ZYBO-Z7-20과 PYNQ 3.0.1을 활용한 동적 기능 교환(DFX) 구현: 종합 튜토리얼

섹션 1: 기초 설정: PYNQ 3.0.1을 위한 ZYBO-Z7-20 준비

동적 기능 교환(Dynamic Function eXchange, DFX)과 같은 고급 기능을 구현하기에 앞서, 안정적이고 검증된 개발 환경을 구축하는 것이 무엇보다 중요합니다. ZYBO-Z7-20 보드는 공식적으로 PYNQ v3.0.1 릴리스를 지원하지 않으므로, 이 초기 단계는 전체 프로젝트의 성공을 좌우하는 핵심적인 과정입니다. 이 섹션에서는 커뮤니티에서 검증된 경로를 통해 ZYBO-Z7-20 보드에 PYNQ v3.0.1 환경을 성공적으로 구축하는 방법을 단계별로 안내합니다.

1.1. 시스템 요구사항 및 툴체인 설치

안정적인 PYNQ 이미지 빌드 및 하드웨어 설계를 위해서는 특정 버전의 소프트웨어와 툴체인을 사용하는 것이 필수적입니다. 버전 불일치는 예측 불가능한 빌드 오류나 런타임 문제를 야기할 수 있으므로, 아래 명시된 환경을 정확히 구성해야 합니다.

먼저, 호스트 개발 환경은 Ubuntu 20.04 LTS 운영체제를 기반으로 해야 합니다. 이는 PYNQ 소스 코드 기반의 이미지 빌드 시스템이 해당 환경에서 개발 및 테스트되었기 때문입니다.[1] 가상 머신(VM)을 사용하는 경우, 최소 150GB 이상의 디스크 공간을 할당하여 빌드 과정에서 발생할 수 있는 공간 부족 문제를 예방해야 합니다.[2]

다음으로, PYNQ v3.0.1과의 완벽한 호환성을 위해 AMD-Xilinx의 **Vivado** 및 **Vitis 2022.1** 버전을 설치해야 합니다.[3] PYNQ 프레임워크는 특정 버전의 Vivado 툴과 긴밀하게 연동되므로, 다른 버전을 사용할 경우 하드웨어 핸드오프 파일(.hwh)의 파싱 오류나 드라이버 비호환성 문제가 발생할 수 있습니다.

마지막으로, Vivado가 ZYBO-Z7-20 보드를 올바르게 인식하고 Zynq Processing System(PS)의 복잡한 설정을 자동화할 수 있도록 Digilent의 Vivado 보드 파일을 설치해야 합니다.[5] 설치 절차는 다음과 같습니다:

1. Digilent의 vivado-boards GitHub 저장소에서 최신 마스터 브랜치의 ZIP 아카이브를 다운로드하여 압축을 해제합니다.[5]
2. 압축 해제된 폴더 내의 new/board_files 디렉토리로 이동합니다.
3. Vivado 설치 경로(예: C:/Xilinx/Vivado/2022.1/data/boards/) 아래에 board_files 디렉토리가 없다면 생성합니다.
4. new/board_files 내의 모든 보드 파일 폴더를 Vivado의 board_files 디렉토리로 복사합니다.

이 과정을 완료하면 Vivado에서 새 프로젝트를 생성할 때 보드 선택 목록에서 ZYBO-Z7-20을 직접 선택할 수 있게 됩니다.

1.2. PYNQ v3.0.1 SD 카드 이미지 획득 및 준비

ZYBO-Z7-20 보드는 PYNQ 프로젝트의 공식 지원 대상이 아니므로, 공식 PYNQ 웹사이트에서는 해당 보드를 위한 SD 카드 이미지를 제공하지 않습니다.[6] 이 문제를 해결하는 가장 효율적인 방법은 커뮤니티에서 빌드하고 공유한 이미지를 사용하는 것입니다. 이는 복잡하고 시간이 많이 소요되는 소스 코드 기반의 전체 이미지 빌드 과정을 생략할 수 있게 해주는 매우 중요한 지름길입니다.[4]

본 튜토리얼에서는 PYNQ 지원 포럼에서 사용자 'gabrielgagne'가 제공하는 압축된 Zybo-Z7-20-3.0.1 이미지를 사용할 것을 강력히 권장합니다.[8] 이 이미지는 USB 주변기기 호환성 문제 등이 해결된 버전으로, DFX 데모를 진행하기에 가장 적합한 기반을 제공합니다.

이미지를 획득한 후, 최소 16GB 용량의 Class 10 microSD 카드에 이미지를 기록해야 합니다.[2] Balena Etcher나 Win32DiskImager와 같은 이미지 라이팅 도구를 사용하여 다운로드한

.img 파일을 microSD 카드에 기록합니다.[9] 이 과정에서 microSD 카드의 모든 기존 데이터는 삭제되므로 사전에 백업해야 합니다.

1.3. 초기 보드 구성 및 네트워크 연결 확인

SD 카드가 준비되면, 보드의 물리적 설정을 진행하고 정상적으로 부팅되는지 확인해야 합니다.

1. 점퍼 설정: ZYBO-Z7-20 보드의 부팅 모드 점퍼(JP5)를 SD 카드로부터 부팅하도록 설정합니다.[2]
2. 케이블 연결: PROG/UART 라고 표시된 micro-USB 포트를 PC와 연결하여 시리얼 터미널 통신 및 JTAG 프로그래밍이 가능하도록 합니다. 또한, 보드의 이더넷 포트를 유선 네트워크에 연결하여 Jupyter Notebook 환경에 원격으로 접속할 수 있도록 합니다.[11]

3. 전원 인가: 보드에 전원을 인가하고 부팅 과정을 관찰합니다.

PC에서 PuTTY나 TeraTerm과 같은 시리얼 터미널 프로그램을 실행하여 보드와 연결된 COM 포트를 엽니다. 전송 속도는 115200으로 설정합니다. 정상적으로 부팅이 진행되면, Linux 부팅 메시지가 터미널에 출력됩니다.[12] 부팅이 완료된 후, 로그인 프롬프트가 나타나면 PYNQ의 기본 사용자 이름과 비밀번호(일반적으로 xilinx)를 사용하여 로그인합니다. ifconfig 명령어를 실행하여 보드에 DHCP를 통해 할당된 IP 주소를 확인합니다.

