되면 시스템의 다음 boot에서 수행됩니다. driver를 수동으로 로드하는 방법은 다음과 같습니다.

3.5 driver 로드 및 언로드

driver를 로드하려면(그리고 Xillybus Lite 작업을 시작하려면) root로 입력하십시오.

```
# modprobe xillybus_lite_of
```

그러면 Xillybus Lite device file(/dev/uio0)가 나타납니다.

시스템이 boot를 수행하고 driver가 위에서 설명한 대로 이미 설치되었을 때 Xillybus Lite 주변기기가 있는 경우에는 이것이 필요하지 않습니다.

kernel의 모듈 목록을 보려면 “lsmod”를 입력하십시오. kernel에서 driver를 제거하려면 다음으로 이동하십시오.

```
# rmmod xillybus_lite_of
```

이렇게 하면 device file이 사라집니다.

문제가 발생한 것 같으면 /var/log/syslog 로그 파일에서 “xillybus”라는 단어가 포함된 메시지를 확인하십시오. 종종 이 로그 파일에서 귀중한 단서를 찾을 수 있습니다.

/var/log/syslog 로그 파일이 없으면 아마도 /var/log/messages일 것입니다.

__uio_register_device 또는 이와 유사한 것과 관련된 “unknown symbol” 오류가 로그 파일에 나타나면 실행 중인 kernel에 UIO 구성 옵션이 없음을 나타냅니다.