

(기계로 한국어 번역)

---

## The guide to Xillybus Block Design Flow for non-HDL users (deprecated)

---

Xillybus Ltd.

[www.xillybus.com](http://www.xillybus.com)

Version 3.3

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 This document may also be slightly outdated in relation to the original.*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

Block Design Flow는 더 이상 사용할 수 없습니다.

이 문서는 기존 프로젝트만을 지원하기 위한 것입니다.

<b>1 소개</b>	<b>4</b>
<b>2 일반 지침</b>	<b>6</b>
2.1 Getting started . . . . .	6
2.2 block design의 주목할만한 요소 . . . . .	7
<b>3 application logic와 통합</b>	<b>9</b>
3.1 기본 . . . . .	9
3.2 Clocking . . . . .	10
3.2.1 일반적인 . . . . .	10
3.2.2 application clock 설정 . . . . .	10
3.2.3 bus_clk 신호 . . . . .	11
<b>4 가속화/코프로세싱 모범 사례</b>	<b>12</b>
4.1 처리량 대 latency . . . . .	12
4.2 데이터 폭 및 성능 . . . . .	12
4.3 해야 할 일과 하지 말아야 할 일 . . . . .	13
<b>5 커스텀 Xillybus IP core 적용하기</b>	<b>15</b>
<b>6 Vivado HLS 통합</b>	<b>18</b>
6.1 개요 . . . . .	18
6.2 HLS synthesis . . . . .	19
6.3 FPGA 프로젝트와 통합 . . . . .	19
6.4 예제 synthesis 코드 . . . . .	21
6.5 synthesis용 C/C++ 코드 수정 . . . . .	24
6.6 simple.c: host 프로그램의 예 . . . . .	25
6.7 practical.c: 실용적인 host 프로그램 . . . . .	28
6.8 Design 고려 사항 . . . . .	33
6.8.1 여러 AXI streams 작업 . . . . .	33
6.8.2 application clock의 주파수 . . . . .	34
6.8.3 logic 재설정 . . . . .	35

# 1

## 소개

---

Xillybus Block Design Flow는 Verilog / VHDL design flow의 대안으로 logic 관련 HDL 언어를 수정하고 디자인하는 데 익숙하지 않은 사람들을 위한 것입니다. 주요 목적은 FPGA에 대한 배경 지식이 없는 설계자가 FPGA 관련 기술을 습득할 필요 없이 coprocessing/acceleration 기능에 액세스할 수 있도록 하는 것입니다. 무엇보다도 AMD의 Vivado High Level Synthesis (HLS)에서 생성된 logic와 Linux 또는 Microsoft Windows를 실행하는 컴퓨터 또는 embedded 플랫폼 간에 데이터를 교환하는 간단한 수단으로 사용됩니다.

Block Design Flow는 Xillybus IP core를 통해 FPGA FIFOs와 통신한다는 Xillybus의 주요 개념에서 벗어나 있습니다. 대신 사용자 응용 프로그램 logic은 AXI Stream 인터페이스를 통해 Xillybus IP block에 직접 연결됩니다. 이것은 작업을 상당히 단순화하지만 차이점을 인식해야 합니다. 특히 Xillybus의 문서에서 FPGA의 FIFOs를 언급할 때 이것은 Block Design Flow와 관련이 없습니다. 각 FIFO 대신 Block Design의 GUI에 간단한 와이어가 있습니다.

Xillybus의 Block Design Flow는 Zynq processor 환경을 설정하거나 logic blocks 사이에 연결하는 데 사용되는 block design 다이어그램과 혼동되어서는 안 됩니다. 이러한 block designs는 적용되는 경우 관련이 없으며 Xillybus의 IP core를 application logic와 연결하기 위해 선택한 방법에 관계없이 공존할 수 있습니다.

Xillybus를 통해 설계자는 다음을 통해 생산적인 애플리케이션 관련 작업에 집중할 수 있습니다.

- compilation을 FPGA bitstream에 있는 그대로 사용할 준비가 된 작업 시작 프로젝트를 제공합니다. 이 프로젝트는 Xillybus의 IP core 덕분에 FPGA와 컴퓨터 host 간의 간단하고 직관적인 데이터 교환을 설정합니다.
- C/C++와 함께 logic design을 시연하기 위한 샘플 High Level Synthesis (HLS) 프로젝트를 제공, 이 가이드에 설명된 핵심 요소(섹션 6 참조),

- Vivado의 block design 도구를 사용하여 IP blocks를 FPGA design에 매우 간단하게 통합할 수 있습니다.
- host에서 간단한 프로그래밍 인터페이스를 제공하는 Linux 및 Windows용 drivers 공급,
- 주어진 프로젝트를 위해 특별히 구성된 data streams로 구성된 맞춤형 Xillybus IP cores를 자동으로 생성하는 웹 도구를 제공합니다.

Block Design Flow는 AMD의 Vivado의 block design 도구에 의존하므로 이 도구에서 다루는 FPGAs로 제한됩니다. 따라서 AMD의 series-7 FPGAs 이상(Ultrascale 장치 포함)만 지원됩니다.

Block Design Flow의 사용 편의성에도 불구하고 Xillybus 기능의 하위 집합에만 액세스할 수 있으므로 Verilog 또는 VHDL 기반 FPGA design에 익숙한 사용자에게는 권장되지 않습니다. 그러나 IP core 또는 HLS 기반 hardware acceleration/coprocessing와 같은 특정 응용 프로그램의 경우 Xillybus의 기능 차이가 미치는 영향은 무시할 수 있습니다.

Block Design Flow는 XillyUSB에서 지원되지 않습니다.

# 2

## 일반 지침

### 2.1 Getting started

원칙적으로 Block Design Flow에 대한 프로젝트 설정은 Vivado를 사용하여 의도한 플랫폼에 대한 해당 Getting Started 가이드에 설명된 대로입니다.

- Xilinx 번들의 경우: [Getting started with Xilinx for Zynq-7000](#)
- PCIe 번들의 경우: [Getting started with the FPGA demo bundle for AMD](#)

이 가이드를 따를 때 **blockdesign/** 하위 디렉토리에 있는 xillydemo-vivado.tcl script를 사용해야 합니다.

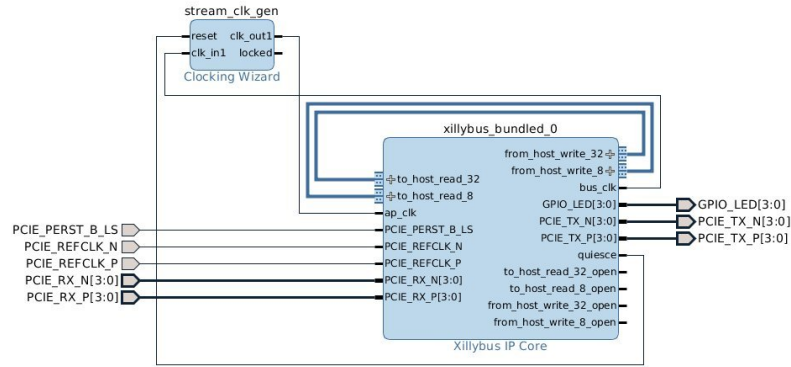
#### 중요한:

이 가이드는 Xillybus 웹사이트의 “FPGA coprocessing for C/C++ programmers”라는 튜토리얼과 함께 하지 **않습니다**. 기술적인 세부 사항과 제시된 예제 프로젝트에는 몇 가지 차이점이 있습니다. 혼란을 피하기 위해 이 가이드(Block Design Flow의 경우) 또는 웹사이트 튜토리얼(Verilog / VHDL design의 경우)을 고수하는 것이 좋습니다.

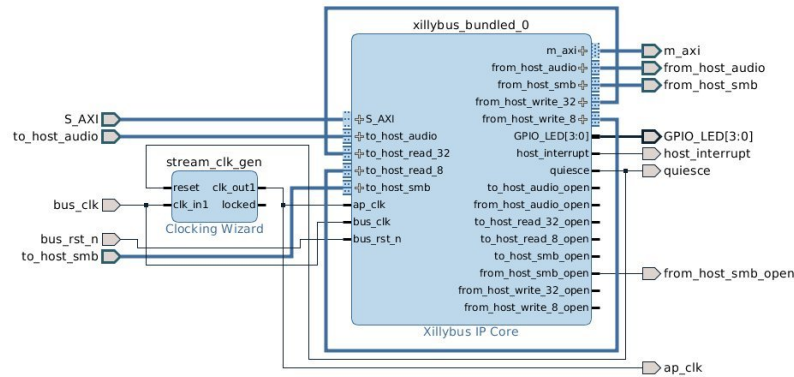
bitfile 생성 및 사용은 위에서 언급한 Getting Started 가이드와 동일하게 수행됩니다. bitfile은 번들 “out of the box”에서 즉시 생성할 수 있으며 이 가이드에 설명된 loopback 테스트는 동일하게 작동합니다. 그러나 4.3 섹션에서 설명한 것처럼 seekable stream xillybus\_mem\_8은 Block Design Flow에서 작동하지 않습니다.

