(기계로 한국어 번역)

Xillybus FPGA designer's guide

Xillybus Ltd.

www.xillybus.com

Version 3.2

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 This document may also be slightly outdated in relation to the original.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

Xillybus의 IP cores는 FIFO 또는 dual-port block RAM을 통해 user application logic와 인터페이스하기 위한 것입니다. 따라서 일반적으로 API를 자세히 이해하지 않고 IP core로 작업하는 것이 가능합니다.

API 규칙의 대부분은 간단한 원칙에서 추론할 수 있습니다. user application logic은 FIFO 또는 block RAM와 똑같이 작동해야 합니다. 이는 application logic이 IP core와 직접 인터페이스하는 경우에도 마찬가지입니다.

무엇보다도 이 원칙은 Xillybus IP core와 user application logic 간의 데이터 교환이 IP core가 지시하는 속도로 발생함을 의미합니다. 데이터 흐름이 일시적으로 중지되고 예측할 수 없는 방식으로 나중에 재개될 수 있습니다. 다른 상황에서는 데이터 흐름이 계속됩니다. 이것은 IP core가 FIFO 또는 block RAM에 직접 연결될 때 문제가 되지 않습니다. 마찬가지로 IP core와 직접 인터페이스하는 user application logic도 예측할 수 없는 방식으로 데이터 흐름이 시작되고 중지되는 경우에도 제대로 작동해야 합니다.

IP core의 불규칙한 액세스 패턴을 버그로 간주하는 것은 일반적인 실수입니다. IP core와 FIFO 사이의 데이터 흐름에서 설명할 수 없는 일시 중지가 의심스러워 보일 수 있지만 오작동을 나타내지는 않습니다.

logic consumption을 줄이기 위해 IP core와 직접 인터페이스(예: FIFO 또는 block RAMs 없이)하는 것은 권장되지 않습니다. 이것은 특히 design의 초기 단계에서 사실입니다. user application logic이 IP core에 직접 연결된 경우 불규칙한 데이터 흐름으로 인해 application logic의 버그가 노출될 수 있습니다.

이 안내서는 application logic을 직접 인터페이스하는 것과 관련된 API에 대해 설명하고 널리 사용되는 응용 프로그램에 대해서도 자세히 설명합니다.

이 가이드에 제시된 세부 정보를 살펴보기 전에 Xillybus에 대한 초기 경험을 얻는 것이 좋습니다. 다음 문서를 참조하십시오.

- Getting started with the FPGA demo bundle for Xilinx
- Getting started with the FPGA demo bundle for Intel FPGA
- Getting started with Xillinux for Zynq-7000
- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

이 가이드는 또한 사용자가 동기식 streams와 비동기식 streams의 차이점을 이해하고 있다고 가정합니다. 이에 대해서는 다음 두 문서 중 하나의 섹션 2에서 설명합니다.

- Xillybus host application programming guide for Linux
- Xillybus host application programming guide for Windows

XillyUSB IP cores는 동일한 API를 노출하며 Xillybus IP cores의 하위 집합입니다. 따라서 별도의 언급이 없는 한 “Xillybus”라는 이름은 이 안내서에서도 XillyUSB IP cores를 의미합니다.

궁금한 분들을 위해 Xillybus 구현 방법에 대한 간략한 설명은 Xillybus host application programming guide for Linux 또는 Xillybus host application programming guide for Windows의 부록 A에서 찾을 수 있습니다.

2

일반 지침

2.1 Clocking

Xillybus IP core와 주고받는 모든 signals는 bus_clk의 rising edge와 동기화되어야 합니다. 이 clock은 IP core에서 제공합니다.

PCIe를 기반으로 하는 Xillybus IP cores의 경우 이 clock은 PCIe 블록에서 생성됩니다. clock의 주파수는 플랫폼에 따라 다릅니다. baseline IP core(개정판 A)의 경우 bus_clk의 빈도는 62.5 MHz, 125 MHz 또는 250 MHz입니다. 이는 최대 대역폭(공시된 대로)이 각각 200 MB/s, 400 MB/s 또는 800 MB/s인지에 따라 다릅니다.

최신 개정판(B, XL 및 XXL)에서 bus_clk의 빈도는 250 MHz입니다.

Zynq 기반 플랫폼에는 일반적으로 100 MHz의 bus_clk이 있습니다. XillyUSB는 125 MHz의 bus_clk와 함께 작동합니다.

제한된 선택 목록 내에서 clock의 주파수를 변경할 가능성이 종종 있습니다. 이는 PCIe 블록 또는 clock을 생성하는 processor core를 구성하여 수행됩니다.

PCIe 블록의 timing constraints가 올바르게 설정된 경우(demo bundles에서와 같이) bus_clk에 의존하는 application logic도 적절한 timing constraints에 의해 보호됩니다. 이 도구는 PCIe 블록의 timing constraints를 기반으로 하는 bus_clk용 timing constraints를 자동으로 생성합니다. Zynq 기반 플랫폼은 물론 XillyUSB에도 동일하게 적용됩니다.

application logic이 bus_clk와 동기화되어야 한다는 의미는 아닙니다. 마찬가지로 데이터 소스 또는 데이터 대상이 bus_clk와 동기화될 필요는 없습니다. 다른 clock이 관련된 경우 dual-clock FIFO는 종종 IP core와 함께 사용됩니다. FIFO의 한쪽은 Xillybus IP core에 연결됩니다. 따라서 이 쪽은 bus_clk와 동기화됩니다. application logic은 FIFO의 반대쪽에 연결됩니다. 이 면은 application logic의 clock와 동기식입니다. 따라서 FIFO는 단기 임시 스토리지뿐만 아니라 clock domain crossing에도 사용됩니다.

2.2 데이터 너비

각 FIFO 또는 메모리 인터페이스는 8비트, 16비트 또는 32비트 폭의 데이터로 작동합니다. 이는 기본 Xillybus IP cores(개정판 A)에 해당됩니다. 최신 개정판과 XillyUSB는 더 넓은 데이터 인터페이스를 지원합니다.

더 넓은 데이터는 더 높은 대역폭 성능을 허용하며 또한 자연 전송 워드가 8비트보다 넓은 애플리케이션에서 더 편리합니다. 반면 read() 및 write() 함수 호출은 길이를 바이트 단위로 정의하기 때문에 host 측의 고유 데이터 너비는 8비트(1바이트)로 유지됩니다.

데이터 너비 선택에 대한 고려 사항은 The guide to defining a custom Xillybus IP core에서 간략하게 설명됩니다.

2.3 FIFO를 통한 인터페이스

demo bundle은 FIFO를 연결하는 방법을 보여줍니다. 양쪽이 IP core에 연결된 FIFO가 있습니다. 이것은 두 개의 streams에서 loopback을 구현합니다.

demo bundle의 FIFOs는 양쪽에서 공통 clock용으로 구성됩니다. 이것은 FIFO가 clock domain crossing에 사용될 때 적합하지 않습니다. 이 경우 dual-clock FIFO("asynchronous FIFO"라고도 함)를 사용해야 합니다.

