# **PYNQ-Z2**에서 동적 기능 교환**(DFX)**을 이용한 실시간 하드웨어 교체 구현

## 동적 기능 교환**(DFX)**의 기본 개념

### **Zynq SoC**에서의 재구성 가능 컴퓨팅 소개

Zynq SoC(System on Chip)는 ARM 기반의 프로세싱 시스템(Processing System, PS)과 FPGA(Field-Programmable Gate Array) 기반의 프로그래머블 로직(Programmable Logic, PL)을 단일 칩에 통합한 하이브리드 아키텍처를 특징으로 합니다. 이 구조는 소프트웨어의 유연성과 하드웨어 가속의 성능을 동시에 활용할 수 있는 강력한 플랫폼을 제공합니다. 특히, 이러한 구조는 동적 기능 교환(Dynamic Function eXchange, DFX) 기술에 매우 적합합니다. DFX는 시스템의 특정 부분은 계속 동작하는 동안 PL의 다른 부분을 동적으로 재구성하는 기술을 의미합니다.[1]

Zynq SoC에서 DFX를 활용할 때의 핵심적인 이점은 PS가 PL 재구성을 위한 고수준 제어기 역할을 수행할 수 있다는 점입니다. 순수 FPGA 설계에서는 재구성 과정을 관리하기 위해 PL 내부에 DFX Controller IP와 같은 별도의 상태 머신을 구현해야 할 수 있습니다.[1] 반면, Zynq에서는 PS의 프로세서가 리눅스와 같은 운영체제 상에서 소프트웨어 코드를 실행하여 PCAP(Processor Configuration Access Port)를 통해 PL로 부분 비트스트림을 전송하고 재구성 프로세스를 직접 제어할 수 있습니다.[1] 이는 외부 프로그래밍 장치 없이 시스템 자체적으로 하드웨어 기능을 업그레이드하거나 교체할 수 있게 하여, DFX 구현의 복잡성을 크게 낮추고 접근성을 높입니다.

### **DFX** 패러다임**:** 정적 로직과 동적 로직

DFX 기술은 하드웨어 설계를 정적인 부분과 동적인 부분으로 분할하는 개념에 기반합니다.

이는 단순한 기술적 기능을 넘어, 하드웨어 시스템을 보다 유연하고 서비스 지향적인 플랫폼으로 바라보는 아키텍처적 전환을 의미합니다. 정적 영역은 안정적인 "하드웨어 운영체제"처럼 동작하며, 필요에 따라 다양한 "하드웨어 애플리케이션"을 로드할 수 있게 됩니다. 이는 필드 업데이트, 시스템 수명 연장, 자원 관리 측면에서 중요한 가치를 가집니다.[1]

DFX 설계의 핵심 구성 요소는 다음과 같습니다:

- 정적 영역 **(Static Region):** 재구성 과정 중에도 변경되지 않고 계속해서 동작하는 설계 영역입니다. 이 영역에는 시스템의 핵심 기능, 메모리 인터페이스, 클럭 생성 회로, 그리고 DFX 프로세스를 관리하는 제어 로직 등이 포함됩니다.[2]
- 재구성 파티션 **(Reconfigurable Partition, RP):** PL 패브릭 내에 동적 로직을 위해 미리 할당된 물리적 영역, 즉 "슬롯"입니다. 이 영역은 설계 단계에서 플로어플래닝을 통해 위치와 크기가 정의됩니다.[2]
- 재구성 모듈 **(Reconfigurable Module, RM):** RP에 로드될 수 있는 특정 기능의 로직 블록입니다. 예를 들어, 본 보고서의 led_shifter_left와 led_shifter_right 모듈이 각각의 RM에 해당합니다. 하나의 RP는 여러 개의 RM을 가질 수 있지만, 특정 시점에는 하나의 RM만 활성화될 수 있습니다.[2]

이들의 관계는 명확합니다. RM은 논리적인 기능의 변형들이며, 이들은 물리적으로 RP라는 정해진 경계 내에 구현됩니다.

## **DFX** 워크플로우**: RTL**부터 동적 재구성까지

DFX 프로젝트의 전체적인 워크플로우는 다음과 같은 단계로 구성됩니다:

1. 설계 분할**:** 전체 시스템을 정적 로직과 여러 개의 RM HDL 모듈로 논리적으로 분할합니다.
2. **Vivado DFX** 프로젝트 설정**:** Vivado Design Suite에서 프로젝트를 생성하고, DFX 기능을 활성화하여 툴이 RP와 RM을 인식하도록 설정합니다.[3]
3. 구현 실행 **(Implementation Runs):** 초기 RM과 정적 로직을 포함하는 "부모(parent)" 실행을 생성하고, 이후 다른 RM들에 대해서는 "자식(child)" 실행을 생성합니다. 자식 실행은 부모 실행에서 구현된 정적 로직의 라우팅 정보를 재사용하여 전체 컴파일 시간을 단축시킵니다.[9]
4. 비트스트림 생성**:** 초기 시스템 구성을 위한 전체 비트스트림(full bitstream)과 각 RM에 해당하는 여러 개의 부분 비트스트림(partial bitstream)을 생성합니다.[2]
5. 동적 로딩**:** 제어기(본 프로젝트에서는 Zynq PS)를 사용하여 PCAP과 같은 구성 포트를 통해 부분 비트스트림을 PL에 동적으로 로드합니다.[3]

## 핵심 설계 고려사항**:** 디커플링**,** 클러킹**,** 리셋

안정적인 DFX 시스템을 구축하기 위해서는 몇 가지 중요한 설계 원칙을 반드시 준수해야 합니다.