Block Design Flow는 Xillybus의 IP core와 인터페이스가 Vivado의 block design 도구에서 발생한다는 점에서 다릅니다. 프로젝트를 생성한 후 Vivado의 왼쪽 메뉴 모음에서 “Open Block Design”을 선택하여 block design을 엽니다.

PCIe 기반 designs에서(즉, Xilinx 아님) 다음 다이어그램이 표시됩니다.



Xilinx를 사용할 때 “Open Block Design”은 Zynq processor의 환경을 엽니다. 이 block design 부분은 Xillybus와 관련된 작업에서 수정되어서는 안 됩니다. 대신 “blockdesign”로 표시된 block을 열어(더블 클릭으로) 다음 block diagram을 표시해야 합니다.



## 2.2 block design의 주목할만한 요소

Xillybus block design에서 주목할 가치가 있는 몇 가지 핵심 요소가 있습니다.

- Xillybus stream 포트 (“from\_host\_\*” 및 “to\_host\_\*”): TDATA, TVALID 및 TREADY의 최소 신호 세트로 구성된 표준 AXI Stream 포트입니다. block design의 각 포트 이름은 인터페이스의 방향을 표시하기 위해 “from\_host” 또는 “to\_host”가 앞에 옵니다. block design의 나머지 포트 이름은 host에 표시된 대로 device file의 이름에서 “xillybus” 접두사를 뺀 것입니다.

예를 들어, Linux host에서 /dev/xillybus\_write\_32라는 이름의 device file 또는 Windows 컴퓨터에서 \\.\xillybus\_write\_32는 from\_host\_write\_32라는 포트의 block design에서 액세스할 수 있습니다.

- Loopbacks: 초기에 from\_host\_write\_32는 to\_host\_read\_32에 연결되고 from\_host\_write\_8은 to\_host\_read\_8에 연결됩니다. 이것은 xillybus\_write\_32로 명명된 device file에 기록된 모든 데이터를 xillybus\_read\_32로 루프백합니다. write\_8/read\_8 쌍도 마찬가지입니다. 이 loopback은 Getting Started 가이드에 설명된 “Hello world” 테스트를 작동하게 만드는 것입니다.

application logic와 통합하려면 해당 loopback 연결을 Vivado의 block design GUI와 함께 제거하고 application logic의 적절한 AXI Stream 포트에 연결해야 합니다.

- 어떤 경우에는 xillybus\_smb 및 xillybus\_audio라는 이름의 streams가 보드의 오디오 인터페이스를 지원하는 데 사용되기 때문에 위의 계층 구조에 연결됩니다. 이러한 streams는 무시되어야 합니다(즉, block design의 processor design 계층으로 가는 나머지 신호로 처리됨).
- 각 Xillybus stream의 “\*\_open” 포트: 각 AXI Stream 포트에는 \_open 접미사가 있는 해당 포트가 있으며, 해당 포트는 관련 Xillybus device file이 host에서 열려 있을 때 높음( '1' )입니다. 이 신호는 선택적으로 stream에 연결된 모든 application logic을 재설정하는 데 사용할 수 있으므로 device file이 열릴 때마다 알려진 상태가 됩니다.
- Clocking Wizard (stream\_clk\_gen) block: Xillybus의 인터페이스에서 가져온 clock을 기반으로 하는 application logic용 clock을 생성합니다. 모든 Xillybus IP core의 AXI Stream ports는 이 block의 출력과 동기화됩니다.  
output clock의 주파수를 제외하고 이 block에서 변경하지 않는 것이 좋습니다. 특히 design의 implementation(timing constraints)와 관련된 특정 scripts는 이름으로 이 block의 출력을 참조하므로 block의 이름은 그대로 유지되어야 합니다(stream\_clk\_gen ).  
아래 섹션 3.2을 참조하십시오.
- 외부 포트(예: GPIO\_LEDS[0:3]): block design 위의 계층 구조에 연결된 포트입니다. 이러한 연결은 변경되어서는 안 되지만 그 신호는 그 안의 블록에 의해 샘플링될 수 있습니다. 예를 들어, Xilinx 번들에서 ap\_clk은 상위 계층으로 이동하지만 block design 보기 내에서도 사용할 수 있습니다.

mem\_8 stream에는 포트가 없습니다. Seekable streams는 block diagram에 제공되지 않습니다. 이에 대한 자세한 내용은 4.3 섹션을 참조하십시오.

# 3

## application logic와 통합

### 3.1 기본

application logic와의 통합은 Vivado의 block design GUI를 사용하여 수행됩니다. IP blocks는 block design에 추가되고 필요에 따라 연결됩니다.

Vivado's High Level Synthesis (HLS)에 의해 생성된 IP blocks의 통합에 대해서는 6 섹션을 참조하십시오.

Xillybus의 문서에서 FPGA의 application logic이 host를 통해 FIFOs와 통신한다고 자주 언급되지만 Block Design Flow의 경우는 그렇지 않습니다 (Verilog / VHDL design flow에 한해). AXI Stream 인터페이스를 생성하는 Xillybus의 IP Core에 있는 glue logic에는 이미 FIFOs가 포함되어 있습니다(bus\_clk와 ap\_clk 사이의 clock domain crossing용). 결과적으로 VHDL / Verilog design flow와 달리 Block Design Flow를 사용할 때 Xillybus의 IP core와 인터페이스하기 위해 application logic에서 FIFOs를 배포할 필요가 없습니다.

FPGA와 host 간의 데이터 교환을 위해 application logic을 전용 AXI Stream 포트에 연결합니다(loopbacks 연결을 끊은 후 가능). 이 포트는 TDATA, TVALID 및 TREADY만 표시하며 특히 TLAST 신호는 표시 하지 않습니다. 결과적으로 각 AXI Stream stream은 infinite data stream을 구현합니다(TLAST 신호가 허용하는 패킷 인터페이스와 반대). 이는 일반적으로 Xillybus device files의 infinite stream 특성과 일치합니다.

Xillybus streams는 다음 두 가이드 중 하나의 섹션 6.3에 설명된 대로 FPGA와 host 간에 패킷을 교환하는 데 사용할 수 있습니다.

- [Xillybus host application programming guide for Linux](#)
- [Xillybus host application programming guide for Windows](#)

## 3.2 Clocking

### 3.2.1 일반적인

단순화를 위해 user application logic을 Xillybus IP Core에 연결하는 모든 신호는 block design의 Clocking Wizard block에 의해 생성되는 단일 clock에 의해 구동되어야 합니다. 이 clock("application clock")는 Clocking Wizard의 clk\_out1이며 Xillybus IP Core block의 ap\_clk 입력이기도 합니다.

이 단일 clock로 전체 user application block을 구동하는 것이 편리한 경우가 많으므로 내부 logic과 인터페이스는 모두 user application block에 의존합니다. 예를 들어, Vivado HLS' synthesizer에 의해 생성된 logic에는 단일 clock input(ap\_clk로 명명)가 있습니다. 이 clock input을 Clocking Wizard의 출력에 연결하면 Xillybus IP Core의 block와 AXI Stream 포트 연결이 제대로 작동합니다.

FPGA 도구는 때때로 clock의 주파수를 주파수 자체(일반적으로 MHz에서)로 참조하고 때로는 clock period(일반적으로 ns에서)를 참조합니다. clock의 주파수는 clock period의 reciprocal이므로 예를 들어 100 MHz는 10 ns의 clock period와 같습니다.

### 3.2.2 application clock 설정

application clock의 주파수는 성능을 향상시키거나 작동하는 bitfile을 달성하기 위한 단계로 설정할 수 있습니다. 더 빠른 clock은 더 높은 처리 처리량을 산출하지만(어떤 다른 병목 현상이 성능을 제한하지 않는 한) FPGA의 logic 요소와 AMD의 활용도에서 더 많은 것을 요구합니다. 도구.

application clock의 주파수가 너무 높게 선택되면 프로젝트의 compilation이 timing constraints를 충족하지 못하여 FPGA bitstream 파일로 실패합니다. 이것은 "timing failure"라고도 합니다. 이 상황은 implementation을 실행하는 도구가 안정적인 작동을 보장하는 방식으로 logic을 활용하지 못한 반면 logic은 정의된 clock의 주파수에 의해 구동된다는 것을 의미합니다. 이 맥락에서 "timing constraints"는 시스템의 clocks 주파수에 대한 요구 사항입니다.

application clock의 주파수를 줄이는 것은 항상 허용되지만(clock generator의 한계 내에서), application clock이 구동하는 logic의 작동 속도가 느려집니다.

application clock의 주파수를 설정하려면 block design 보기에서 Clock Wizard( stream\_clk\_gen)의 block을 두 번 클릭합니다. Vivado에서 구성 창이 열립니다. "Output Clocks" 탭을 선택하고 clk\_out1에 대한 "Output Clock Requested" 주파수를 변경합니다. "Actual" 열의 주파수는 clock synthesizer에 의해 생성될 주파수를 보여줍니다. output clock은 허용된 값의 제한된 집합에서 선택되는 유리수를 input clock에 곱하여 파생되므로 요청된 주파수와 약간 다를 수 있습니다.

clock이 application logic 및 Xillybus IP core와의 인터페이스에만 사용되는 경우 요청된 주파수에서 약간의 전환은 무해합니다.