FIFO가 host에서 FPGA로 stream에 사용되는 경우 이 FIFO의 "full" 신호는 Xillybus IP core에 연결되어야 합니다. IP Core는 이 신호를 사용하여 버스트 데이터 전송을 시작할 수 있는지 여부를 결정합니다.

동일한 원칙이 FPGA에서 host까지 stream에 적용됩니다. "empty" 신호는 동일한 목적을 위해 Xillybus IP core에 연결되어야 합니다. IP core는 일반 FIFO의 동작을 예상합니다(FWFT, First Word Fall Through와 반대).

버스트가 시작되면 Xillybus IP core는 계속해서 다음 신호("empty" 및 "full")에 의존합니다. 이러한 신호는 IP core가 빈 FIFO에서 읽거나 전체 FIFO에 쓰는 것을 방지합니다.

그러나 FIFO가 데이터 버스트에 대한 준비가 되었음을 나타내더라도 Xillybus IP core는 버스트를 즉시 시작하지 않을 수 있습니다. IP core는 FIFO가 연속 버스트를 허용하더라도 중간에 데이터 버스트를 중지할 수도 있습니다. 데이터 흐름의 패턴이 분명히 무작위인 것은 정상입니다.

일반적인 규칙은 Xillybus IP core가 연결된 모든 FIFOs를 동일하게 제공하려고 시도한다는 것입니다. IP core는 FIFOs가 "empty" 또는 "full"을 자주 활성화하지 않기 때문에 더 빨리 채워지는 경향이 있는 FIFOs에 더 긴 버스트를 부여합니다.

이 간단한 중재 방법은 빠르게 채워지는 경향이 있는 FIFOs와의 효율적인 통신을 보장합니다. 동시에 더 낮은 속도로 데이터를 수신하는 FIFOs의 낮은 latency가 달성됩니다.

FIFO의 깊이는 원칙적으로 Xillybus IP core가 모든 깊이에서 작동합니다. 그러나 예상되는 데이터 흐름에 대처하려면 이 속성을 선택해야 합니다. 2 kBytes 깊이의 FIFO는 높은 데이터 속도에서도 거의 항상 비동기식 stream에 대한 올바른 선택입니다. 그러나 이것은 때때로 시행 착오의 문제입니다.

Xillybus core가 overflow 또는 underflow를 유발할 만큼 충분히 긴 시간 동안 이 크기의 FIFO를 무시할 가능성이 없기 때문에 2 kBytes의 깊이는 일반적으로 충분합니다. 물론 이것은 host에서 실행되는 user application software가 충분히 빠르게 데이터를 소비하거나 공급하는 한 사실입니다. 그렇지 않은 경우 솔루션은 DMA buffers를 더 크게 만드는 것일 수 있습니다. 더 큰 FIFO로 이 문제를 해결하려는 시도는 FPGA의 메모리가 훨씬 적기 때문에 비합리적입니다.

2.4 “empty” 및 “full” 신호의 동작

정상적으로 작동하는 FIFO에서 “empty” 신호는 read enable이 하이가 된 후 clock cycle 하나만 로우에서 하이로 변경될 수 있습니다. 마찬가지로 “full” 신호는 write enable이 하이가 된 후 clock cycle 하나만 로우에서 하이로 변경할 수 있습니다.

물론 이 두 신호는 언제든지 낮음으로 변경될 수 있습니다.

Xillybus IP core는 다음 동작에 의존합니다. FIFO가 데이터 전송 준비가 되었음을 나타내면(해당하는 경우 낮은 “empty” 또는 “full” 사용) IP core의 state machine이 일련의 이벤트를 시작할 수 있습니다. 이렇게 하면 최소한 하나의 데이터 요소가 전송됩니다. 따라서 IP core가 FIFO에서 데이터를 가져오기 전에 “empty” 신호가 높음으로 변경되면 IP core가 하나의 clock cycle 동안 “empty” 신호를 무시할 가능성이 있습니다. 이러한 이벤트는 IP core 자체의 무결성과 관련하여 무해하지만 예상치 못한 예측 불가능한 데이터 흐름으로 이어질 수 있습니다.

“full” 신호에도 동일하게 적용됩니다. IP core가 FIFO에 데이터 워드를 쓰기 전에 이 신호가 낮음에서 높음으로 변경되면 IP core는 하나의 clock cycle 동안 “full” 신호를 무시할 수 있습니다. 다시 한 번 말하지만 이것은 IP core 자체에는 무해하지만 데이터 워드 하나가 손실될 수 있습니다.

적절하게 설계된 FIFO는 Xillybus IP core와 통신할 준비가 됨과 동시에 재설정되는 경우에만 이러한 결함 조건을 생성할 수 있습니다. 어쨌든 이 상황은 일반적으로 피해야 합니다.

application logic이 IP core와 직접 연결된 경우(FIFO 없이) “empty” 또는 “full”와 관련하여 표준 FIFO의 동작을 모방하는 것이 중요합니다.

3

신호 설명

3.1 FPGA 신호의 명명 규칙

두 개의 전역 신호인 bus_clk 및 quiesce를 제외하고 모든 신호는 간단한 규칙을 따릅니다. 예를 들어, write enable 신호의 이름은 user_w_write_32_wren일 수 있습니다. 이 이름은 네 가지 구성 요소로 나뉩니다.

1. “user” 접두사는 모든 사용자 인터페이스 신호에 공통입니다.
2. “w” 부분은 이 신호가 host에서 FPGA(host “write”)로 stream에 속함을 나타냅니다. FPGA에서 host까지의 Streams에는 대신 “r”가 있습니다. 주소 신호는 양방향에 적용되기 때문에 이 부분이 없습니다. host의 관점은 “w” 또는 “r”의 선택과 관련하여 취해진 것입니다.
3. “write_32” 문자열은 관련 device file의 이름에 나타납니다./dev/xillybus_write_32 또는 /dev/xillyusb_00_write_32(해당되는 경우).
4. 접미사는 신호의 의미를 나타냅니다.

이 섹션의 나머지 부분에서는 혼동을 피하기 위해 device file 이름(세 번째 구성 요소)을 {devfile}로 표시합니다.

각 신호 이름 뒤에는 신호가 IP core에 대한 입력임을 나타내기 위해 (IN)이 붙고, IP core의 출력인 경우 (OUT)가 표시됩니다.

3.2 host에서 FPGA로의 전송을 위한 신호

- user_w_{devfile}_data (**OUT**) – 이 신호는 쓰기 주기 동안 데이터를 포함합니다.

- user_w_{devfile}_wren (**OUT**) – 이 신호는 FIFO에 대한 write enable 신호입니다. FIFO(또는 FIFO의 동작을 모방하는 다른 logic)에 기록되어야 하는 user_w_{devfile}_data 신호에 유효한 데이터가 있는 경우 하이입니다.

- user_w_{devfile}_full (**IN**) – 이 신호는 IP core에 더 이상 데이터를 쓸 수 없음을 알려줍니다.