마지막으로, PC의 웹 브라우저를 열고 주소창에 http://<board_ip>:9090을 입력하여 Jupyter Notebook 환경에 접속합니다. PYNQ의 로그인 화면이 나타나면, 성공적으로 PYNQ 환경이 구축된 것입니다.

이처럼 공식 지원이 없는 하드웨어에 최신 소프트웨어 프레임워크를 적용하는 과정은 커뮤니티의 기여에 크게 의존합니다. 공식 문서만으로는 해결할 수 없는 문제에 직면했을 때, 커뮤니티 포럼이나 GitHub 저장소에서 공유되는 정보와 사전 빌드된 결과물은 프로젝트를 성공으로 이끄는 결정적인 역할을 합니다. 본 튜토리얼이 커뮤니티에서 제공하는 검증된 이미지를 기반으로 하는 이유도 바로 여기에 있습니다. 이는 사용자가 DFX라는 핵심 주제에 집중하기 전에 환경 설정의 어려움으로 인해 좌절하는 것을 방지하기 위함입니다.

아래 표는 본 튜토리얼을 성공적으로 수행하기 위한 필수 구성 요소를 요약한 것입니다.

표 **1:** 소프트웨어 및 하드웨어 구성 매트릭스

구성 요소	버전	출처/참고
호스트 OS	Ubuntu 20.04.x LTS	PYNQ 빌드 시스템 요구사항 [2]
Xilinx 툴	Vivado & Vitis 2022.1	PYNQ v3.0.1 호환성 [4]
대상 보드	Digilent ZYBO-Z7-20	파트: xc7z020clg400-1 [11]
PYNQ 이미지	Zybo-Z7-20-3.0.1 (커뮤니티)	PYNQ 포럼 링크 [8]
MicroSD 카드	16GB 이상, Class 10	[2]

섹션 2: 하드웨어 아키텍처: Vivado에서 DFX 시스템 설계

이 섹션에서는 DFX 프로젝트의 핵심인 하드웨어 설계를 Vivado 환경에서 처음부터 구축하는 전 과정을 상세히 다룹니다. Verilog를 사용한 하드웨어 모듈 코딩, IP 패키징, 블록 디자인 생성, 그리고 DFX의 필수 요소인 플로어플래닝(Floorplanning)까지 단계별로 안내합니다.

2.1. 설계 개념: 정적 기반과 재구성 가능한 연산 유닛

본 튜토리얼에서 구현할 DFX 시스템의 아키텍처는 명확하고 교육적인 목적에 초점을 맞춥니다. 시스템은 크게 두 부분으로 나뉩니다:

- 정적 로직(**Static Logic**): Zynq Processing System(PS)과 AXI 인터커넥트를 포함하며, 시스템이 동작하는 동안 절대 변경되지 않는 영역입니다. 이 정적 로직은 전체 시스템의 안정적인 기반 역할을 합니다.
- 재구성 가능 파티션(**Reconfigurable Partition, RP**): 동적으로 하드웨어 로직이 교체될 수 있는 예약된 영역입니다. 이 RP는 정적 로직의 AXI 버스에 연결됩니다.

이 RP에는 두 가지 종류의 **재구성 가능 모듈(Reconfigurable Modules, RMs)**이 동적으로 로드될 것입니다: 하나는 32비트 덧셈기(Adder)이고, 다른 하나는 32비트 곱셈기(Multiplier)입니다. PS는 소프트웨어(Python)의 제어에 따라 이 두 모듈 중 하나를 선택하여 RP에 로드하고, 입력 데이터를 전달하며, 연산 결과를 다시 읽어오는 역할을 합니다. 이 간단하면서도 강력한 설계는 DFX의 핵심 원리를 명확하게 보여줍니다.[14]

2.2. 재구성 가능 모듈(RM)의 Verilog 구현 및 IP 패키징

재구성 가능 모듈(RM)을 설계할 때, PS와의 통신을 위한 표준 인터페이스를 정의하는 것이 중요합니다.

2.2.1. AXI4-Lite 인터페이스 표준

PS와 PL(Programmable Logic) 간의 제어 및 데이터 교환을 위해 AXI4-Lite 프로토콜을 사용합니다. AXI4-Lite는 레지스터 기반의 경량화된 인터페이스로, 간단한 제어 신호나 소량의

데이터를 주고받는 데 이상적입니다.[16] 복잡한 버스트(burst) 전송 기능이 없어 리소스 사용량이 적고 구현이 용이하여, 본 튜토리얼과 같이 간단한 연산 모듈을 제어하는 데 매우 적합합니다.

2.2.2. 권장 워크플로우: Vivado IP Packager

AXI 인터페이스 로직을 직접 코딩하는 것은 복잡하고 오류가 발생하기 쉽습니다. 대신, Vivado가 제공하는 Tools -> Create and Package New IP 마법사를 사용하는 것이 훨씬 효율적이고 안정적인 방법입니다.[18] 이 마법사는 기능적으로 완전한 AXI4-Lite 슬레이브 인터페이스를 포함하는 Verilog 템플릿을 자동으로 생성해주어, 개발자는 핵심적인 사용자 로직 구현에만 집중할 수 있습니다.[21]

2.2.3. RM Alpha (덧셈기) Verilog 소스 코드

IP Packager를 통해 adder_ip라는 이름의 AXI4-Lite 주변장치를 생성합니다. 마법사가 생성한 ..._S00_AXI.v 파일에 아래와 같은 사용자 로직을 추가합니다. 이 코드는 AXI 버스를 통해 두 개의 32비트 피연산자를 입력받아 덧셈을 수행하고, 그 결과를 출력 레지스터에 저장합니다.