Clocking Wizard의 다른 매개변수는 변경하면 안 됩니다.

### 3.2.3 bus\_clk 신호

Xillybus IP Core의 내부 logic은 bus\_clk에 의해 구동되며, bus\_clk에서 application clock의 파생을 허용하기 위해 block design에 노출됩니다. application logic은 내부 logic 및 Xillybus IP Core와의 인터페이스를 위해 ap\_clk만 필요하기 때문에 일반적으로 이 신호에 대한 다른 용도는 없습니다.

그러나 bus\_clk의 주파수는 처리량 병목 현상을 발견하기 위해 관심을 가질 수 있습니다. 예를 들어 bus\_clk이 100 MHz에서 실행되는 경우 Xillybus의 내부 data pipe이 bus\_clk의 속도로 실행되기 때문에 32비트 폭 데이터 인터페이스를 통과할 수 있는 최대 이론적 대역 폭은 400 MB/s입니다. ap\_clk이 더 높은 주파수에서 실행되고 ap\_clk의 각 주기에서 데이터가 푸시되면 AXI Stream 흐름 제어 신호(TREADY 및 TVALID)로 인해 데이터 속도가 느려질 가능성이 있습니다.

이러한 이유로 bus\_clk의 주파수는 애플리케이션의 처리량을 최대화하려고 시도할 때, 특히 데이터 인터페이스에 긴 버스트(또는 연속) 데이터 트래픽이 포함될 것으로 예상되는 경우 고려해야 합니다.

bus\_clk의 주파수는 “Clocking Options” 탭에서 찾을 수 있으며, primary input clock의 주파수는 clk\_in1입니다. 이 매개변수는 입력에서 예상되는 주파수를 Clocking Wizard에 알리므로 특정 Xillybus 번들에 대한 bus\_clk의 주파수를 아는 데 사용할 수 있습니다.

# 4

## 가속화/코프로세싱 모범 사례

### 4.1 처리량 대 latency

향상된 명령어 세트(예: x86 제품군의 MMX 명령, AES용 암호화 확장, ARM의 NEON 확장)를 기반으로 하는 기존 하드웨어 가속과 GPGPU 및 FPGA와 같은 외부 하드웨어를 사용한 가속 간에는 상당한 차이가 있습니다. 향상된 instruction sets는 processor의 실행 흐름의 일부이기 때문에 긴 기계 코드 명령 시퀀스를 더 짧은 시퀀스로 대체하고 결과를 사용할 수 있을 때까지 필요한 사이클 수를 줄입니다.

반면에 외부 하드웨어 가속(FPGA 가속 포함)은 데이터를 외부 하드웨어로/로부터 전송하는 상당한 latency로 인해 결과를 사용할 수 있을 때까지 반드시 시간을 줄이는 것은 *아닙니다*. 또한 처리 시간은 pipelining 및 clock의 더 낮은 주파수로 인해 processor보다 훨씬 더 길 수 있습니다.

따라서 외부 하드웨어 가속의 장점은 latency(결과를 얻는 속도)가 아니라 처리량(데이터가 처리되는 속도)입니다. 이 이점을 활용하려면 다음 작업을 시작하기 전에 한 작업의 결과를 기다리는 것보다 가속 하드웨어와 주고받는 *데이터 흐름* 을 유지하는 것이 중요합니다.

FPGA로 적절한 가속을 위한 기술은 다음 두 문서 중 하나의 섹션 6.6에 자세히 설명되어 있습니다.

- [Xillybus host application programming guide for Linux](#)
- [Xillybus host application programming guide for Windows](#)

### 4.2 데이터 폭 및 성능

비교적 높은 데이터 대역폭이 필요한 애플리케이션의 경우 데이터 집약적 streams에 대해 32비트 너비 streams(또는 그 이상)를 사용하는 것이 좋습니다. 8비트 및 16비트 너비의 streams가 host의 bus를 덜 효율적으로 활용하기 때문입니다.

그 이유는 단어가 bus의 속도로 Xillybus 내부 데이터 경로를 통해 전송되기 때문입니다. 결과적으로 8비트 워드를 전송하는 데 32비트 워드와 동일한 시간 슬롯이 걸리므로 효과적으로 4배 느려집니다.

이는 데이터 경로가 더 느린 데이터 요소로 채워지기 때문에 주어진 시간에 기본 전송을 위해 경쟁하는 다른 streams에도 영향을 미칩니다.

이 지침은 좁은 streams를 동일한 효율로 전송하는 개정판 B/XL/XXL Xillybus IP cores에 적용되지 않습니다.

### 4.3 해야 할 일과 하지 말아야 할 일

Block Design Flow로 작업할 때 주의해야 할 몇 가지 문제가 있습니다.

- 주소 포트가 있는 Streams(“address/data streams”, “seekable streams”)는 Block Design Flow에서 지원되지 않습니다. Xillybus IP Core에 이러한 streams가 포함되어 있으면 GUI에서는 포트가 나타나지 않지만 host 쪽에서는 정상적으로 나타납니다. host의 이러한 stream에서 읽기를 시도하면 즉각적인 end-of-file 조건이 생성됩니다. 다른 쪽 끝에 data sink이 없기 때문에 write() 함수 호출은 반환되지 않습니다. 따라서 FPGA logic의 혼동과 약간의 낭비를 피하기 위해 Block Design Flow와 함께 사용하도록 의도된 사용자 지정 IP cores에서 seekable streams를 사용하지 않는 것이 좋습니다.
- 3.2.2 섹션에 설명된 대로 필요한 경우 출력 주파수를 변경하는 것을 제외하고 “stream\_clk\_gen”(Clocking Wizard)라는 block을 변경하지 마십시오. input clock의 주파수를 변경하거나 구성을 변경하거나 design에서 제거하고 새로운 Clocking Wizard IP block로 교체하면 timing constraints를 충족하지 못할 수 있습니다(일부 timing constraints 예외가 block의 이름을 참조하기 때문일 수 있음). 잘못된 입력 주파수를 설정하면 FPGA design의 불안정한 동작이 발생할 수 있습니다.
- clocks가 연결되는 방식에 주의를 기울이는 것이 중요합니다. 특히 bus\_clk와 ap\_clk을 섞지 마십시오.
- Xillybus streams가 비동기식인지 확인하십시오. 기본 IP core의 경우와 stream의 의도된 용도가 “Data exchange with coprocessor”일 때 사용자 지정 IP cores의 autoselect 선택의 경우입니다. 이로 인해 host에서 수행된 write() 함수 호출이 DMA buffers에 데이터를 위한 충분한 공간이 있는 경우 즉시 반환되어 보다 원활한 데이터 전송과 더 높은 대역폭 성능을 보장합니다.

이 주제에 대한 더 나은 이해를 위해 [Xillybus host application programming guide for Linux](#) 또는 [Xillybus host application programming guide for Windows](#)의 섹션 2를 참조하십시오.

## 5

## 커스텀 Xillybus IP core 적용하기

웹 애플리케이션을 통해 사용자는 사용자 정의 Xillybus IP cores를 구성 및 다운로드하여 Xillybus의 웹사이트에서 직접 streams의 수와 속성을 선택할 수 있습니다. 특수 생성된 사용자 지정 IP core는 일반적으로 몇 분 후에 사이트에서 다운로드됩니다.

맞춤형 IP core를 생성하고 다운로드하려면 Xillybus 웹사이트에서 [IP Core Factory](#)를 방문하십시오. 프로세스는 매우 간단하며 필요한 경우 [The guide to defining a custom Xillybus IP core](#)에서 무료 정보를 제공합니다.

### 중요한:

*Seekable streams* ("address/data" 인터페이스 포함)는 AXI Stream 연결이 주소 와 이어를 지원할 수 없기 때문에 Block Design Flow에서 보이지 않습니다. core에 이러한 streams를 사용하는 것은 상당히 무해하지만 FPGA logic 리소스에 약간의 낭비를 일으키고 host 측에 나타나지만 block design에는 나타나지 않는 혼동을 일으킬 수 있습니다.