  중요 : 'full' 신호는 쓰기 주기 후에 clock cycle에서만 낮음에서 높음으로 변경될 수 있습니다. 모든 표준 FIFOs는 이와 같이 작동하므로 이 규칙은 IP core가 application logic와 직접 연결된 경우(즉, 중간에 FIFO가 없는 경우)에만 관련됩니다.

  이 규칙의 이유는 Xillybus IP core가 낮은 'full' 신호를 녹색 신호로 취급하여 host에서 데이터 전송을 시작하기 때문입니다. 이 규칙을 준수하지 않으면 'full' 조건을 무시하는 산발적인 쓰기가 발생할 수 있습니다.

  'full' 신호의 일반적인 Verilog 구현은 다음과 같아야 합니다.

```
always @(posedge bus_clk)
  if (ready_to_get_more_data)
    user_w_mydevice_full <= 0; // Turn low any time
  else if (user_w_mydevice_wren && { ... some condition ... } )
    user_w_mydevice_full <= 1; // Only in conjunction with wren
```

  VHDL에서도 동일:

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if (ready_to_get_more_data = '1') then
      user_w_mydevice_full <= '0'; -- Turn low any time
    elsif (user_w_mydevice_wren = '1' and { some condition } )
      user_w_mydevice_full <= '1'; -- Turn high only with wren
    end if;
  end if;
end process;
```

- user_w_{devfile}_open (**OUT**) – 이 신호는 host의 관련 device file이 쓰기용으로 열려 있을 때 높음입니다(파일이 읽기 전용으로 열려 있는 경우 허용되는 경우 이 신호를 변경하지 않음). 이 신호는 파일이 닫힐 때 FIFO 또는 다른 logic을 재설정하는 데 사용될 수 있습니다(active low reset로 사용됨).

  host의 여러 processes에서 파일을 연 경우(예: fork() 기능 호출의 결과) 이 신호는 모든 processes가 파일을 닫을 때까지 높게 유지됩니다.

3.3 FPGA에서 host로의 전송을 위한 신호

- user_r_{devfile}_data (**IN**) – 이 신호는 읽기 주기 동안 데이터를 포함합니다. 이 신호는 FIFO가 변경했을 때만 변경할 수 있습니다. 즉, user_r_{devfile}_rden이 높으면 clock cycle에서만 변경될 수 있습니다.

- user_r_{devfile}_rden (**OUT**) – 이 신호는 FIFO에 대한 read enable 신호입니다. 이 신호가 높을 때 user_r_{devfile}_data는 다음 clock cycle에 유효한 데이터를 포함해야 합니다.

- user_r_{devfile}_empty (**IN**) – 이 신호는 더 이상 데이터를 읽을 수 없음을 core에 알립니다.

  중요 : 'empty' 신호는 읽기 주기 후에 clock cycle에서만 낮음에서 높음으로 변경될 수 있습니다. 모든 표준 FIFOs는 이와 같이 작동하므로 이 규칙은 IP core가 application logic와 직접 연결된 경우(즉, 중간에 FIFO가 없는 경우)에만 관련됩니다.

  이 규칙의 이유는 Xillybus IP core가 낮은 'empty' 신호를 녹색 신호로 취급하여 host로 데이터 전송을 시작하기 때문입니다. 이 규칙을 준수하지 않으면 FIFO가 비어 있음을 무시하는 산발적인 읽기가 발생할 수 있습니다.

  'empty' 신호의 일반적인 Verilog 구현은 다음과 같아야 합니다.

```
always @(posedge bus_clk)
  if (ready_to_give_more_data)
    user_r_mydevice_empty <= 0; // Turn low any time
  else if (user_r_mydevice_rden && { ... some condition ... } )
    user_r_mydevice_empty <= 1; // Turn high only with rden
```

  VHDL에서도 동일:

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if (ready_to_give_more_data = '1') then
      user_r_mydevice_empty <= '0'; -- Turn low any time
    elsif (user_r_mydevice_rden = '1' and { some condition } )
      user_r_mydevice_empty <= '1'; -- Turn high only with rden
    end if;
  end if;
end process;
```

- user_r_{devfile}_eof (**IN**) – 이 신호는 core가 end-of-file을 생성하도록 지시합니다. 높은 'eof'의 결과는 또한 core가 FIFO에서 더 이상 읽지 않는다는 것입니다(예: user_r_{devfile}_rden은 파일이 닫혔다가 다시 열릴 때까지 낮게 유지됨).

  host에서 application software는 이 신호가 높아지기 전에 IP core가 수신한 모든 데이터 읽기를 완료합니다. 그래야만 host가 read() 기능을 호출할 때 EOF를 수신합니다.

  'eof' 신호는 application software에서 읽지 않은 데이터가 여전히 있는 경우 host에서 EOF를 즉시 발생시키지 않습니다. host에서 EOF를 제공하는 것은 상식적으로, 즉 host에서 모든 데이터를 읽은 후에 이루어집니다.

  'eof' 신호가 하이가 된 후에는 하이를 유지하거나 로우로 변경하는 것은 중요하지 않습니다. IP core는 파일이 닫힐 때까지 EOF 요청을 기억합니다. 'empty' 신호는 'eof'가 high로 바뀌는 것과 동시에 high로 바뀌더라도 상관없습니다. 사실 'eof'가 하이인 순간부터 'empty' 신호는 파일이 닫힐 때까지 전혀 문제가 되지 않습니다.

  'empty' 신호와 유사하게 'eof' 신호는 읽기 주기 후에 clock cycle에서만 하이로 변경할 수 있습니다. 그러나 한 가지 예외가 있습니다. 'empty' 신호가 이미 하이일 때 'eof'는 언제든지 하이로 변경될 수 있습니다. 이 예외는 데이터를 기다리는 동안 휴면 상태인 경우 host에서 read() 함수 호출을 즉시 종료하는 데 사용할 수 있습니다.

  이 규칙을 따르지 않고 'eof'를 높음으로 변경하면 EOF가 생성되지만 정확하게 작동하지 않을 수 있습니다. 일부 데이터는 EOF 직전에 손실되거나 관련 없는 데이터가 EOF 이전 또는 EOF 이후에 추가될 수 있습니다(따라서 application software는 EOF 이후에 데이터를 수신하므로 불법임).

  'eof'가 이 규칙을 준수하는지 확인하는 한 가지 가능성은 'eof'를 combinatorial function의 출력으로 정의하는 것입니다. Verilog에서는 다음과 같이 작성할 수 있습니다.

```
assign user_r_mydevice_eof = user_r_mydevice_empty && [ ... ];
```

  또는 VHDL에서:

```
user_r_mydevice_eof <= user_r_mydevice_empty and [ ... ];
```

  이 방법을 사용하면 'empty'가 로우일 때 'eof' 신호가 항상 로우입니다.