Verilog

```
// This user logic is inserted into the Verilog template generated by the IP Packager.

// -- User logic start --
// User logic registers
reg [31:0] operand_a;
reg [31:0] operand_b;
reg [31:0] result;

// AXI address decoding and register read/write logic
// This logic is placed inside the always @( posedge S_AXI_ACLK ) block for writes
// and the logic for reads.

// Write logic
always @( posedge S_AXI_ACLK )
begin
```

```
  if ( S_AXI_ARESETN == 1'b0 )
  begin
    operand_a <= 32'h0;
    operand_b <= 32'h0;
  end
  else
  begin
    if (slv_reg_wren) // slv_reg_wren is a signal from the template indicating a valid write
    begin
      case ( axi_awaddr ) // Assuming 4 registers, using 2 bits of address
        2'h0: // Address offset 0x00 (mapped to 0x10 in block design)
          operand_a <= S_AXI_WDATA;
        2'h1: // Address offset 0x04 (mapped to 0x18 in block design)
          operand_b <= S_AXI_WDATA;
        default:
          begin
            operand_a <= operand_a;
            operand_b <= operand_b;
          end
      endcase
    end
  end
end

// Continuous calculation logic
always @(*)
begin
    result = operand_a + operand_b;
end

// Read logic
always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
  begin
    slv_reg_rden <= 1'b0;
    reg_data_out <= 32'h0;
  end
  else
  begin
    // On a read request, output the correct register data
    if (slv_reg_rden)
    begin
```

```
    case ( axi_araddr )
      2'h2: // Address offset 0x08 (mapped to 0x20 in block design)
        reg_data_out <= result;
      default:
        reg_data_out <= 32'h0;
    endcase
  end
 end
end
// -- User logic end --
```

2.2.4. RM Beta (곱셈기) Verilog 소스 코드

덧셈기와 동일한 절차로 multiplier_ip라는 이름의 IP를 생성하고, 아래와 같이 곱셈 로직을 추가합니다. 두 모듈의 AXI 레지스터 맵(주소 구조)을 동일하게 설계하여 소프트웨어에서 일관된 방식으로 접근할 수 있도록 하는 것이 중요합니다.

Verilog

```
// This user logic is inserted into the multiplier IP's Verilog template.
// The structure is identical to the adder, only the calculation logic changes.

// -- User logic start --
// User logic registers
reg [31:0] operand_a;
reg [31:0] operand_b;
reg [31:0] result;

// Write logic (identical to adder)
always @( posedge S_AXI_ACLK )
begin
 if ( S_AXI_ARESETN == 1'b0 )
 begin
  operand_a <= 32'h0;
  operand_b <= 32'h0;
 end
 else
```

```
  begin
    if (slv_reg_wren)
    begin
      case ( axi_awaddr )
        2'h0: operand_a <= S_AXI_WDATA;
        2'h1: operand_b <= S_AXI_WDATA;
        default: begin
            operand_a <= operand_a;
            operand_b <= operand_b;
          end
      endcase
    end
  end
end

// Continuous calculation logic
always @(*)
begin
    result = operand_a * operand_b;
end

// Read logic (identical to adder)
always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
  begin
    slv_reg_rden <= 1'b0;
    reg_data_out <= 32'h0;
  end
  else
  begin
    if (slv_reg_rden)
    begin
      case ( axi_araddr )
        2'h2: reg_data_out <= result;
        default: reg_data_out <= 32'h0;
      endcase
    end
  end
end
// -- User logic end --
```

2.2.5. RM 패키징

각 모듈의 Verilog 코드 작성이 완료되면, IP Packager 창의 Review and Package 단계에서 Re-Package IP를 클릭하여 두 개의 개별 IP 코어를 완성합니다. 이 IP들은 이제 Vivado IP 카탈로그에 추가되어 블록 디자인에서 재사용할 수 있습니다.[19]

2.3. DFX를 위한 Vivado 프로젝트 생성 단계별 절차

이제 패키징된 IP들을 사용하여 DFX 프로젝트를 구성합니다. 이 과정은 일반적인 Vivado 프로젝트 흐름과 다르며, 특정 순서를 반드시 지켜야 합니다.

1. 프로젝트 초기화 및 정적 블록 디자인: 앞서 설치한 ZYBO-Z7-20 보드 파일을 사용하여 새 RTL 프로젝트를 생성합니다. IP Integrator -> Create Block Design을 통해 새 블록 디자인을 만들고, ZYNQ7 Processing System IP를 추가합니다. 상단에 나타나는 Run Block Automation을 실행하여 ZYBO-Z7 보드에 맞는 PS의 기본 설정을 자동으로 적용합니다.[9]
2. 재구성 가능 계층 생성: PYNQ가 DFX 영역을 인식하기 위해서는 해당 로직이 블록 디자인 내에서 계층(Hierarchy) 구조로 묶여 있어야 합니다. 이는 PYNQ의 DFX API가 계층 이름을 기준으로 부분 재구성을 수행하기 때문입니다.[26] 블록 디자인 캔버스에서 마우스 오른쪽 버튼을 클릭하고 Create Hierarchy를 선택하여 reconfigurable_region이라는 이름의 계층을 생성합니다.
3. 기본 RM 인스턴스화: 생성된 reconfigurable_region 계층 내부로 들어가서 IP 카탈로그로부터 adder_ip를 추가합니다. 이 덧셈기 IP의 AXI 인터페이스를 Zynq PS의 M_AXI_GP0 포트에 연결하고, Run Connection Automation을 실행하여 클럭 및 리셋 신호를 자동으로 연결합니다. 이 덧셈기는 전체 비트스트림에 포함될 기본 모듈 역할을 합니다.
4. DFX 프로젝트로 변환: Vivado 메뉴에서 Tools -> Enable Dynamic Function eXchange를 선택하고 Convert를 클릭합니다. 이 작업은 되돌릴 수 없다는 경고를 확인합니다. 이 단계를 거쳐야만 Vivado가 DFX 관련 기능들을 활성화합니다.[27]
5. 재구성 가능 파티션 정의: Sources 패널에서 블록 디자인 파일(.bd)을 확장하여 reconfigurable_region 계층을 찾습니다. 이 계층을 마우스 오른쪽 버튼으로 클릭하고 Create Partition Definition을 선택합니다. 이 과정은 논리적 계층을 물리적으로 재구성 가능한 파티션(RP)으로 지정하는 역할을 합니다.[27]
6. DFX 마법사 사용: Tools -> Dynamic Function eXchange Wizard를 실행하여 재구성 설정을 구성합니다.
   - 1단계: Edit Reconfigurable Modules 페이지에서 + 버튼을 눌러 multiplier_ip를 기존 파티션 정의에 추가합니다. 이제 이 파티션은 덧셈기 또는 곱셈기를 가질 수 있습니다.
   - 2단계: Edit Configurations 페이지에서 두 개의 'Configuration'을 생성합니다. 첫 번째는 config_adder로, reconfigurable_region 파티션에 adder_ip RM을 할당합니다.