사용자 지정 IP core가 정의되면 번들을 생성하고 다운로드합니다.

사용자 지정 IP core 번들의 README 파일에 있는 지침은 Verilog / VHDL design flow와 관련이 있으므로 무시해야 합니다. 대신 다음 단계를 수행해야 합니다.

- 사용자 정의 IP core의 파일을 위한 새 디렉토리를 작성하십시오. 이 디렉토리의 absolute path는 이 사용자 정의 IP core를 사용하는 동안 고정된 상태로 유지되어야 하므로 실수로 삭제되지 않는 위치에 두는 것이 좋습니다.  
다운로드한 사용자 정의 IP core 번들의 압축을 이 디렉토리에 풉니다.
- Vivado에서 block design을 엽니다.
- 참조를 위해 block design의 보기를 pdf 파일로 저장합니다. block design 영역의 아무 곳이나 마우스 오른쪽 버튼으로 클릭하고 "Save as pdf file..."을 선택합니다.

- Tools 메뉴(메인 메뉴 표시줄)에서 “Run Tcl Script...”를 선택합니다. 사용자 지정 IP core 번들의 압축을 푼 디렉터리로 이동하고 xillybus\_block 하위 디렉터리로 이동합니다. insertcore.tcl을 선택합니다.
- script는 기존 Xillybus IP Core를 커스텀 IP core로 교체하고, 애플리케이션과 관련이 없는 배선도 재연결을 시도합니다. 개체는 자동 재구성으로 인해 block design 다이어그램에서도 이동할 수 있습니다.
- application logic의 AXI Stream 인터페이스를 업데이트된 Xillybus IP core에 연결합니다.
- script를 실행하기 전에 생성된 pdf 파일과 비교하고 필요에 따라 수정합니다.  
**application logic 관련 연결은 다시 연결되지 않으며 다른 연결도 누락될 수 있습니다.**
- Xillybus IP Core의 block 아래 및 아래 캡션이 새 IP core의 이름과 일치하는지 확인하십시오.

script는 이름으로 blocks, 포트 및 인터페이스를 찾습니다. 따라서 이러한 이름이 사용자에게 의해 변경된 경우 연결 복원에서 부분적으로(그리고 자동으로) 실패할 수 있습니다.

이 시점부터 프로젝트의 implementation은 이전과 같이 수행될 수 있습니다. Xillybus의 host용 driver(Linux 및 Windows 유사)는 새로운 IP core의 구성을 자동으로 감지하기 때문에 맞춤형 IP core에서도 작동합니다.

따라서 사용자 지정 IP core로 교체한 후 host에 아무 것도 설치할 필요가 없습니다.

참고로 insertcore.tcl script의 실행 단계는 다음과 같습니다.

- 사용자 지정 IP core의 디렉터를 Vivado의 IP Catalog에 있는 IP Core repositories 목록에 추가하고 repositories를 강제로 다시 검색하여 새 사용자 지정 IP Core를 검색하여 Catalog에 추가합니다.
- 있는 경우 block design에서 이전 Xillybus IP를 제거합니다.
- 사용자 지정 IP core를 design에 추가하고 필요한 경우 해당 버전을 업그레이드하십시오.
- 이름 목록을 찾고 이러한 이름을 가진 모든 포트를 상호 연결하여 위의 계층 구조로 가는 와이어와 stream\_clk\_gen의 block에 대한 와이어를 다시 연결합니다(있는 경우).
- Zynq에만 해당: Xillybus IP core의 bus 주소를 기본값으로 설정합니다(0x50000000에서 시작하는 4 kB 세그먼트).

- 프로젝트의 synthesis 실행을 재설정하여 다음 implementation이 변경 사항을 반영하도록 합니다.

# 6

## Vivado HLS 통합

### 6.1 개요

이 섹션에서는 간단한 C function에서 IP block로의 compilation와 Xillybus의 Block Design flow에 통합되는 방법을 보여줍니다.

이 섹션의 기반이 되는 예제 프로젝트는 다음에서 다운로드할 수 있습니다.

<https://xillybus.com/downloads/hls-axis-starter-1.0.zip>

다운받은 파일은 Xillybus 프로젝트와 쉽게 관련이 있는 디렉토리에 압축을 푸는 것이 좋습니다. 이후 단계에서 이동할 수 없기 때문입니다.

예제 프로젝트에서 두 가지 다른 종류의 C 소스를 구별하는 것이 중요합니다.

- 실행을 위한 코드: 다른 컴퓨터 프로그램과 마찬가지로 컴퓨터 또는 embedded 플랫폼(“host”)에서 실행되며 FPGA를 사용하여 특정 작업을 오프로드합니다.  
예제 프로젝트에서 샘플 파일은 host/ 하위 디렉토리에서 찾을 수 있습니다.
- synthesis용 코드: Vivado HLS에서 logic로 변환하기 위한 것입니다.  
예제 프로젝트에서 coprocess/example/src/main.c에서 찾을 수 있습니다.

일반적인 C/C++ 프로그래밍과 달리 host 프로그램은 synthesized function을 호출하지 않습니다. 오히려 기능을 실행하는 데 필요한 데이터를 데이터 구조로 구성하고 간단한 API를 사용하여 synthesized function로 전송합니다. 이에 대해서는 나중에 설명합니다. 나중 단계에서 유사한 API와 함께 synthesized function에서 보낸 데이터 구조로 반환 데이터를 수집합니다.

## 6.2 HLS synthesis

이 섹션에서 사용된 C의 예제 코드는 섹션 6.4에 요약되어 있습니다.

Vivado HLS를 시작하고 HLS 프로젝트를 엽니다. 시작 페이지에서 “Open Project”를 선택하고 HLS 프로젝트 번들이 압축 해제된 위치로 이동한 다음 이름이 “coprocess”인 폴더를 선택합니다.

프로젝트의 부품 번호 변경: Solution >Solution Settings... >Synthesis를 선택하고 “Part Selection”을 의도한 FPGA로 변경합니다.

Solution >Synthesis >Active Solution을 선택하여 프로젝트(“synthesize”)의 compilation을 시작합니다(또는 도구 모음에서 해당 아이콘 클릭). 여러 warnings(경상)를 포함하여 많은 텍스트가 console에 나타납니다. 오류가 발생하지 않아야 합니다.

성공적인 compilation은 HLS의 console 탭에 있는 마지막 몇 줄 중 다음 메시지로 쉽게 인식됩니다.

```
Finished C synthesis.
```

synthesis 보고서는 synthesis가 성공한 경우에만 console 탭 위에도 나타납니다.

Vivado HLS에 대한 자세한 내용은 사용 설명서(UG902)를 참조하십시오.

## 6.3 FPGA 프로젝트와 통합

Vivado HLS에서 Solution >Export RTL을 선택하고 “IP Catalog”을 Format Selection로 선택합니다. “Evaluate Generated RTL”의 경우 Verilog을 선택하고 이 아래의 확인란을 선택하지 마십시오. OK를 클릭합니다.

몇 분이 걸릴 수 있으며 다음과 같이 끝납니다.

```
Finished export RTL.
```

이제 Vivado(즉, Vivado HLS가 아님)에서 Xillydemo 프로젝트(섹션 2.1에 설정된 대로)를 열고 Block Design을 엽니다. Xilinx( Zynq )를 사용하는 경우 “blockdesign”라는 block을 엽니다.

다음과 같이 HLS IP block을 추가합니다. block design 다이어그램 영역의 아무 곳이나 마우스 오른쪽 버튼으로 클릭하고 “IP Settings...”를 선택합니다. “Repository Manager” 탭에서 repository를 추가하기 위해 녹색 더하기 기호를 클릭합니다. 6.2 섹션에서 선택한 동일한 “coprocess” 디렉토리로 이동하여 선택하여 HLS 프로젝트를 엽니다. Vivado는 하나의 repository가 추가되었음을 나타내는 팝업 창으로 응답해야 합니다. “OK” 버튼을 두 번 클릭하여 확인합니다.