- user_r_{devfile}_open (**OUT**) – 이 신호는 host의 관련 device file이 읽기용으로 열려 있을 때 높음입니다(파일이 쓰기 전용으로 열려 있는 경우 허용되는 경우 이 신호를 변경하지 않음). 이 신호는 파일이 닫힐 때 FIFO 또는 다른 logic을 재설정하는 데 사용될 수 있습니다(active low reset로 사용됨).

host의 여러 processes에서 파일을 연 경우(예: fork() 함수 호출의 결과) 이 신호는 모든 processes가 파일을 닫을 때까지 높게 유지됩니다.

'eof' 신호와 'open' 신호 사이에는 직접적인 연결이 없습니다. 'eof'와 관계없이 host에서 파일이 닫히면 'open' 신호가 낮음으로 변경됩니다. 그러나 application software는 일반적으로 파일을 닫음으로써 EOF에 응답합니다. 따라서 이러한 신호 사이에 연결이 있다고 잘못 믿기 쉽습니다.

3.4 메모리 인터페이스 신호

Xillybus device file은 주소 신호를 갖도록 구성할 수 있습니다. application software는 파일에서 seeking용 표준 API를 사용하여 이 신호에 값을 할당합니다(예: lseek()). 또한 주소의 increment는 읽기 주기 및 쓰기 주기의 결과로 FPGA에서 자동으로 발생합니다.

표준 block RAM은 IP core와 쉽게 연결됩니다. 이것은 FIFOs와 관련하여 위에서 이미 언급한 신호와 아래에 자세히 설명된 신호를 사용하여 수행됩니다. 그 결과 block RAM의 메모리 어레이를 host에서 파일로 사용할 수 있습니다. 파일에 대한 읽기 및 쓰기 작업은 메모리 어레이에 대한 읽기 및 쓰기 작업으로 이어집니다. host는 메모리 어레이의 단일 메모리 요소 또는 세그먼트에 액세스할 수 있습니다. 이는 읽기 또는 쓰기 작업의 길이에 따라 다릅니다.

FPGA에서 block RAM처럼 작동하는 registers 어레이를 구현하는 것도 가능합니다. 이렇게 하면 registers가 host에서 쉽게 액세스할 수 있습니다.

'empty' 및 'full' 신호는 wait states가 필요한 메모리의 읽기 및 쓰기 작업 속도를 늦추거나 작업을 잠시 지연시켜야 하는 또 다른 이유가 있을 때 사용할 수 있습니다.

메모리 인터페이스에는 두 가지 추가 신호가 필요합니다.

- user_{devfile}_addr (**OUT**) – 이 신호에는 현재 주소가 포함되어 있습니다. read enable이 높으면 이 주소에서 읽기 작업이 필요합니다. write enable이 높으면 이 주소에 대한 쓰기 작업이 필요합니다. 이 신호를 block RAM의 address input에 직접 연결하면 예상대로 작동합니다. 이 신호의 폭은 최대 32비트까지 구성할 수 있습니다.

  주소 값은 최대 주소(주소 신호의 폭에 따라 다름)에서 읽기 또는 쓰기 작업 후에 0으로 돌아갑니다. lseek()에 대한 함수 호출 값이 범위를 벗어나면 LSBs만 이 신호에 복사됩니다.

- user_{devfile}_addr_update (**OUT**) – 이 신호는 host에서 lseek()에 대한 함수 호출의 결과로 하나의 clock cycle 동안 높음입니다. 'addr' 신호가 업데이트된 값을 가지므로 동일한 clock cycle에서 'update' 신호가 높음입니다.

이 신호의 목적은 application logic이 주소 업데이트의 결과로 데이터 읽기를 준비하는 데 시간이 필요함을 나타낼 기회를 주는 것입니다. 이는 이러한 업데이트에 대한 응답으로 'empty' 신호를 높음으로 변경하여 수행됩니다.

이를 위해 'empty'는 읽기 주기 후에 하나의 clock cycle만 high로 변경할 수 있다는 규칙에 대한 한 가지 예외가 있습니다. 'update'가 높음 이후인 clock cycle에서도 높음으로 변경될 수 있습니다.

따라서 다음 Verilog 코드는 정확합니다.

```
always @(posedge bus_clk)
  if ( { ... memory is ready ... } )
    user_r_mydevice_empty <= 0;
  else if ((user_mydevice_addr_update) &&
           ( user_mydevice_addr > { ... some limit ...} ))
    user_r_mydevice_empty <= 1;
```

VHDL에서도 마찬가지입니다.