두 번째는 config_multiplier로, 동일한 파티션에 multiplier_ip RM을 할당합니다.
    - **3**단계: 마법사를 완료하면, Vivado는 Design Runs 탭에 config_adder와 config_multiplier 각각에 대한 합성(synthesis) 및 구현(implementation) 실행(run)을 자동으로 생성합니다.[27]

2.4. 핵심 단계**: Pblock**을 이용한 플로어플래닝

DFX 설계에서 플로어플래닝은 선택이 아닌 필수 과정입니다. 구현 단계는 재구성 가능 파티션(RP)에 대한 물리적인 경계 정보 없이는 성공적으로 완료될 수 없습니다.

2.4.1. 플로어플래닝의 필요성

Pblock은 FPGA의 물리적 리소스(CLB, DSP, BRAM 등)의 특정 영역을 예약하는 제약 조건(constraint)입니다. DFX에서는 Pblock을 사용하여 RP를 위한 전용 공간을 할당합니다. 이렇게 함으로써, 정적 로직의 배치 및 배선(placement and routing)이 RP의 경계 내로 침범하지 않도록 보장합니다. 또한, RP와 정적 로직 간의 모든 신호 연결은 '파티션 핀'이라는 고정된 라우팅 지점을 통해 이루어집니다. 이 고정된 경계와 연결 지점 덕분에, 새로운 부분 비트스트림이 로드되어 RP 내부 로직이 완전히 변경되더라도 정적 로직은 아무런 영향을 받지 않고 안정적으로 동작할 수 있습니다.[14]

2.4.2. Pblock 생성 방법

1. Design Runs 탭에서 config_adder 구성의 합성(synthesis)을 실행합니다.
2. 합성이 완료되면 Open Synthesized Design을 클릭합니다.
3. Netlist 탭에서 reconfigurable_region 인스턴스를 찾습니다.
4. 해당 인스턴스를 마우스 오른쪽 버튼으로 클릭하고 Floorplanning -> Draw Pblock을 선택합니다.
5. Device 뷰에서 Zynq-7020 칩의 로직 영역 중 적절한 위치에 사각형 모양으로 Pblock을 그립니다. 너무 작게 그리면 리소스 부족으로 구현에 실패할 수 있으므로, Utilization 보고서를 참고하여 충분한 크기로 지정해야 합니다.

2.4.3. Pblock 속성 설정

7-시리즈 FPGA(Zynq-7000 포함)에서 DFX를 사용할 때 매우 중요한 속성이 있습니다. 생성된 Pblock을 선택하고 Pblock Properties 창에서 RESET_AFTER_RECONFIG 속성을 true로 설정합니다. 이 옵션은 부분 비트스트림이 로드된 후 RP 내부의 모든 플립플롭을 초기 상태로 리셋하는 역할을 하여, 재구성된 모듈이 예측 가능한 상태에서 동작을 시작하도록 보장합니다.[27] 이 설정은 XDC 제약 조건 파일에 다음과 같이 저장됩니다:

```
set_property RESET_AFTER_RECONFIG true [get_pblocks
pblock_reconfigurable_region_inst_0]
```

2.5. 비트스트림 생성 및 하드웨어 내보내기

모든 설정이 완료되면, Generate Bitstream을 실행합니다. Vivado는 config_adder와 config_multiplier 두 구성에 대해 각각 구현 및 비트스트림 생성을 수행합니다. 이 과정이 완료되면, 프로젝트 디렉토리에는 각 구성을 위한 전체 비트스트림과 각 RM을 위한 부분 비트스트림이 생성됩니다.[29]

PYNQ에서 DFX를 사용하기 위해서는 비트스트림 파일뿐만 아니라, 각 하드웨어 구성의 메타데이터를 담고 있는 .hwh 파일이 반드시 필요합니다. 이 파일은 PYNQ의 Overlay 클래스가 하드웨어의 주소 맵, IP 종류, 인터럽트 등의 정보를 파악하는 데 사용됩니다.[30] 따라서, 각 구성에 대해 하드웨어를 별도로 내보내야 합니다.

1. Design Runs 탭에서 config_adder의 구현 결과를 Open Implemented Design으로 엽니다.
2. File -> Export -> Export Hardware를 실행하여 .xsa 파일을 생성합니다.
3. config_multiplier의 구현 결과에 대해서도 동일한 과정을 반복하여 또 다른 .xsa 파일을 생성합니다.

.xsa 파일은 실제로는 ZIP 아카이브이므로, 확장자를 .zip으로 변경하여 압축을 해제하면 내부에 있는 .hwh 파일을 추출할 수 있습니다.[32]

DFX 빌드 과정은 복잡하며 다양한 출력 파일을 생성합니다. 아래 표는 PYNQ 오버레이에 필요한 핵심 파일들을 정리하고, PYNQ 파일 시스템에 배포하기 위한 권장 파일명을 제시합니다. 이 표는 사용자가 올바른 파일을 식별하고 준비하는 데 도움이 될 것입니다.

표 **2: DFX** 출력 파일 매니페스트

Vivado 출력 파일 (경로 단순화)	PYNQ용 파일명 변경	PYNQ에서의 역할

.../impl_1/top.bit	dfx_demo.bit	덧셈기 RM을 포함한 전체 비트스트림 (초기 로드용)
.../impl_1/top.hwh	dfx_demo.hwh	정적 디자인 및 덧셈기 인터페이스 메타데이터
.../impl_1/reconfigurable_region_adder_ip_0_partial.bit	adder.bit	덧셈기 RM을 위한 부분 비트스트림
.../impl_1/reconfigurable_region_adder_ip_0_partial.hwh	adder.hwh	덧셈기 RM 내부 구조에 대한 메타데이터
.../impl_2/reconfigurable_region_multiplier_ip_0_partial.bit	multiplier.bit	곱셈기 RM을 위한 부분 비트스트림
.../impl_2/reconfigurable_region_multiplier_ip_0_partial.hwh	multiplier.hwh	곱셈기 RM 내부 구조에 대한 메타데이터

섹션 **3:** 오버레이 배포**:** 하드웨어를 **PYNQ** 파일 시스템으로 전송

Vivado에서 하드웨어 설계 및 비트스트림 생성을 완료한 후, 생성된 결과물들을 ZYBO 보드의 PYNQ 환경으로 옮겨야 소프트웨어에서 이를 제어할 수 있습니다. 이 섹션에서는 이 과정을 정확하게 수행하는 방법을 안내합니다.