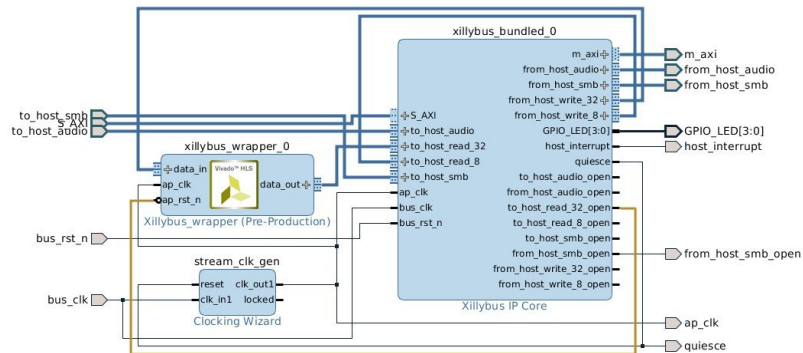
이제 IP block을 block design에 추가합니다. 다시 한 번 block design 다이어그램 영역의 아무 곳이나 마우스 오른쪽 버튼으로 클릭합니다. “Add IP..”를 선택하고 목록에서 Xillybus\_wrapper IP를 선택합니다(검색 상자에 “wrapper”를 입력하면 더 쉽게 할 수 있음).

xillybus\_wrapper\_0라는 새로운 block이 다이어그램에 나타납니다. to\_host\_read\_32와 from\_host\_write\_32 사이를 연결하는 전선을 분리합니다(즉, loopback 분리).

그런 다음 다음과 같이 xillybus\_wrapper block을 연결합니다.

- from\_host\_write\_32와 함께 data\_in
- to\_host\_read\_32와 함께 data\_out
- to\_host\_read\_32\_open와 함께 ap\_rst\_n
- ap\_clk이 있는 ap\_clk(Clocking Wizard의 clk\_out1 출력이기도 함)

결과는 다음과 같아야 합니다(Xilinx 기반 block design에 대해 표시됨).



ap\_rst\_n와 to\_host\_read\_32\_open 사이의 연결은 xillybus\_read\_32 device file이 host에서 열리지 않는 한 logic을 xillybus\_wrapper의 block 내부에서 리셋 상태로 유지합니다(파일이 열리지 않을 때 to\_host\_read\_32\_open은 로우이고 리셋 입력은 활성 로우임). host에서 실행 중인 소프트웨어가 이 block와 통신을 시도하기 전에 이 device file을 연다고 가정하면 소프트웨어가 실행될 때마다 logic의 일관된 응답이 보장됩니다.

이 시점에서 implementation을 수행하여 bitstream을 얻을 수 있습니다. Vivado 창 하단에서 “Design Runs” 탭을 선택하고 “synth\_1”을 마우스 오른쪽 버튼으로 클릭한 다음 “Reset Runs”를 선택합니다. synth\_1 재설정을 확인합니다.

그런 다음 왼쪽 막대에서 “Generate Bitstream”을 클릭합니다.

## 6.4 예제 synthesis 코드

HLS가 Xillybus와 함께 작동하는 방식을 명확히 하기 위해 예제에서는 간단한 사용자 지정 함수인 mycalc()에서 다루는 삼각 사인 계산과 integer를 사용한 간단한 작업을 보여줍니다.

coprocess/example/src/main.c는 다음과 같이 시작합니다.

```
#include <math.h>
#include <stdint.h>

extern float sinf(float);

int mycalc(int a, float *x2) {
    *x2 = sinf(*x2);
    return a + 1;
}
```

평소와 같이 두 개의 #include statements가 있습니다. 사인 함수에는 “math.h” 포함이 필요합니다.

그리고 “synthesized function”의 역할을 하는 간단한 기능인 mycalc()이 있습니다. floating point는 물론 integer에서도 산술 연산을 보여주는 아주 간단한 함수입니다. High-Level Synthesis Guide UG902는 보다 유용한 작업을 구현하는 방법에 대한 자세한 정보를 제공합니다.

다음 main.c에는 synthesized function과 Xillybus 사이의 다리 역할을 하는 래퍼 기능인 xillybus\_wrapper()가 있으며, 따라서 앞뒤로 데이터를 패킹하고 언패킹합니다.

예제의 경우 host에서 data stream까지 “data\_in” 인수로 표시되는 integer 및 floating point 형식의 숫자를 허용합니다. “data\_out” 인수를 사용하여 integer에 1을 더한 값과 floating point 숫자의 (삼각) 사인을 반환합니다.

```

void xillybus_wrapper(int *data_in, int *data_out) {
#pragma AP interface axis port=data_in
#pragma AP interface axis port=data_out
#pragma AP interface ap_ctrl_none port=return

uint32_t x1, tmp, y1;
float x2, y2;

// Handle input data
x1 = *data_in++;
tmp = *data_in++;
x2 = *((float *) &tmp); // Convert uint32_t to float

// Run the calculations
y1 = mycalc(x1, &x2);
y2 = x2; // This helps HLS in the conversion below

// Handle output data
tmp = *((uint32_t *) &y2); // Convert float to uint32_t
*data_out++ = y1;
*data_out++ = tmp;
}

```

xillybus\_wrapper()는 두 개의 pointers로 선언되며 둘 다 int 유형의 변수입니다. 이러한 함수 인수는 block design에 포함하기 위해 예정 IP block의 두 AXI Stream 포트에 바뀝니다. 각각은 HLS에 “axis” 유형의 인터페이스로 간주되어야 함을 알리는 #pragma 문이 있습니다.

“#pragma AP”와 “#pragma HLS”는 서로 바꿔 사용할 수 있습니다. 전자는 C Synthesizer의 이전 이름( Auto Pilot )을 기반으로 하고 후자는 AMD의 최근 문서에서 볼 수 있습니다.

“int”는 HLS에서 32비트 워드로 간주되므로 각 AXI Stream 인터페이스는 32비트 폭의 데이터 인터페이스를 갖습니다.

물론 인수 목록과 pragmas를 변경하여 AXI Stream 입력 및 출력 세트를 얻을 수 있습니다.

ap\_ctrl\_none에 대한 pragma 선언은 (존재하지 않는) return value에 대한 포트를 생성하지 않도록 compiler에 지시합니다.

다음으로 “execution”에 대한 몇 가지 코드가 있습니다. 입력 데이터를 가져옵니다. 각 \*data\_in++ 작업은 host에서 시작되는 32비트 워드를 가져옵니다. 표시된 코드에서 첫 번

째 단어는 unsigned integer로 해석되고 x1에 배치됩니다. 두 번째 워드는 32비트 float로 처리되어 x2에 저장됩니다.

그런 다음 mycalc(), “synthesized function”에 대한 함수 호출이 있습니다. 이 함수는 하나의 결과를 반환 값으로 반환하고 두 번째 데이터 조각은 x2를 변경하여 되돌아갑니다.

래퍼 함수는 x2의 업데이트된 값을 새 변수 y2로 복사합니다. 이 코드의 compilation이 processor에서 실행되도록 의도된 경우 중복 작업으로 나타날 수 있습니다. 그러나 HLS를 사용할 때 이것은 compiler가 나중에 float로의 변환을 처리하도록 하기 위해 필요합니다. 이것은 HLS compiler의 다소 기이한 동작을 반영하지만 이것은 pointer 사용의 섬세한 문제 중 하나입니다. 메모리 어레이와 이에 대한 pointer가 C 코드에 정의되어 있어도 HLS compiler는 그것들을 생성하지 않습니다. pointer의 사용은 우리가 달성하고자 하는 것에 대한 힌트일 뿐이며 때로는 이러한 힌트를 약간 밀어야 합니다.

마지막으로 결과는 host로 다시 전송됩니다. 각 \*data\_out++는 float에서 적절한 변환과 함께 32비트 워드를 컴퓨터로 보냅니다.

\*data\_in++ 및 \*data\_out++ 연산자는 실제로 pointers를 이동하지 않으며 기본 메모리 어레이가 없습니다. 오히려, 이는 AXI stream 인터페이스에서(그리고 결국 Xillybus streams에서 또는 Xillybus streams로) 데이터를 이동하는 것을 상징합니다. 따라서 “data\_in” 및 “data\_out” 변수가 사용되는 유일한 방법은 \*data\_in++ 및 \*data\_out++입니다(High-Level Synthesis Guide는 다른 가능성, 특히 고정 크기 어레이를 제공함).

또한 이 코드는 logic로 변환되고 processor에 의해 실행되지 않기 때문에 이러한 C 명령의 유일한 의미는 데이터의 입력 stream이 주어지면 예상되는 데이터 stream 출력을 생성하는 것입니다. 그러나 데이터가 언제 방출되는지에 대한 약속은 없습니다(HLS 보고서에 제공된 가능한 latencies 범위 제외).

따라서 입력 데이터의 할당 순서는 들어오는 데이터가 해석되는 방식을 강제한다는 점에서 중요합니다. 반면에 첫 번째로 보내는 출력 y1은 첫 번째 입력이 도착하는 x1에만 의존하기 때문에 두 번째 입력이 도착하기 전에 첫 번째 출력을 보낼 수 있습니다. 이는 코드 실행의 직관적인 순차적 특성과 모순되지만 전체 결과가 동일하기 때문에 하드웨어 가속의 맥락에서 의미가 없습니다.