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if ( { ... memory is ready ... } ) then
      user_r_mydevice_empty <= '0';
    elsif (user_mydevice_addr_update = '1'
          and user_mydevice_addr > { ... some limit ...} )
      user_r_mydevice_empty <= '1';
    end if;
  end if;
end process;
```

'empty'는 언제든지 로우로 변경될 수 있으므로 모든 주소 업데이트(주소에 관계없이)의 결과로 'empty'를 하이로 변경한 다음 'empty'를 다시 로우로 변경할 수 있는지 logic이 평가하도록 하는 것이 합리적입니다.

'full' 신호도 유사한 방식으로 높음으로 변경될 수 있지만 이것이 왜 유용한지 명확하지 않습니다.

host의 관련 device file이 닫히면(즉, user_w_{devfile}_open와 user_r_{devfile}_open이 모두 낮을 때) 주소가 재설정되므로 값이 0으로 변경됩니다. 그러나 이것은 주소 업데이트로 간주되지 않습니다. 즉, user_{devfile}_addr_update는 낮게 유지됩니다.

3.5 quiesce 신호

host가 IP core가 완전히 비활성화될 것으로 예상할 때 quiesce 신호는 하이입니다(quiescent state). 일반적으로 다음과 같은 경우입니다.

- host가 아직 driver를 로드하지 않았거나 host가 host를 언로드했습니다.
- Windows에서: host가 hibernation에 진입하려고 할 때.
- XillyUSB 사용 시: 또한 장치가 컴퓨터에 전혀 연결되지 않은 경우에도 마찬가지입니다.

이 신호의 의도는 synchronous reset로 사용되는 것이지만 이 신호는 거의 필요하지 않습니다. IP core가 비활성 상태(예: quiescent state)이면 모든 파일이 닫힙니다. 따라서 application logic은 *_open 신호만을 reset 신호로 사용할 수 있습니다. 'quiesce' 신호는 reset의 보다 글로벌한 형태로 사용될 수 있습니다.

4

data acquisition 구현

4.1 소개

예를 들어 FPGA에서 컴퓨터로 데이터를 캡처해야 하는 경우가 자주 발생합니다.

- video 신호 소스의 Frame grabbing.
- 아날로그-디지털 변환기(ADC)의 데이터.
- FPGA에서 디버그 정보를 수신합니다.

이와 같은 애플리케이션의 경우 데이터 속도가 높을 수 있습니다. 그럼에도 불구하고 데이터 흐름의 연속성은 보장되어야 합니다. 데이터 손실은 허용되지 않습니다.

data acquisition 애플리케이션은 FIFO에 데이터를 기록함으로써 Xillybus로 쉽게 구현됩니다. 이 섹션에서는 host에 도착하는 데이터가 연속적임을 보장하는 방법에 중점을 둡니다.

이론적으로 운영 체제가 application software에서 CPU를 원하는 만큼 빼앗을 수 있기 때문에 주변 장치와 컴퓨터 간에 지속적인 데이터 속도를 보장하는 것은 불가능합니다.

그럼에도 불구하고 연속적인 stream 데이터를 유지하는 방법이 있습니다. 이 목표를 달성하기 위한 첫 번째이자 분명한 조건은 필요한 속도로 데이터를 전송할 수 있는 Xillybus stream을 사용하는 것입니다. 또한 특정 host programming 기술을 사용해야 합니다. 이 문제는 두 프로그래밍 가이드에서 광범위하게 논의됩니다.

- Xillybus host application programming guide for Linux
- Xillybus host application programming guide for Windows

특히, 높은 데이터 전송률로 작업하는 방법을 설명하는 이 두 가이드의 섹션 4에 주의하십시오.

고대역폭 응용 프로그램의 경우 다음 두 가지 가이드 중 하나의 섹션 5를 참조하는 것이 좋습니다. 여기에는 다음과 같은 몇 가지 주제가 포함되어 있습니다.

- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

그러나 design이 완벽하게 수행되더라도 데이터 stream의 연속성이 깨질 가능성이 항상 있습니다. 운영 체제의 특성상 application software에서 CPU를 장기간 박탈하는 것이 허용됩니다.

따라서 첫 번째 목표는 데이터 stream의 연속성이 실제로 깨지지 않도록 하는 것입니다. 두 번째 목표는 모든 노력에도 불구하고 이런 일이 발생하면 이 이벤트가 주목되도록 하는 것입니다. 더 중요한 것은 host에 도착하는 모든 데이터가 연속적이라는 점입니다.

이 두 번째 목표를 달성하기 위해 application logic은 연속성이 끊어지는 지점에서 데이터 흐름을 중지해야 합니다. 이 시점 이후에 EOF가 host로 전송되어 host에 문제가 발생했음을 알립니다. 이러한 방식으로 application software는 도착하는 데이터가 실제로 연속적이라는 사실에 의존할 수 있습니다.

이상적으로는 이 정지 메커니즘이 활성화되지 않아야 합니다. 그러나 그럴 때 문제를 인식하고 해결할 수 있는 기회를 제공합니다.

다음에서는 Xillybus를 사용하여 연속 소스에서 데이터를 캡처하는 방법을 보여줍니다. 이 섹션의 강조점은 host에 도착하는 모든 데이터가 데이터 소스의 신뢰할 수 있는 사본임을 확인하는 것입니다.

4.2 예제 코드

정지 메커니즘을 포함하기 위해 design을 수정하는 방법에는 두 가지가 있습니다.

- 연속성이 깨지면 데이터 흐름을 중지하는 수정된 FIFO를 사용하십시오. 이 수정된 FIFO는 표준 FIFO의 래퍼입니다. 또한 EOF 신호도 생성합니다.
- FIFO를 사용하는 logic을 수정합니다.

아래에 표시되고 설명된 예제 코드는 이 link에서 다운로드할 수 있습니다.

http://xillybus.com/downloads/xillycapture.zip

zip 파일은 세 가지 파일로 구성됩니다.

- Verilog로 작성된 eof_fifo.v

- Verilog로 작성된 xillycapture.v
- VHDL로 작성된 xillycapture.vhd

예제를 시험해 보는 방법에는 두 가지가 있습니다. 두 가지 방법 모두에서 xillydemo.v 또는 xillydemo.vhd를 편집해야 합니다.

첫 번째 가능성은 eof_fifo.v를 사용하는 것입니다. fifo_32x512의 instantiation을 eof_fifo의 instantiation로 교체하세요. 그런 다음 user_r_read_32_eof를 이 FIFO의 eof port에 연결합니다.

또한 loopback 대신 데이터 소스를 생성하는 것이 좋습니다. 이는 테스트용으로만 만들어진 가짜 데이터 소스일 수 있습니다(예: counter).

두 번째 가능성은 xillycapture.v 또는 xillycapture.vhd를 사용하는 것입니다. demo bundle에서 read_32와 관련된 신호를 분리하고 대신 이 파일 중 하나에 예제 코드를 삽입하십시오.

이 두 파일에는 “slowdown”라는 신호가 있습니다. 이 신호의 목적은 가짜 데이터 소스의 데이터 속도를 줄이는 것입니다. 실제 데이터 소스를 사용할 때는 이 신호를 제거해야 합니다.

두 가지 가능성 모두에서 예제 코드는 표준 dual clock FIFO의 instantiation을 수행합니다. 이 FIFO의 너비는 32 bits입니다. 예제 코드의 synthesis를 수행하기 전에 도구(예: Vivado 또는 Quartus)를 사용하여 이 FIFO를 생성하세요. 이 FIFO의 이름은 async_fifo_32여야 합니다. 512 단어의 깊이이면 충분합니다.

이 섹션의 나머지 부분은 xillycapture.v 및 xillycapture.vhd를 기반으로 합니다. 그러나 설명된 원리는 eof_fifo.v를 이해하는 데에도 관련이 있습니다.

4.3 FIFO 연결

데이터 소스가 capture_clk와 동기화되어 있다고 가정해 보겠습니다. 따라서 데이터는 일반 방식으로 표준 dual-clock FIFO에 연결됩니다. 이 FIFO는 데이터 소스와 Xillybus IP core를 연결합니다.

Verilog에서:

```
 async_fifo_32 fifo_32
   (
    .rst(!user_r_read_32_open),
    .wr_clk(capture_clk),
    .rd_clk(bus_clk),
    .din(capture_data),
    .wr_en(capture_en),
    .rd_en(user_r_read_32_rden),
    .dout(user_r_read_32_data),
    .full(capture_full),
    .empty(user_r_read_32_empty)
    );
```

그리고 VHDL에서:

```
fifo_32 : async_fifo_32
  port map(
    rst         => reset_32,
    wr_clk      => capture_clk,
    rd_clk      => bus_clk,
    din         => capture_data,
    wr_en       => capture_en,
    rd_en       => user_r_read_32_rden,
    dout        => user_r_read_32_data,
    full        => capture_full,
    empty       => user_r_read_32_empty
    );

reset_32 <= not user_r_read_32_open;
```

이것은 demo bundle와 매우 유사합니다. FIFO는 파일이 닫히면 재설정되고 user_r_read_32_* 신호는 demo bundle와 같이 연결됩니다.

4.4 Data acquisition 컨트롤

capture_en 신호는 write enable 신호로 작동합니다. FIFO에 데이터 쓰기를 방지하는 세 가지 상황이 있습니다.

- 파일이 닫힐 때
- FIFO가 꽉 찼을 때

- 파일을 연 이후 FIFO가 과거에 가득 찼을 때

따라서 capture_en(Verilog에서)의 조건은 다음과 같이 요약됩니다.