3.1. DFX 결과물 정리

먼저, 호스트 PC에서 작업의 편의를 위해 하나의 폴더에 필요한 모든 파일을 모으는 것이 좋습니다. 앞서 "DFX 출력 파일 매니페스트" 표에서 정의한 대로, Vivado 프로젝트의 각기 다른 출력 디렉토리에 흩어져 있는 파일들을 복사해오고, PYNQ에서 사용하기 쉽도록 파일명을 변경합니다. 이 정리 단계는 파일 누락이나 잘못된 파일 사용으로 인한 오류를 방지하는 데 매우

중요합니다.

정리가 완료된 폴더에는 다음과 같이 총 6개의 파일이 있어야 합니다:

- dfx_demo.bit
- dfx_demo.hwh
- adder.bit
- adder.hwh
- multiplier.bit
- multiplier.hwh

3.2. ZYBO 보드에 오버레이 디렉토리 생성

다음으로, ZYBO 보드에 이 파일들을 저장할 전용 디렉토리를 생성해야 합니다. PYNQ 프레임워크는 /home/xilinx/pynq/overlays/ 디렉토리 아래에 있는 오버레이들을 관리합니다. SSH 클라이언트(예: PuTTY)나 Jupyter Notebook의 터미널을 사용하여 보드에 접속한 후, 다음 명령어를 실행하여 새로운 오버레이 디렉토리를 생성합니다.

Bash

```
sudo mkdir -p /home/xilinx/pynq/overlays/dfx_demo
```

이 명령어는 dfx_demo라는 이름의 오버레이 폴더를 생성합니다. sudo를 사용하는 이유는 시스템 디렉토리에 대한 쓰기 권한이 필요하기 때문입니다.

3.3. 설계 파일 전송

이제 호스트 PC에 정리해 둔 6개의 파일을 ZYBO 보드의 새로 생성된 디렉토리로 전송합니다. scp (Secure Copy) 명령어를 사용하거나 FileZilla와 같은 그래픽 SFTP 클라이언트를 사용하면 편리합니다. scp를 사용하는 경우, 호스트 PC의 터미널에서 다음 명령어를 실행합니다. (<board_ip>는 보드의 IP 주소로 대체해야 합니다.)

Bash

```
scp /path/to/local/folder/* xilinx@<board_ip>:/home/xilinx/pynq/overlays/dfx_demo/
```

전송이 완료되면, 보드의 터미널에서 ls -l /home/xilinx/pynq/overlays/dfx_demo 명령어를 실행하여 파일들이 올바르게 전송되었는지 확인합니다. 최종적으로 보드 상의 디렉토리 구조는 다음과 같아야 합니다 [34]:

```
/home/xilinx/pynq/overlays/dfx_demo/
├── dfx_demo.bit
├── dfx_demo.hwh
├── adder.bit
├── adder.hwh
├── multiplier.bit
└── multiplier.hwh
```

이처럼 정확한 파일명과 디렉토리 구조를 갖추는 것은 PYNQ의 Overlay 클래스가 전체 비트스트림과 부분 비트스트림, 그리고 각각에 해당하는 메타데이터 파일을 올바르게 찾아 로드하는 데 필수적입니다.

섹션 **4:** 소프트웨어 인터페이스**:** 동적 제어를 위한 **Jupyter Notebook**

이 섹션은 프로젝트의 정점으로, 복잡한 하드웨어 DFX 기능을 간단한 Python 코드를 통해 제어하는 과정을 보여줍니다. Jupyter Notebook 환경에서 PYNQ 라이브러리를 사용하여 동적으로 하드웨어를 재구성하고, 변경된 하드웨어와 상호작용하는 방법을 단계별로 시연합니다.

4.1. PYNQ 부분 재구성 API

PYNQ는 부분 재구성(PR) 또는 DFX를 위한 고수준의 Python API를 제공하여 사용자가 저수준의 하드웨어 제어 메커니즘을 몰라도 쉽게 DFX를 활용할 수 있도록 합니다.

Overlay 클래스를 인스턴스화하면, PYNQ는 지정된 .bit 파일과 이름이 같은 .hwh 파일을 자동으로 찾아 파싱합니다. 이 초기 .hwh 파일(dfx_demo.hwh)의 메타데이터를 기반으로, PYNQ는 정적 디자인 영역과 기본으로 로드된 재구성 가능 모듈(RM)에 대한 정보를 포함하는 IP 딕셔너리를 생성합니다.[30]

DFX를 수행하는 핵심 함수는 Overlay.pr_download(partial_region, partial_bitstream) 입니다.[30] 이 함수는 두 개의 인자를 받습니다:

1. partial_region: 재구성할 대상인 재구성 가능 파티션(RP)의 이름입니다. 이 이름은 Vivado 블록 디자인에서 생성한 계층(Hierarchy)의 이름과 일치해야 합니다 (본 튜토리얼에서는 reconfigurable_region).
2. partial_bitstream: RP에 로드할 부분 비트스트림 파일의 이름입니다 (예: multiplier.bit).

pr_download 함수가 호출되면 내부적으로 두 가지 중요한 작업이 순차적으로 일어납니다. 첫째, PYNQ는 Linux 커널의 FPGA Manager를 통해 지정된 부분 비트스트림을 Zynq의 PCAP(Processor Configuration Access Port)으로 전송하여 PL의 해당 영역을 재구성합니다. 둘째, 재구성이 완료된 후, PYNQ는 로드된 부분 비트스트림과 이름이 같은 새로운 .hwh 파일(예: multiplier.hwh)을 찾아 파싱합니다. 그리고 이 새로운 메타데이터를 사용하여 Overlay 객체의 내부 IP 딕셔너리를 동적으로 업데이트합니다. 이 과정을 통해 Python 객체는 물리적인 하드웨어의 변경 사항을 즉시 반영하게 되며, 사용자는 새로운 하드웨어 모듈에 즉시 접근할 수 있습니다. 이 동적 메타데이터 파싱 및 객체 모델 업데이트 기능이야말로 PYNQ DFX 지원의 핵심적인 강력함입니다.[26]

4.2. 완전한 주석 포함 Jupyter Notebook 코드

아래는 DFX 데모를 위한 전체 Jupyter Notebook 코드입니다. 각 셀은 실행 가능한 코드 블록이며, Markdown 설명을 통해 각 단계의 목적과 동작 원리를 상세히 설명합니다.