또한 data\_in AXI stream에 지속적으로 데이터가 공급되면 래퍼 기능 “runs”가 다음과 같이 반복적으로 수행됩니다.

```
while (1) // This while-loop isn't written anywhere!
    xillybus_wrapper(data_in, data_out);
```

가능한 한 빨리 \*data\_in++ 명령을 통해 새 데이터를 가져와 logic의 내부 pipeline(HLS 보고서에 따르면 예제 프로젝트에서 70단계 이상)를 채울 가능성이 높습니다. 따라서 한 쌍의 단어를 가져와 처리하고 두 개의 출력 단어를 내보낸 다음 두 번째 단어 쌍을 가져온 processor의 코드 실행과 달리 HLS 해석은 data\_in에서 70단어를 가져올 수 있습니다.

data\_out AXI stream에서.

## 6.5 synthesis용 C/C++ 코드 수정

추가 AXI Stream 포트는 예제와 같이 래퍼 함수에 인수를 추가하고 이를 인터페이스 포트로 선언하여 생성할 수 있습니다.

물론 예제 design의 C 코드에서 다른 변경을 하는 것도 가능합니다.

\*data\_in++ 및 \*data\_out++와 동일한 스타일로 I/O를 구현하거나 다른 가능성에 대해서는 High-Level Synthesis Guide (UG902)를 참조하는 것이 좋습니다. 코딩 기술을 배우기 위한 권장 소스이기도 합니다.

### 중요한:

변경한 후 Vivado에서 "Generate Bitstream"을 클릭하지 마십시오. 아래에 자세히 설명된 대로 block을 업그레이드하지 않고 bitstream의 반복적인 implementation을 실행하면 bitfile의 implementation이 성공적으로 보일 수 있지만 HLS block의 오래된 버전을 기반으로 합니다.

샘플 프로젝트에서 변경 사항이 적용된 후 섹션 6.2의 "HLS synthesis"에서 시작하여 Vivado가 있는 implementation로 이동하고 Vivado에서 HLS block을 업데이트합니다.

다시 말해:

- Vivado HLS: HLS에서 프로젝트의 compilation을 실행합니다. HLS synthesizer는 새 파일을 시작하기 전에 항상 이전 compilations에서 생성된 파일을 정리합니다.
- Vivado HLS: IP Catalog bundle로 내보냅니다.
- Vivado( Vivado HLS 아님 )에서 xillybus\_wrapper 블록을 업그레이드합니다(실제로는 변경 후 업데이트). block design view를 열고 페이지 상단의 메시지에 응답하여 block을 업그레이드해야 한다는 메시지에 응답합니다. 이 메시지를 찾을 수 없으면 Tcl Console에 "report\_ip\_status -name status"를 입력합니다. 하단의 "Upgrade Selected" 버튼을 클릭합니다. 그러면 성공적인 업그레이드를 확인하는 대화 상자와 출력 제품 생성을 요청하는 대화 상자가 나타납니다. 두 번째 대화 상자에서 "Skip"을 클릭합니다.
- Vivado: design runs가 무효화되었는지 확인하십시오. Vivado 창 하단에서 "Design Runs" 탭을 선택하십시오. synth\_1의 경우 Status 열에 Synthesis Out-of-date라고 표시되어야 합니다.

- Vivado: **design runs가 무효화되지 않은 경우 다음을** 시도하십시오. IP 카탈로그 새로 고침: block design 다이어그램 영역의 아무 곳이나 마우스 오른쪽 버튼으로 클릭하고 “IP Settings...”를 선택하십시오. “Repository Manager” 탭에서 하단의 “Refresh All” 버튼을 클릭합니다. 동일한 대화 상자의 “General” 탭에서 “Clear Cache”를 클릭해야 할 수도 있습니다. 그런 다음 xillybus\_wrapper의 block 업그레이드로 돌아가십시오.

위의 이전 항목에서 *design runs*가 무효화된 것으로 확인된 경우 이러한 작업이 필요하지 않습니다.

- Vivado: synth\_1 실행 재설정
- Vivado: bitstream 생성

## 6.6 simple.c: host 프로그램의 예

예제 프로젝트에는 simple.c 및 practical.c라는 두 개의 C 파일로 샘플 host 프로그램이 있습니다. 이것은 프로젝트의 host 측면을 보여줍니다.

둘 다 Linux host용으로 작성되었으며 예를 들어 compilation용으로 작성되었습니다.

```
# gcc -O3 -Wall simple.c -o simple
```

그러나 Windows에 쉽게 적용됩니다(아래 참조).

### 중요한:

simple.c는 특히 다음과 같은 단점 때문에 실제 host 프로그래밍의 예로 사용해서는 안 됩니다.

- 하나의 단일 요소만 처리됩니다. write() 및 read() 함수 호출 쌍을 반복하면 성능이 저하됩니다.
- write() 및 read() 작업의 반환 값이 제대로 작동하는지 확인해야 합니다. 이것은 단순화를 위해 생략되었지만 프로그램을 신뢰할 수 없게 만듭니다.

섹션 6.7은 더 나은 코딩 기술을 설명합니다.

simple.c 파일은 #include statements로 시작합니다.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
```

그 다음에는 main() 함수의 클래식 선언과 일부 변수 선언이 이어집니다.

```
int main(int argc, char *argv[]) {
    int fdr, fdw;

    struct {
        uint32_t v1;
        float v2;
    } tologic, fromlogic;
```

struct 변수는 아래에서 설명합니다.

프로그램은 named pipes처럼 동작하고 logic와 통신하는 데 사용되는 두 개의 device files를 여는 것으로 시작합니다: /dev/xillybus\_read\_32 및 /dev/xillybus\_write\_32. Xillybus 번들 설정에서 이 두 파일이 Xillybus의 driver에 의해 생성되었음을 상기하십시오.

섹션 6.3에서 지적했듯이 ap\_rst\_n은 block design 다이어그램에서 to\_host\_read\_32\_open에 연결되어 있으므로 /dev/xillybus\_read\_32를 열면 logic이 재설정되지 않습니다. 이것이 데이터 전송 전에 두 파일이 모두 열리는 이유입니다.

```
fdr = open("/dev/xillybus_read_32", O_RDONLY);
fdw = open("/dev/xillybus_write_32", O_WRONLY);

if ((fdr < 0) || (fdw < 0)) {
    perror("Failed to open Xillybus device file(s)");
    exit(1);
}
```

다음으로 실제 실행입니다. "tologic" 구조는 logic로 전송하기 위한 몇 가지 값으로 채워지며, 그 후 메모리에서 xillybus\_write\_32로 직접 기록됩니다. 효과적으로 이것은 8바이트, 더 정확하게는 2개의 32비트 워드를 씁니다. 첫 번째는 tologic.v1에 입력된 정수 123이고 두 번째는 tologic.v2에 있는 float입니다. 따라서 tologic 구조는 데이터의 logic 예상과 일치하도록 설정되었습니다. 첫 번째 \*data\_in++ 명령에 의한 하나의 integer, 두 번째 명령

에 의한 하나의 float.

```
tologic.v1 = 123;
tologic.v2 = 0.78539816; // ~ pi/4

// Not checking return values of write() and read(). This must
// be done in a real-life program to ensure reliability.

write(fdw, (void *) &tologic, sizeof(tologic));
read(fdr, (void *) &fromlogic, sizeof(fromlogic));

printf("FPGA said: %d + 1 = %d and also "
       "sin(%f) = %f\n",
       tologic.v1, fromlogic.v1,
       tologic.v2, fromlogic.v2);
```

6.4 섹션에서 래퍼 코드가 `data_in stream`에서 2개의 32비트 단어를 가져오는 것을 상기 하십시오. 첫 번째 단어는 “x1”로 이동하고 두 번째 단어는 “tmp”로 이동한 다음 “tmp”은 즉시 float로 변환됩니다. 이것은 “tologic” 구조의 2개의 32비트 요소와 일치합니다.

그런 다음 FPGA에서 데이터를 다시 읽습니다. 동일한 원칙이 “fromlogic”에도 적용됩니다.

`simple.c`는 일반적인 마무리로 끝납니다.