```
assign capture_en = capture_open && !capture_full &&
                    !capture_has_been_full ;
```

그리고 VHDL에서:

```
  capture_en <= capture_open and not capture_full
                and not capture_has_been_full ;
```

capture_open 신호는 capture_clk의 clock domain용 user_r_read_32_open의 복사본입니다.

실제 애플리케이션에서는 종종 FIFO에 쓰기 위한 다른 조건이 있습니다. 예를 들어, video frame의 시작을 기다리거나 특정 오류 조건을 기다리십시오(디버깅을 위해 data acquisition을 사용하는 경우). 이러한 종류의 조건은 필요에 따라 이 표현식에 추가할 수 있습니다(logic AND 덕분에).

신호 capture_has_been_full은 FIFO가 가득 차면 높음으로 변경되고 파일이 닫힐 때만 다시 낮음으로 돌아갑니다. 따라서 FIFO가 가득 차면 data acquisition이 중지되고 파일이 열려 있는 한 다시 시작되지 않습니다.

중요한:

예제 코드에는 가짜 데이터 소스의 속도를 늦추는 데 도움이 되는 capture_en에 대한 다른 정의가 있습니다. 실제 적용을 위해서는 capture_en을 위와 같이 변경해야 합니다.

이제 Verilog에서 capture_has_been_full을 구현하는 코드:

```
always @(posedge capture_clk)
  begin
    if (!capture_full)
      capture_has_been_nonfull <= 1;
    else if (!capture_open)
      capture_has_been_nonfull <= 0;

    if (capture_full && capture_has_been_nonfull)
      capture_has_been_full <= 1;
    else if (!capture_open)
      capture_has_been_full <= 0;
   end
```

그리고 VHDL:

```
process (capture_clk)
begin
  if (capture_clk'event and capture_clk = '1') then
    if  ( capture_full = '0' ) then
      capture_has_been_nonfull <= '1' ;
    elsif  ( capture_open = '0' ) then
      capture_has_been_nonfull <= '0' ;
    end if;

    if (capture_full = '1' and capture_has_been_nonfull = '1') then
      capture_has_been_full <= '1' ;
    elsif  ( capture_open = '0' ) then
      capture_has_been_full <= '0' ;
    end if;

  end if;
end process;
```

FIFO의 capture_full이 높아지면 capture_has_been_full도 높아집니다. 파일이 닫히면 capture_has_been_full이 낮아집니다.

다른 신호인 capture_has_been_nonfull은 다른 문제를 해결합니다. FIFO의 'full' 신호는 FIFO가 재설정되는 한 높음입니다. 이러한 이유로 'full' 신호가 높으면 capture_has_been_full은 높지 않아야 합니다. 즉, capture_has_been_full은 capture_full이 로우(FIFO가 재설정에서 벗어남을 의미)한 다음 하이(FIFO가 실제로 가득 찼음을 의미)일 때만 하이여야 합니다.

따라서 이 코드는 약간 복잡하지만 일단 원리를 이해하면 매우 간단합니다.

4.5 EOF 생성

다음 두 가지 조건이 충족되면 end-of-file이 생성됩니다.

- FIFO의 모든 데이터가 소비되었습니다(즉, IP core에서 모든 데이터를 읽음).
- FIFO가 과거에 가득 차 있었기 때문에 더 이상 데이터가 FIFO에 기록되지 않습니다.

Verilog에서는 다음과 같이 작성됩니다.

```
assign user_r_read_32_eof = user_r_read_32_empty && has_been_full;
```

그리고 VHDL에서(이것은 combinatorial function임에 유의하십시오):

```
user_r_read_32_eof <= user_r_read_32_empty and has_been_full;
```

예제 코드에서 볼 수 있듯이 has_been_full은 clock domain crossing을 통해 capture_has_been_full의 값을 bus_clk로 복사합니다.

user_r_read_32_eof는 API에서 허용하는 대로 로우에서 하이로 이동합니다. 이는 3.3 섹션에서 제안한 대로 user_r_read_32_empty와 함께 logical AND가 있기 때문입니다.

4.6 테스트 실행

중요한:

이 테스트 실행은 의도적으로 IP core의 부적합한 구성의 나쁜 예를 보여줍니다. 이 고의적인 실수의 목적은 EOF가 어떻게 작동하는지 보여주기 위한 것입니다. 이 테스트에 사용된 IP core에는 작은 buffers와 synchronous stream이 있었습니다. 이는 data acquisition 애플리케이션에 대한 잘못된 선택입니다. 적절하게 구성된 IP core는 아래와 같이 제대로 작동하지 않습니다.

전송된 데이터의 반복성을 보장하기 위해 데이터 소스는 전송된 단어 수를 계산하는 단순 카운터로 선택됩니다. EOF까지의 데이터 양은 임의적입니다. EOF는 컴퓨터가 다른 일을 하느라 바빠서 device file에서 읽는 작업을 잠시 소홀히 했을 때 발생했습니다.

Linux에 대한 테스트 실행이 표시되지만 Windows에서도 실행할 수 있습니다. command line utilities 실행에 대한 자세한 내용은 다음 가이드 중 하나에서 찾을 수 있습니다.

- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

테스트 실행은 다음과 같습니다.

```
$ cat /dev/xillybus_read_32 > first
$ cat /dev/xillybus_read_32 > second
$ ls -l
total 77740
-rw-rw-r--. 1 liveuser liveuser 71727100 Jul 13 15:31 first
-rw-rw-r--. 1 liveuser liveuser  7874556 Jul 13 15:31 second
```

따라서 첫 번째 시도에서는 약 71 MB가 수집되었지만 두 번째 시도에서는 7 MB만 수집되었습니다. 각 실행의 데이터 양은 운영 체제가 다른 작업을 수행하기 위해 읽기 프로세스를 무시하기 전에 수신된 데이터 양에 따라 다릅니다. 아마도 디스크에 쓰기 위해 읽기 프로세스가 잠시 중지되었을 것입니다.

그러나 모든 데이터를 /dev/null로 전송하여 삭제하더라도 결국 중지됩니다(dd 유틸리티에 대한 자세한 내용은 “man dd” 시도).

```
$ dd if=/dev/xillybus_read_32 of=/dev/null bs=1M
0+34365 records in
0+34365 records out
140756988 bytes (141 MB) copied, 18.0364 s, 7.8 MB/s
$ dd if=/dev/xillybus_read_32 of=/dev/null bs=1M
0+6027 records in
0+6027 records out
24684540 bytes (25 MB) copied, 3.16028 s, 7.8 MB/s
```

이 두 테스트 모두에서 컴퓨터의 마우스를 움직이면 데이터 흐름이 중지되었습니다. 이것은 운영 체제를 충분히 산만하게 만들었습니다.

다시 한 번 강조하는 것이 중요합니다. 이것은 synchronous stream이 사용되기 때문에 정말 나쁜 결과입니다. asynchronous stream와 정확한 양의 DMA buffers를 사용하면 이런 종류의 문제는 전혀 예상되지 않습니다.

마지막으로 파일 중 하나에 무엇이 있는지 살펴보겠습니다.