셀 **1:** 라이브러리 임포트

이 셀에서는 PYNQ의 핵심 클래스인 Overlay와 시간 지연을 위한 time 라이브러리를 임포트합니다.

Python

```
from pynq import Overlay
import time
```

셀 **2:** 기본 오버레이 로드

Overlay 클래스를 사용하여 DFX 오버레이를 로드합니다. 이 단계에서 dfx_demo.bit 전체 비트스트림이 PL에 프로그래밍됩니다. 이 비트스트림에는 정적 영역과 기본 RM인 덧셈기가 포함되어 있습니다. PYNQ는 dfx_demo.hwh 파일을 함께 로드하여 하드웨어 구조를 파악합니다.

Python

```
# 오버레이를 로드합니다. 이 과정에서 PL은 정적 영역과
# 기본 덧셈기 RM을 포함하는 전체 비트스트림으로 프로그래밍됩니다.
dfx_overlay = Overlay('/home/xilinx/pynq/overlays/dfx_demo/dfx_demo.bit')
```

셀 **3:** 초기 모듈 **(**덧셈기**)**과 상호작용

오버레이가 로드되면, Vivado에서 정의한 reconfigurable_region 계층이 dfx_overlay 객체의 속성으로 노출됩니다. 이 속성을 통해 계층 내의 IP(여기서는 adder_ip_0)에 접근할 수 있습니다. 덧셈 연산을 수행하여 하드웨어가 올바르게 동작하는지 확인합니다.

Python

```
# 재구성 가능 영역은 계층 이름을 기반으로 속성으로 노출됩니다.
# AXI 레지스터에 접근하기 위해 IP 객체를 가져옵니다.
adder_ip = dfx_overlay.reconfigurable_region.adder_ip_0

# AXI 레지스터에 피연산자 값을 씁니다.
# 주소 오프셋은 Vivado IP Packager에서 정의된 값입니다.
operand_a = 100
operand_b = 50
adder_ip.write(0x10, operand_a) # 피연산자 A를 위한 주소
adder_ip.write(0x18, operand_b) # 피연산자 B를 위한 주소
```

```
# 결과 레지스터에서 값을 읽습니다.
result = adder_ip.read(0x20) # 결과를 위한 주소
print(f"덧셈기 결과: {operand_a} + {operand_b} = {result}")
assert result == 150
print("덧셈기 검증 완료.")
```

셀 **4:** 부분 재구성 실행

이제 DFX의 핵심 단계를 실행합니다. pr_download 함수를 호출하여 reconfigurable_region에 있는 덧셈기를 곱셈기로 교체합니다. PYNQ는 multiplier.bit 부분 비트스트림을 PL에 로드합니다.

Python

```
# 이것이 핵심적인 DFX 단계입니다.
# PYNQ에게 'reconfigurable_region'에 곱셈기의 부분 비트스트림을
# 로드하도록 지시합니다.
print("부분 재구성을 수행합니다: 덧셈기를 곱셈기로 교체 중...")
dfx_overlay.pr_download('reconfigurable_region', 'multiplier.bit')
print("재구성 완료.")
```

셀 **5:** 새로운 모듈 **(**곱셈기**)** 검증 및 상호작용

pr_download가 완료되면 dfx_overlay 객체는 내부적으로 업데이트됩니다. PYNQ는 multiplier.hwh 파일을 파싱하여 IP 딕셔너리를 갱신했기 때문에, 이제 adder_ip_0 속성은 사라지고 multiplier_ip_0 속성을 통해 새로운 곱셈기 IP에 접근할 수 있습니다. 곱셈 연산을 수행하여 DFX가 성공적으로 이루어졌음을 확인합니다.

Python

```
# pr_download 이후, 오버레이 객체는 업데이트되었습니다.
# 'adder_ip_0' 속성은 더 이상 존재하지 않으며, 'multiplier_ip_0'을 사용할 수 있습니다.
# 속성을 다시 조회하여 새로운 IP 객체를 가져옵니다.
multiplier_ip = dfx_overlay.reconfigurable_region.multiplier_ip_0
```

```
# AXI 레지스터에 피연산자 값을 씁니다.
operand_a = 12
operand_b = 10
multiplier_ip.write(0x10, operand_a)
multiplier_ip.write(0x18, operand_b)

# 결과 레지스터에서 값을 읽습니다.
result = multiplier_ip.read(0x20)
print(f"곱셈기 결과: {operand_a} * {operand_b} = {result}")
assert result == 120
print("곱셈기 검증 완료.")
```

이 Jupyter Notebook은 DFX의 전 과정을 명확하게 보여줍니다. 초기 하드웨어 상태를 확인하고, 단 한 줄의 Python 코드로 하드웨어의 일부를 동적으로 변경한 후, 즉시 변경된 하드웨어와 상호작용하는 일련의 과정을 통해 PYNQ가 제공하는 높은 수준의 추상화와 생산성을 체감할 수 있습니다.

섹션 **5:** 결론 및 고급 개념

본 튜토리얼은 Digilent ZYBO-Z7-20 보드와 PYNQ v3.0.1 프레임워크를 사용하여 동적 기능 교환(DFX)을 구현하는 전체 과정을 상세하게 다루었습니다. 이 마지막 섹션에서는 달성한 성과를 요약하고, 실제 시스템에 DFX를 적용할 때 고려해야 할 고급 주제와 추가적인 탐구 방향을 제시합니다.

5.1. 성과 요약

이 튜토리얼을 통해 사용자는 다음과 같은 일련의 복잡한 하드웨어/소프트웨어 통합 작업을 성공적으로 수행했습니다:

1. 환경 구축: 공식 지원이 없는 ZYBO-Z7-20 보드에 커뮤니티 기반의 PYNQ v3.0.1 환경을 성공적으로 구축하여 프로젝트의 기반을 마련했습니다.
2. **DFX** 하드웨어 설계: Vivado를 사용하여 정적 로직과 재구성 가능 파티션(RP)으로 구성된 DFX 시스템을 설계했습니다. Verilog로 덧셈기와 곱셈기라는 두 개의 독립적인 재구성 가능 모듈(RM)을 구현하고 IP로 패키징했습니다.
3. 물리적 제약 조건 설정: DFX의 핵심 요구사항인 플로어플래닝을 수행하여 Pblock으로 RP의 물리적 경계를 정의하고, 안정적인 재구성을 위한 RESET_AFTER_RECONFIG 속성을

설정했습니다.