```
close(fdr);
close(fdw);

return 0;
}
```

`/dev/xillybus_write_32`로 전송된 데이터의 양을 래퍼 함수의 `*data_in++` 작업 수와 일치 시키는 것이 중요합니다. 전송된 데이터가 너무 적으면 synthesized function이 전혀 실행 되지 않을 수 있습니다. 너무 많으면 다음 실행이 잘못될 수 있습니다.

이 예에서는 “tologic” 및 “fromlogic”에 대해 동일한 구조 형식이 선택되었지만 이를 고수할 필요는 없습니다. 전송 및 수신된 데이터가 래퍼 기능의 `*data_in++` 및 `*data_out++` 작업 수와 동기화되는 것이 중요합니다.

이 프로그램의 실행은

```
# ./simple
FPGA said: 123 + 1 = 124 and also sin(0.785398) = 0.707107
```

마지막으로 다음 조정 중 일부 또는 전부를 수행해야 할 수 있는 Windows 사용자에게 대한

참고 사항입니다.

- 파일 이름 문자열을 “/dev/xillybus\_read\_32”에서 “\\\\.\\xillybus\_read\_32”로 변경합니다(Windows의 실제 파일 이름은 \\.xillybus\_read\_32이지만 escaping이 필요함). 두 번째 파일 이름이 “\\\\.\\xillybus\_write\_32”로 변경됩니다.
- unistd.h에 대한 #include 문을 io.h로 교체
- open(), read(), write() 및 close()에 대한 함수 호출을 \_open(), \_read(), \_write() 및 \_close()로 교체합니다.

## 6.7 practical.c: 실용적인 host 프로그램

simple.c 예제는 간결한 방식으로 데이터 교환을 설명하지만 실제 시스템에서는 몇 가지 변경이 필요합니다.

다음과 같은 차이점이 가장 두드러집니다.

- 처리를 위해 단일 데이터 세트를 생성하는 대신 구조 배열이 할당되어 전송됩니다. 마찬가지로 데이터 배열이 logic에서 수신됩니다. 이것은 I/O overhead 뿐만 아니라 소프트웨어와 하드웨어에 의해 야기되는 latencies의 영향을 감소시킵니다. 이것은 하드웨어 가속으로 성능을 향상시키는 중요한 방법입니다.
- 프로그램은 쓰기를 위한 프로세스와 데이터 읽기를 위한 프로세스의 두 가지 프로세스로 분기됩니다. 이 두 작업을 독립적으로 만들면 양쪽에서 처리할 데이터가 부족하여 처리가 지연되는 것을 방지할 수 있습니다. 이 독립성은 threads(특히 Windows에서) 또는 select() 함수 호출을 사용하여 달성할 수 있습니다.
- read() 및 write() 함수 호출은 올바르게 수행되어 안정적인 I/O를 보장합니다. 이 목적을 위해 추가된 while 루프는 성가신 것처럼 보일 수 있지만 로드 상태에서 자주 발생하는 이러한 함수 호출(모든 바이트 읽기 또는 쓰기가 아님)의 부분 완료에 올바르게 응답하는 데 필요합니다. EINTR 오류는 POSIX signals에 적절하게 대응하기 위해 필요에 따라 처리됩니다. POSIX signals는 실수로 실행 중인 프로세스로 전송될 수 있습니다.

이제 practical.c에 대해 간략히 살펴보겠습니다. 먼저 headers:

```
#include <stdio.h>
#include <unistd.h>

#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
```

그리고 동일한 구조에 N, 데이터 청크당 요소 수를 정의합니다.

```
#define N 1000

struct packet {
    uint32_t v1;
    float v2;
};
```

일반적인 main() 함수 정의 및 일부 변수:

```
int main(int argc, char *argv[]) {

    int fdr, fdw, rc, donebytes;
    char *buf;
    pid_t pid;
    struct packet *tologic, *fromlogic;
    int i;
    float a, da;
```

이전과 같이 열린 파일:

```
fdr = open("/dev/xillybus_read_32", O_RDONLY);
fdw = open("/dev/xillybus_write_32", O_WRONLY);

if ((fdr < 0) || (fdw < 0)) {
    perror("Failed to open Xillybus device file(s)");
    exit(1);
}
```

실제 실행은 fork()에서 두 개의 프로세스로 시작됩니다.

```
pid = fork();

if (pid < 0) {
    perror("Failed to fork()");
    exit(1);
}
```

아버지 프로세스는 처리할 데이터를 준비하고 FPGA에 기록합니다. 이 프로세스에서 사용하지 않기 때문에 `read file descriptor`를 닫습니다. 열린 상태로 유지하면 두 프로세스가 모두 `file descriptor`를 닫을 때까지(또는 종료될 때까지) `device file`이 열린 상태로 유지됩니다. 이는 여기서 원하는 동작이 아닙니다.

```
if (pid) {
    close(fdr);

    tologic = malloc(sizeof(struct packet) * N);
    if (!tologic) {
        fprintf(stderr, "Failed to allocate memory\n");
        exit(1);
    }
}
```

다음으로 구조체 배열을 데이터로 채웁니다. 이것은 처리를 위해 각 데이터 세트에 대한 구조를 정의하는 것이 의미 있는 이유를 설명합니다.

```
// Fill array of structures with just some numbers
da = 6.283185 / ((float) N);

for (i=0, a=0.0; i<N; i++, a+=da) {
    tologic[i].v1 = i;
    tologic[i].v2 = a;
}

buf = (char *) tologic;
```

“buf”는 구조의 배열을 가리키는 pointer에서 `char`의 `buffer`로 정의됩니다. 데이터를 보내는 `while` 루프는 `buffer`를 전송할 데이터 청크로 취급하기 때문에 이 변환이 필요합니다.

다음은 데이터 쓰기를 위한 `while` 루프입니다. 불필요하게 복잡해 보일 수 있지만 데이터가 안정적으로 기록되도록 하는 가장 짧은 방법입니다. 실제 응용 프로그램에서와 같이 이 코드를 채택하는 것이 좋습니다.

```

donebytes = 0;

while (donebytes < sizeof(struct packet) * N) {
    rc = write(fdw, buf + donebytes,
              sizeof(struct packet) * N - donebytes);

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

    if (rc <= 0) {
        perror("write() failed");
        exit(1);
    }

    donebytes += rc;
}

```

이 예에서는 단일 청크만 전송되고 다른 쪽 끝에서 수신됩니다. 실제 코드에서는 위의 두 코드를 반복하는 것이 맞습니다.

성능 테스트는 32 kBytes의 청크 크기가 일반적으로 최상의 결과를 제공하는 것으로 나타났습니다.

이 예에서는 하나의 청크만 전송되므로 프로세스가 종료됩니다. 파일을 닫기 전에 1초 동안 휴면하면 모든 데이터가 삭제되기 전에 logic이 재설정되지 않습니다. 이것은 ap\_rst\_n이 to\_host\_read\_32\_open로 가고 from\_host\_write\_32\_open이 전혀 연결되지 않았기 때문에 block design이 섹션 6.3에 표시된 것과 같을 때 의미가 없습니다.

그럼에도 불구하고 이것은 빨리 종료해야 하는 경우가 아니면 file descriptor를 즉시 닫지 않는 좋은 규칙을 보여줍니다. 이렇게 하면 프로젝트가 더 복잡해질 때 혼란을 줄일 수 있습니다.

```

sleep(1); // Let the output drain

close(fdw);
return 0;

```

다음으로 비슷한 방식으로 시작하는 자식 프로세스가 있습니다.

```
} else {
    close(fdw);

    fromlogic = malloc(sizeof(struct packet) * N);
    if (!fromlogic) {
        fprintf(stderr, "Failed to allocate memory\n");
        exit(1);
    }

    buf = (char *) fromlogic;
```

다시 한 번, 이것은 device file에서 데이터를 읽을 때 권장되는 방법입니다.

```
donebytes = 0;

while (donebytes < sizeof(struct packet) * N) {
    rc = read(fdr, buf + donebytes,
             sizeof(struct packet) * N - donebytes);

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

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

    if (rc == 0) {
        fprintf(stderr, "Reached read EOF!? Should never happen.\n");
        exit(0);
    }

    donebytes += rc;
}
```

그런 다음 데이터가 인쇄됩니다.