```
$ hexdump -C -v first | head
00000000  f8 fb a2 01 f9 fb a2 01  fa fb a2 01 fb fb a2 01  |................|
00000010  fc fb a2 01 fd fb a2 01  fe fb a2 01 ff fb a2 01  |................|
00000020  00 fc a2 01 01 fc a2 01  02 fc a2 01 03 fc a2 01  |................|
00000030  04 fc a2 01 05 fc a2 01  06 fc a2 01 07 fc a2 01  |................|
00000040  08 fc a2 01 09 fc a2 01  0a fc a2 01 0b fc a2 01  |................|
00000050  0c fc a2 01 0d fc a2 01  0e fc a2 01 0f fc a2 01  |................|
00000060  10 fc a2 01 11 fc a2 01  12 fc a2 01 13 fc a2 01  |................|
00000070  14 fc a2 01 15 fc a2 01  16 fc a2 01 17 fc a2 01  |................|
00000080  18 fc a2 01 19 fc a2 01  1a fc a2 01 1b fc a2 01  |................|
00000090  1c fc a2 01 1d fc a2 01  1e fc a2 01 1f fc a2 01  |................|
```

예상대로 데이터에는 카운트업 시퀀스가 포함되어 있습니다. 데이터 생성에 사용되는 카운터는 재설정되지 않으므로 시퀀스가 0에서 시작하지 않습니다.

4.7 버퍼링된 데이터 양 모니터링

특정 stream에 속하는 Xillybus의 buffers에 얼마나 많은 데이터가 보관되어 있는지 추적하고 싶은 경우가 많습니다. 이는 latency 제어, overflow 또는 underflow 방지, 또는 read() 또는 write()에 대한 함수 호출 중에 application software가 절전 모드로 전환되는 것을 방지하는 데 도움이 될 수 있습니다.

예를 들어, FPGA에서 host로의 데이터 흐름과 관련하여: IP Core는 FPGA의 FIFO에서 이 데이터를 읽었지만 application software는 아직 이 데이터를 사용하지 않았기 때문에 buffers에 저장된 데이터의 양이 있을 수 있습니다. 이와 같이 얼마나 많은 데이터가 대기하고 있는지 아는 것이 종종 바람직합니다.

마찬가지로 반대 방향으로: application software가 stream에 쓴 데이터가 있을 수 있지만 FPGA의 FIFO에는 아직 도달하지 않았습니다. 직접적인 이유는 FPGA의 FIFO가 가득 차서 IP core에서 더 이상 데이터를 받아들일 수 없기 때문입니다. 그러나 실제 설명은 데이터가 application logic에서 소비되기를 기다리고 있다는 것입니다.

Xillybus는 buffers의 데이터 양을 추정하기 위한 전용 기능을 제공하지 않습니다. 그러나 다음과 같이 Xillybus의 기존 기능을 사용하여 이 기능을 구현하는 간단한 방법이 있습니다.

제안된 솔루션을 설명하기 위해 demo bundle의 streams(FPGA에서 host, 32비트) 중 하나가 data acquisition에 사용된다고 가정해 보겠습니다.

다음 카운터는 파일이 열린 이후 FIFO(IP core에 의해)에서 가져온 데이터 단어 수를 계산하는 데 사용됩니다.

```
reg [31:0] count_data;

always @(posedge bus_clk)
  if (!user_r_read_32_open)
    count_data <= 0;
  else if (user_r_read_32_rden)
    count_data <= count_data + 1;
```

count_data는 3.4 섹션에서 제안한 대로 registers 어레이의 register일 수 있습니다.

대체 솔루션은 다른 Xillybus stream(FPGA에서 host로)를 IP core에 추가하는 것입니다. 이 stream은 count_data를 이 추가 stream의 data port(즉, 일반적으로 FIFO의 데이터 출력에 연결되는 port)에 직접 연결하여 count_data의 값을 host로 보내는 데 사용됩니다.

이 stream의 'eof' port 및 'empty' port는 지속적으로 낮게 유지되어야 합니다. 이 stream은 IP Core Factory의 "use" 매개변수를 "Command and status"로 설정하여 synchronous stream로 구성해야 합니다. 결과적으로 application software는 count_data의 업데이트

된 값을 얻기 위해 언제든지 이 stream에서 4바이트를 읽을 수 있습니다.

count_data는 bus_clk와 동기식이므로 Xillybus IP core의 data port에 직접 연결할 수 있습니다.

buffers의 데이터 양은 count_data와 application software가 열린 이후 device file(예: 이 예에서는 /dev/xillybus_read_32)에서 읽은 데이터 양의 차이로 계산할 수 있습니다. 물론 소프트웨어는 이 stream에서 읽는 데이터의 양을 추적해야 합니다.

반대 방향(host에서 FPGA로)에서 유사한 카운터를 FPGA에서 다음과 같이 유지할 수 있습니다.

```
reg [31:0] count_data;