4. 결과물 생성 및 배포: DFX 빌드 흐름을 통해 전체 및 부분 비트스트림과 각각에 해당하는 .hwh 메타데이터 파일을 모두 생성하고, 이를 PYNQ 오버레이로 구성하여 보드에 배포했습니다.
5. 소프트웨어 기반 동적 제어: Jupyter Notebook에서 PYNQ의 고수준 API를 사용하여, 단 몇 줄의 Python 코드로 실행 중에 하드웨어의 일부 기능을 동적으로 교체하고, 변경된 하드웨어의 동작을 즉시 검증하는 데 성공했습니다.

이 과정을 통해 사용자는 DFX의 개념적 이해를 넘어, 실제 하드웨어에서 이를 구현하고 소프트웨어와 통합하는 실질적인 경험을 얻었습니다.

5.2. 추가 탐구를 위한 권장 사항

본 튜토리얼에서 다룬 내용은 DFX의 기본적인 구현에 초점을 맞추었습니다. 실제 상용 시스템이나 더 복잡한 애플리케이션에 DFX를 적용하기 위해서는 다음과 같은 고급 개념들을 고려해야 합니다.

- **AXI** 버스 안정성 확보: 실제 시스템에서는 재구성 가능 모듈이 AXI 버스를 통해 메모리나 다른 주변장치와 지속적으로 통신할 수 있습니다. 만약 AXI 트랜잭션이 진행 중인 상태에서 부분 재구성이 발생하면, 버스가 응답 없는 상태(hang)에 빠져 전체 시스템이 멈출 수 있습니다. 이러한 문제를 방지하기 위해 AMD-Xilinx는 AXI Shutdown Manager 또는 DFX Decoupler와 같은 IP를 제공합니다.[37] 이 IP들은 재구성을 시작하기 전에 해당 AXI 인터페이스를 안전하게 정지(quiesce)시키고, 재구성이 완료된 후 다시 활성화하여 시스템 버스의 안정성을 보장하는 중요한 역할을 합니다.
- 다중 영역 **DFX (Multi-Region DFX)**: 본 튜토리얼은 단일 재구성 파티션(RP)을 사용했지만, 하나의 FPGA 설계 안에 여러 개의 독립적인 RP를 둘 수도 있습니다. 이를 통해 시스템의 각기 다른 부분을 서로 독립적으로, 동시에 재구성할 수 있어 시스템의 유연성을 극대화할 수 있습니다. 예를 들어, 한쪽 RP에서는 비디오 인코딩 코덱을 교체하는 동안 다른 쪽 RP에서는 통신 프로토콜 가속기를 변경하는 시나리오가 가능합니다.
- 추상 셸 컴파일 **(Abstract Shell Compilation)**: 새로운 재구성 가능 모듈(RM)을 개발할 때마다 전체 정적 디자인과 함께 매번 구현(implementation)을 다시 실행하는 것은 시간이 많이 소요됩니다. '추상 셸'은 이러한 문제를 해결하기 위한 고급 DFX 흐름입니다.[39] 이 방식에서는 먼저 정적 디자인 부분만 완전히 구현하여 '셸'을 만듭니다. 이 셸에는 RP의 경계와 파티션 핀에 대한 모든 물리적 정보가 포함되어 있습니다. 새로운 RM을 개발할 때는 전체 디자인을 다시 구현할 필요 없이, 이 사전 컴파일된 추상 셸에 맞춰 RM만 독립적으로 합성하고 구현하면 됩니다. 이는 새로운 기능 모듈의 개발 및 검증 시간을 획기적으로 단축시켜 줍니다.

이러한 고급 기법들을 학습하고 적용함으로써, 사용자는 DFX 기술을 더욱 강력하고 안정적으로 활용하여 리소스 효율적이고 유연하며 미래 지향적인 FPGA 기반 시스템을 구축할 수 있을

것입니다.

참고 자료

1. nick-petrovsky/PYNQ-ZYBO - GitHub, 9월 25, 2025에 액세스, https://github.com/nick-petrovsky/PYNQ-ZYBO
2. Build a custom PYNQ image for Zybo-Z7-20 - GitHub, 9월 25, 2025에 액세스, https://github.com/gabrielpgagne/pynq-zybo-z720
3. Z7_Color_AID-Blog#01: First Impression of PYNQ on Arty-Z7-20 - element14 Community, 9월 25, 2025에 액세스, https://community.element14.com/challenges-projects/design-challenges/eye-on-intelligence-challenge/b/blog/posts/eye-on-intelligence-challenge---z7_5f00_color_5f00_aid-blog-01-first-impression-of-pynq-on-arty-z7-20
4. PYNQ SD Card image — Python productivity for Zynq (Pynq) - Read the Docs, 9월 25, 2025에 액세스, https://pynq.readthedocs.io/en/latest/pynq_sd_card.html
5. Install Digilent's Board Files - Digilent Reference, 9월 25, 2025에 액세스, https://digilent.com/reference/programmable-logic/guides/install-board-files
6. PYNQ supported boards and PYNQ pre-built images, 9월 25, 2025에 액세스, http://www.pynq.io/boards.html
7. Z7_Color_AID Blog #2: Reproduce PYNQ Image on Ubuntu 20.04 - element14 Community, 9월 25, 2025에 액세스, https://community.element14.com/challenges-projects/design-challenges/eye-on-intelligence-challenge/b/blog/posts/the-eye-on-intelligence-challenge-blog-2-reproduce-pynq-image-on-ubuntu-20-04
8. PYNQ 3.0.1 for Zybo Z7-20 - Learn - PYNQ, 9월 25, 2025에 액세스, https://discuss.pynq.io/t/pynq-3-0-1-for-zybo-z7-20/6578
9. PYNQ Tutorial 1: GPIO - HackMD, 9월 25, 2025에 액세스, https://hackmd.io/@ween168/SyNPIkUb1x
10. Pynq 2.7 for Zybo-Z7 - Learn, 9월 25, 2025에 액세스, https://discuss.pynq.io/t/pynq-2-7-for-zybo-z7/4124
11. Zybo Z7 Reference Manual - Digilent, 9월 25, 2025에 액세스, https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual
12. Zybo Z7 HDMI Input/Output Demo - Digilent Reference, 9월 25, 2025에 액세스, https://digilent.com/reference/programmable-logic/zybo-z7/demos/hdmi
13. Zybo Z7 - Digilent Reference, 9월 25, 2025에 액세스, https://digilent.com/reference/programmable-logic/zybo-z7/start
14. Tutorial on Partial Reconfiguration of Image Processing Blocks using Vivado and SDK - ivPCL, 9월 25, 2025에 액세스, http://ivpcl.unm.edu/ivpclpages/Research/drastic/PRWebPage/PRLabPresentation.pdf
15. XAPP1231, Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite - Xilinx Wiki - Confluence, 9월 25, 2025에 액세스, https://xilinx-wiki.atlassian.net/wiki/display/A/XAPP1231+-+Partial+Reconfiguration+of+a+Hardware+Accelerator+with+Vivado+Design+Suite
16. Testing Verilog AXI4-Lite Peripherals - David Nadlinger, 9월 25, 2025에 액세스,