```

for (i=0; i<N; i++)
    printf("%d: %f\n", fromlogic[i].v1, fromlogic[i].v2);

sleep(1); // Let the output drain

close(fdr);
return 0;
}
}

```

다시 한 번, 프로세스는 file descriptor를 닫기 전에 1초 동안 휴면하며, 다시 한 번 이 특정 경우에는 필요하지 않습니다. file descriptor를 닫으면 실제로 logic이 재설정되지만 이 경우에는 모든 출력을 가져왔기 때문에 무해합니다. 이 지점에 도달할 때까지.

앞에서 언급했듯이 빨리 종료하는 것이 도움이 되지 않는 한 이 1초의 절전 모드는 특히 디버깅을 위해 다른 출력 streams가 생성되는 경우 혼란을 줄일 수 있습니다.

## 6.8 Design 고려 사항

### 6.8.1 여러 AXI streams 작업

예제 프로젝트는 각 방향으로 하나의 stream의 기본 사례를 보여줍니다. 그러나 AXI streams로 선언하기 위한 pragmas와 함께 래퍼 함수에 인수를 추가하여 IP block에서 입력 및/또는 출력을 위해 streams를 추가하는 것은 간단합니다.

예를 들어, 하나가 아닌 세 개의 입력 streams:

```

void xillybus_wrapper(int *d1, int *d2, int *d3, int *data_out) {
#pragma AP interface axis port=d1
#pragma AP interface axis port=d2
#pragma AP interface axis port=d3
#pragma AP interface axis port=data_out
#pragma AP interface ap_ctrl_none port=return

    *data_out++ = thefunc(*d1++, *d2++, *d3++);
}

```

5 섹션에 설명된 대로 사용자 지정 IP core를 구성하여 Xillybus IP core에 streams를 추가하는 것도 똑같이 간단합니다.

추가 streams는 특히 다음과 같은 다양한 시나리오에서 유용할 수 있습니다.

- 데이터 및 메타 정보를 별도의 streams로 전송합니다. 예를 들어 데이터를 패킷으로 분할해야 하는 경우 해당 길이를 전용 stream로 보내고 데이터를 다른 전용 stream로 보내십시오. 이렇게 하면 길이가 알려지기 전에 패킷의 시작 부분을 보낼 수 있습니다.
- 서로 다른 이미지의 픽셀 스캔과 같이 자연스럽게 별도로 배열된 데이터 전송(아래에서 자세히 설명).
- 디버깅: 검증을 위해 중간 데이터를 host로 보냅니다.

여러 streams로 작업할 때 모든 것을 염두에 두는 것이 중요합니다. logic의 실행 흐름은 입력 stream에 데이터가 부족하거나 출력 stream의 해당 device file이 열리지 않거나 데이터가 있는 overflow가 있는 경우 중단될 수 있습니다. 이것은 출력 stream이 디버깅을 위한 것이라면 특히 중요합니다. 정상적인 작동을 위해 시스템을 사용할 때 디버깅을 위한 stream을 잊어버리기 쉽습니다. 이 stream의 데이터가 사용되지 않기 때문에 일반적으로 몇 번의 데이터 주기 후에 혼란스러운 실행 중단이 발생합니다.

언뜻 보기에 부적절해 보일 수 있는 방식으로 logic 하드웨어에 데이터를 공급하는 것이 종종 합리적입니다. 예를 들어, 위에 표시된 3개 입력 예제는 각 작업에 대해 3개의 데이터 요소가 필요한 이미지 처리 알고리즘에 유용할 수 있습니다. 이미지가 왼쪽에서 오른쪽으로, 위에서 아래로 스캔된다고 가정합니다. 픽셀 출력을 생성하기 위해 알고리즘은 현재 이미지의 픽셀과 함께 이전 두 이미지의 각 픽셀이 필요합니다. 이 경우 현재 이미지를 stream 하나를 통해 FPGA로 보내고 이전 이미지 두 개를 다른 두 streams를 통해 병렬로 보낼 수 있습니다.

이것은 I/O 데이터 대역폭의 낭비와 불필요한 메모리 복사로 보일 수 있습니다. 특히, processor가 “shuffling data”에 너무 많이 관련되어 있다는 것은 잘못된 생각일 수 있습니다. 주관적인 인식은 제쳐두고 메모리 복사의 구현은 모든 최신 processor 아키텍처에서 고도로 최적화된 작업이며 processor에는 종종 다른 애플리케이션 관련 작업이 로드되어 메모리 복사 로드를 무시할 수 있습니다.

따라서 logic에 데이터를 직접 공급하는 것이 리소스 활용 측면에서 차선책이지만 processor에 일반적으로 처리해야 할 다른 고중량 작업이 있다는 점을 고려할 때 processor의 추가 로드는 일반적으로 다소 작습니다. 이것은 종종 design을 상당히 단순화하기 위한 합리적인 가격입니다.

### 6.8.2 application clock의 주파수

HLS에 의해 생성된 logic은 stream\_clk\_gen의 block에 의해 생성된 block design의 application clock에 의해 구동됩니다. 이 clock은 logic의 타임베이스이므로 실행 속도는 clock의 주파수에 비례합니다. AXI stream 포트의 데이터 전송이 병목 현상이 되지 않는 한 application clock 주파수가 높을수록 처리 처리량이 비례적으로 빨라집니다.

그러나 FPGA의 logic 리소스와 필요한 작업을 구현하는 데 어떻게 활용되었는지에 따라 application clock의 주파수가 얼마나 높을 수 있는지에 대한 제한이 있습니다. 다음은 design 프로세스의 관련 이정표입니다.

1. Vivado HLS를 사용하면 사용자가 design에 대해 clock의 의도된 주파수를 설정하여 application clock에 대해 원하는 주파수를 지정할 수 있습니다(Solution >Solution Settings 포함). 이 매개변수는 HLS에서 단지 힌트로 사용되어 필요할 때 더 빠른 logic을 생성하기 위해 추가 노력을 기울일 수 있습니다.
2. Vivado HLS가 compilation을 완료하면 도달할 수 있는 clock의 주파수 추정치를 표시합니다(HLS의 GUI에 있는 Synthesis 탭의 “Performance Estimates” 섹션에 있는 “Timing” 아래).
3. 사용자는 3.2 섹션(특히 3.2.2 섹션)에 설명된 대로 Vivado의 block design에서 application clock의 주파수를 설정합니다. 자연스러운 선택은 항목 2에서 추정된 clock의 주파수 또는 그 이하입니다. 이것은 Vivado HLS가 아닌 Vivado에서 수행됩니다.
4. Vivado가 전체 design의 implementation을 FPGA용 bitstream로 완료하면 clock와 관련된 모든 요구 사항을 충족하도록 logic을 구성하는 데 성공했는지 여부를 사용자에게 알려줍니다. 여기에는 항목 3에 설정된 clock의 주파수를 충족하는 것이 포함됩니다.

그래서 그것은 마지막 이정표로 요약되며, Vivado가 항목 3에서 선택한 application clock의 주파수와 관련된 timing constraints를 충족할 수 있었다면.

HLS 및 stream\_clk\_gen의 기본 clock period는 10 ns( 100 MHz )입니다. 다음과 같은 경우가 아니라면 이 선택을 유지하는 것이 가장 좋습니다.

- Vivado는 timing constraints를 충족하지 못하며, 이 경우 더 느린 clock을 선택해야 합니다.
- 처리 처리량을 늘리려는 동기가 있는 경우 더 빠른 clock을 요구하는 시도가 이루어져야 합니다. 이것은 종종 clock의 주파수를 조정하고 개선된 결과에 도달하기 위해 design 자체 및 HLS pragmas를 변경하는 반복적인 프로세스입니다.

### 6.8.3 logic 재설정

C/C++ 코드가 logic로 변환되면 실제로 실행되지 않고 자체 실행 흐름의 상태를 유지합니다. logic이 processor의 프로그램 실행 동작을 모방하도록 하려면 무엇보다도 프로그램의 시작 부분에서 실행이 시작되는지 확인하는 것이 중요합니다. 이것은 logic을 재설정함으로써 달성됩니다.

대부분의 경우 직관적인 동작은 FPGA의 프로그램이 host의 프로그램이 실행을 시작할 때 처음부터 시작한다는 것입니다. host에서 실행되는 모든 프로세스는 액세스하기 전에 device files를 열고 이러한 파일은 적어도 프로세스가 종료될 때 반드시 닫히므로 하나 이상의 device files가 닫힐 때 logic을 재설정하는 것이 당연합니다.

Xillybus IP Core의 각 stream에는 \*\_open 포트가 있으며 해당 device file이 열릴 때 하이('1')입니다. HLS block에는 액티브 로우 리셋 입력 ap\_rst\_n(기본값)가 있으므로 \*\_open 출력을 ap\_rst\_n 입력에 직접 연결하면 원하는 결과를 얻을 수 있습니다. 파일이 닫힐 때 \*\_open 신호는 로우('0')입니다. 이것은 logic을 리셋 상태로 유지합니다.

모든 device files가 열릴 때까지 또는 그 중 하나가 열릴 때까지 logic을 유지하기 위해 여러 \*\_open 포트를 결합하는 것이 바람직할 수 있습니다. 이것은 Vivado의 IP catalog에서 사용할 수 있는 간단한 logic gate blocks를 추가하여 달성됩니다. 리셋 신호를 생성하는 방법의 선택은 host 프로그램이 설정되는 방법에 따라 다릅니다.

어느 쪽이든 host가 리셋 신호가 비활성화되도록 하기 위해 필요에 따라 device files를 열 때까지 HLS block와 데이터 교환을 시도하지 않는지 확인하는 것이 중요합니다. 단순성을 위해 데이터 교환을 시작하기 전에 HLS block와 관련된 모든 device files를 열고 정리를 위해 모두 닫는 것이 가장 좋습니다.