always @(posedge bus_clk)
  if (!user_w_write_32_open)
    count_data <= 0;
  else if (user_w_write_32_wren)
    count_data <= count_data + 1;
```

이것은 동일한 원리로 작동합니다. application software는 관련 device file에 쓰는 데이터의 양을 추적합니다. application software는 buffers에 얼마나 많은 데이터가 저장되어 있는지 알아야 할 때 count_data를 읽습니다. 이 데이터 양은 (device file이 열린 이후로) 쓰여진 데이터 양과 count_data 값의 차이로 계산됩니다.

지금까지의 논의에서 FIFOs의 데이터는 계산에 포함되지 않았습니다. Xillybus가 buffers에 보관하는 데이터만 고려했습니다. 때로는 FIFOs에 저장된 데이터를 포함하여 종단간 번호를 얻고 싶을 때가 있습니다. 이를 위해 FIFOs의 반대쪽 작업을 계산해야 합니다. 즉, FPGA에서 host까지 stream에 대해 FIFO에 기록되는 요소의 수입니다. 반대 방향으로 이것은 FIFO에서 읽은 요소의 수입니다.

그러나 FIFO의 다른 쪽이 다른 clock(예: 이전에 제시된 capture_clk)와 동기화된 경우 구현하기가 더 어려울 수 있습니다. 이는 count_data가 이 다른 clock와 동기화되어야 하기 때문입니다. 결과적으로 count_data의 값을 IP core에 연결하기 위해서는 clock domain crossing이 필요합니다. 따라서 두 개의 다른 clocks가 FIFO에 연결될 때 정확성과 단순성 사이에 균형이 있습니다.

5

simulation에 대한 권장 방법

5.1 일반적인

만족스러운 simulation은 취향과 작업 방식의 문제입니다. 그럼에도 불구하고 항상 simulation와 관련하여 가정이 이루어집니다. 이러한 가정에는 특정 기능 요소가 예상대로 작동한다는 기대가 포함됩니다. 따라서 simulation로 이러한 기능적 요소를 검토하는 것은 무의미합니다. 시뮬레이션에 도움이 되는 특정 기능 요소도 있을 수 있지만 그렇게 하는 것은 너무 복잡하거나 시간이 많이 걸립니다.

이 섹션에서는 simulation 프로세스에 대한 몇 가지 가정과 제한 사항을 제안합니다. Xillybus IP core와 관련된 시스템의 simulation에 대한 접근 방식도 논의됩니다. 이 가이드라인은 본질적으로 이 문서의 나머지 부분에 있는 가이드라인보다 논의의 여지가 더 많습니다.

Xillybus IP core 및 driver는 다양한 시나리오에서 광범위하게 테스트된 복잡한 시스템입니다. 따라서 simulation의 도움으로 IP core 자체에서 버그를 찾을 가능성은 거의 없습니다. 테라바이트의 데이터 전송과 광범위한 사용 패턴에서 버그가 발견되지 않은 경우 simulation은 그러한 버그를 드러내지 않을 것입니다.

또한 IP core의 동작은 host의 응답에 크게 좌우됩니다. driver와 application software는 거의 예측할 수 없는 서로 다른 방식과 서로 다른 delays로 응답합니다. 또한 bus의 latency(PCIe, AXI 또는 USB)도 마찬가지로 임의적이므로 예측할 수 없습니다. 따라서 포괄적인 simulation은 거의 불가능합니다.

이를 고려하여 FIFO와 Xillybus IP core가 연결되는 지점까지는 application logic의 simulation을 수행하는 것이 좋습니다. 따라서 IP core는 데이터 방향에 따라 이 FIFO를 비우거나 채우는 black box로 시뮬레이션됩니다.

5.2 asynchronous streams 시뮬레이션

stream이 asynchronous로 구성되면 IP core는 FIFO가 overflow 또는 underflow 상태에 도달하지 않도록 FIFO와 FIFO 간에 데이터를 전송합니다(stream의 방향에 따라 다름).

이는 host의 응용 프로그램 소프트웨어가 I/O 작업을 충분히 자주 수행하고 Xillybus의 대역폭 기능이 임무에 적합한 한 사실입니다. 이 두 가지 조건은 적절하게 설계된 프로젝트의 결과입니다. simulation 덕분에 design의 두 가지 측면을 검증하는 것이 도움이 될 수 있습니다.

- FIFO가 overflow 또는 underflow에 도달하는지 여부(방향에 따라 다름).
- application logic이 4.5 섹션에서 제안한 것처럼 이러한 결함 상황에 올바르게 응답하는지 여부.

적절한 작동을 시뮬레이션하기 위해 관련 'open' 신호가 높으면 IP core가 최대 속도로 FIFO로 또는 FIFO에서 데이터를 전송한다고 가정할 수 있습니다(host에서 파일을 열었음을 나타냄).

host에서 FPGA에 이르는 stream의 경우 FIFO가 underflow로 인해 어떤 일이 발생하는지 테스트하는 것이 좋습니다. FIFO가 비어 있는 것처럼 보이게 하여 이 이벤트를 시뮬레이션하는 것이 좋습니다. 예를 들어 FIFO가 test bench의 일부인 경우 test bench는 'empty' 신호(application logic에 연결됨)를 높음으로 변경합니다. 또는 host의 데이터 흐름을 시뮬레이트하는 test bench 부분이 일정 기간 동안 데이터를 FIFO로 푸시하기 위해 단순히 멈출 수 있습니다. 그 결과 FIFO가 비어 있게 됩니다.

마찬가지로 FPGA에서 host로 stream의 경우: 'full' 라인을 하이로 변경하여 FIFO의 overflow를 테스트할 수 있습니다. 또는 test bench가 일정 시간 동안 FIFO에서 데이터 가져오기를 중지하여 동일한 효과를 얻을 수 있습니다.

데이터 stream의 연속성을 깨는 한 가지 가능한 이유는 application logic이 stream의 대역폭 제한(또는 IP core의 총 대역폭 제한)을 초과하려고 하기 때문입니다. 이러한 가능성이 있는 경우 test bench가 대역폭 제한을 시뮬레이션하는 것도 권장됩니다. 이는 test bench(IP core로 작동)가 stream의 의도된 대역폭에 의해 제한되는 데이터 속도로 FIFO를 채우거나 비우도록 함으로써 수행할 수 있습니다.

그러나 application logic은 대역폭 제한을 초과할 수 없기 때문에 많은 응용 프로그램에서 이러한 종류의 simulation은 필요하지 않습니다.

5.3 synchronous streams 시뮬레이션

simulation의 경우 synchronous stream의 주요 차이점은 IP core의 데이터 흐름이 연속적이지 않다는 것입니다. synchronous stream에서 IP core는 host(read() 또는 write())에

서 보류 중인 함수 호출이 있는 경우에만 FIFO와 데이터를 전송합니다.

따라서 IP core의 동작은 I/O에 대한 application software의 요청에 더 의존합니다. 따라서 test bench에서 IP core를 시뮬레이션하는 부분은 application software의 액세스 패턴을 염두에 두고 작성해야 합니다.

synchronous stream에 대해 overflow 또는 underflow를 시뮬레이트하는 것은 관련이 없을 수 있습니다. stream의 목적이 대량의 데이터를 교환하는 것일 때 synchronous stream이 덜 선호되는 옵션이기 때문입니다. 그럼에도 불구하고 이러한 조건을 시뮬레이션하는 방법은 asynchronous streams와 동일합니다.

5.4 simulation을 위한 단순화된 방법

overflow 및 underflow에 대한 응답 테스트에 관심이 없다면 IP core의 simulation에 대한 더 간단한 방법이 있습니다. 예를 들어, host에서 FPGA 방향으로: FIFO는 read enable 신호가 높을 때 각 rising clock edge에 대한 파일에서 데이터 워드를 읽어 test bench에서 간단히 구현할 수 있습니다. FIFO에 대한 이 단순화된 보기는 host가 관련 device file에 데이터를 충분히 빨리 기록하여 FIFO가 비워지는 것을 방지한다는 가정에 의존합니다.

반대 방향으로 test bench는 write enable 신호가 높을 때 파일에 워드를 씁니다. 이전과 마찬가지로 host는 데이터를 충분히 빠르게 읽어 FIFO가 가득 차는 것을 항상 방지한다는 가정이 있습니다.

이 접근 방식은 데이터 흐름의 연속성이 중단될 수 있는 가능성을 간과하지 않습니다. 오히려 이 접근 방식은 중단된 데이터 흐름이 simulation의 범위를 벗어나는 결과일 가능성이 가장 높다는 것을 인식합니다. 너무 얕은 DMA buffers, application software의 낮은 응답성 또는 host의 전반적인 상태로 인한 CPU 결핍. 이러한 이벤트가 실제로 발생하면 application logic은 host가 이를 인식하도록 해야 합니다. 위에서 이미 제안한 것처럼 이 메커니즘은 시뮬레이션할 수 있습니다.

그러나 이 접근 방식은 application logic이 stream의 대역폭 제한을 초과하려고 시도할 가능성을 무시합니다. 이러한 시나리오가 현실적인 가능성이 있는 경우 simulation에 대한 이 단순화된 방법은 적합하지 않을 수 있습니다.