https://klickverbot.at/blog/2016/01/testing-verilog-axi4-lite-peripherals/
17. Buidilng an AXI-Lite slave the easy way - ZipCPU, 9월 25, 2025에 액세스, https://zipcpu.com/blog/2020/03/08/easyaxil.html
18. k0nze/zedboard_axi4_master_burst_example: Quick Example how to generate an custom AXI4 IP with AXI4-Full interface (burst) for the Zynq (ZedBoard) - GitHub, 9월 25, 2025에 액세스, https://github.com/k0nze/zedboard_axi4_master_burst_example
19. Creating a Custom AXI4-Lite IP - ArkX, 9월 25, 2025에 액세스, https://arkx.dev/design/fpga/creating-a-custom-axi4-lite-ip/
20. AXI4-Lite Interface Wrapper for Custom RTL in Vivado 2021.2 - Hackster.io, 9월 25, 2025에 액세스, https://www.hackster.io/whitney-knitter/axi4-lite-interface-wrapper-for-custom-rtl-in-vivado-2021-2-8a7009
21. Best way to create a AXI4-lite slave interface : r/FPGA - Reddit, 9월 25, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/ag6edi/best_way_to_create_a_axi4lite_slave_interface/
22. Building Custom AXI IP, 9월 25, 2025에 액세스, https://www.xilinx.com/content/dam/xilinx/training/embedded/embd/Building-Custom-AXI-IP.pdf
23. Calan Digital - Custom Axi lite slave - Google Sites, 9월 25, 2025에 액세스, https://sites.google.com/site/calandigital/tutorials/red-pitaya/custom-axi-lite-slave
24. Path to Programmable III Training Blog #04: My first Custom IP in Vivado, 9월 25, 2025에 액세스, https://community.element14.com/challenges-projects/design-challenges/pathprogrammable3/b/blog/posts/path-to-programmable-iii-training-blog-04-my-first-custom-ip-in-vivado
25. Tutorial: Creating a hardware design for PYNQ - Learn, 9월 25, 2025에 액세스, https://discuss.pynq.io/t/tutorial-creating-a-hardware-design-for-pynq/145
26. PYNQ, Partial Reconfiguration, Part 2 - DJ Park, 9월 25, 2025에 액세스, https://dj-park.github.io/posts/2022/1/PYNQ_PR_2/
27. How-To on Partial Reconfiguration with Vivado - 01signal.com, 9월 25, 2025에 액세스, https://www.01signal.com/vendor-specific/xilinx/partial-reconfiguration/part2-vivado-flow/
28. Partition Pins - 2025.1 English - UG909, 9월 25, 2025에 액세스, https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration/Partition-Pins
29. Partial Reconfiguration with Xilinx VIVADO tool-An Example Design - YouTube, 9월 25, 2025에 액세스, https://www.youtube.com/watch?v=TfAaa8Eaqw4
30. Partial Reconfiguration — Python productivity for Zynq (Pynq), 9월 25, 2025에 액세스, https://pynq.readthedocs.io/en/latest/overlay_design_methodology/partial_reconfiguration.html
31. Partial Reconfiguration — Python productivity for Zynq (Pynq) - Read the Docs,

9월 25, 2025에 액세스, https://pynq.readthedocs.io/en/v2.7.0/overlay_design_methodology/partial_reconfiguration.html
32. .bit file to .hwh file - Adaptive Support, 9월 25, 2025에 액세스, https://adaptivesupport.amd.com/s/question/0D52E00006hpSAISA2/bit-file-to-hwh-file?language=en_US
33. Registers Addressing for .HWH file - Support - PYNQ, 9월 25, 2025에 액세스, https://discuss.pynq.io/t/registers-addressing-for-hwh-file/1667
34. byuccl/PYNQ-PRIO - GitHub, 9월 25, 2025에 액세스, https://github.com/byuccl/PYNQ-PRIO
35. Blog #5: CNN HW Accelerator for Handwriting Recognition - Integrating the HW Accelerator as a PYNQ Overlay - element14 Community, 9월 25, 2025에 액세스, https://community.element14.com/challenges-projects/design-challenges/eye-on-intelligence-challenge/b/blog/posts/blog-5-integrating-the-hw-accelerator-as-a-pynq-overlay
36. Overlay — Python productivity for Zynq (Pynq) - Read the Docs, 9월 25, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_libraries/overlay.html
37. Designing with Dynamic Function eXchange (DFX) Using the Vivado Design Suite - Logtel, 9월 25, 2025에 액세스, https://www.logtel.com/training/designing-with-dynamic-function-exchange-dfx-using-the-vivado-design-suite/
38. How exactly Overlay load works? - Support - PYNQ, 9월 25, 2025에 액세스, https://discuss.pynq.io/t/how-exactly-overlay-load-works/6337
39. Vivado Design Suite Tutorial: Dynamic Function eXchange (UG947) - 2025.1 English, 9월 25, 2025에 액세스, https://docs.amd.com/r/en-US/ug947-vivado-partial-reconfiguration-tutorial
40. Adding Reconfigurable Modules to Partitions in the Composable Pipeline Design - PYNQ, 9월 25, 2025에 액세스, https://discuss.pynq.io/t/adding-reconfigurable-modules-to-partitions-in-the-composable-pipeline-design/5455