- 디커플링 **(Decoupling):** 재구성 과정 동안 RP의 출력 신호는 정의되지 않은 상태(undefined state)가 됩니다. 이 불안정한 신호가 정적 영역으로 전파되어 시스템 전체의 오작동이나 멈춤 현상을 유발하는 것을 방지하기 위해, 정적 로직과 RP 사이의 인터페이스를 논리적으로 분리(decoupling)하는 과정이 필수적입니다.[1] 디커플링은 정적 로직에 위치한 멀티플렉서, 클럭 게이팅, 또는 레지스터의 인에이블 신호를 제어하는 방식으로 구현될 수 있습니다.[4]
- 인터페이스 무결성 **(Interface Integrity):** 특정 RP에 로드될 모든 RM들은 반드시 동일한 인터페이스(포트 이름, 방향, 비트 폭)를 가져야 합니다. 이는 단순한 코딩 규칙이 아니라 물리적인 제약 조건입니다. 정적 로직은 RP 경계에 있는 고정된 "파티션 핀"으로 라우팅됩니다. 만약 RM의 인터페이스가 다르면 논리적 연결이 물리적 연결 지점과 일치하지 않아 구현 단계에서 오류가 발생합니다.[5]
- 클러킹 및 리셋 **(Clocking and Reset):** MMCM이나 PLL과 같은 클럭 생성 및 수정 블록은 반드시 정적 영역에 위치해야 합니다.[6] 또한, 부분 비트스트림이 로드된 후 새로운 RM을 안정적인 초기 상태로 만들기 위한 적절한 리셋 시퀀스가 중요합니다. Vivado에서는 Pblock 속성에서 RESET_AFTER_RECONFIG를 설정하여 재구성 완료 후 자동으로 초기화 시퀀스를 수행하도록 할 수 있습니다.[4]

# 재구성 가능한 **LED** 시프터 모듈의 하드웨어 설계

## 시스템 아키텍처 및 인터페이스 정의

본 프로젝트의 하드웨어 아키텍처는 매우 간단하게 구성됩니다. 시스템은 클럭 분주기(clock divider)와 리셋 로직을 포함하는 정적 영역과, LED 점멸 패턴 로직이 위치할 단일 RP로 구성됩니다.

정적 영역과 RP 간의 인터페이스는 모든 RM에 걸쳐 일관되게 유지되어야 합니다. 본 설계에서 RP의 인터페이스는 다음과 같이 정의됩니다.

- clk_i: 클럭 입력 신호
- rst_ni: 액티브 로우(active-low) 리셋 입력 신호
- leds_o: PYNQ-Z2 보드의 LED 4개에 연결될 4비트 출력 벡터

led_shifter_left와 led_shifter_right 두 RM은 모두 이 인터페이스 정의를 정확히 따라야 합니다.

## Verilog 구현: led_shifter_left 모듈

led_shifter_left 모듈은 LED 불빛이 오른쪽에서 왼쪽으로 이동하는 패턴을 생성합니다. 이 모듈은 내부 카운터를 사용하여 PYNQ-Z2의 125 MHz 시스템 클럭을 분주하여 사람이 인지할 수 있는 속도로 LED 상태를 변경합니다. 모듈이 재구성 가능하도록 표시되면 Vivado는 해당 모듈과 그 경계에 DONT_TOUCH 속성을 자동으로 설정합니다. 이는 정적 영역과 동적 영역 간의 경계를 넘어선 최적화를 방지하는 중요한 역할을 합니다.[5] 따라서 설계자는 경계를 넘는 모든 신호를 명시적으로 레지스터링하여 타이밍 문제를 예방하고 안정적인 인터페이스를 보장해야 할 책임이 있습니다.

다음은 led_shifter_left 모듈의 전체 Verilog 소스 코드입니다.

Verilog

```
`timescale 1ns / 1ps

module led_shifter_left (
    input  wire        clk_i,
    input  wire        rst_ni,
    output reg  [3:0]  leds_o
);

    // 파라미터: 클럭 분주를 위한 카운터 최대값 (125MHz -> 2Hz)
    localparam COUNTER_MAX = 26'd31250000;

    reg [25:0] counter_reg;
    reg [3:0]  led_pattern_reg;

    // 클럭 분주기 및 LED 패턴 시프터 로직
    always @(posedge clk_i or negedge rst_ni) begin
        if (!rst_ni) begin
            counter_reg     <= 26'd0;
            led_pattern_reg <= 4'b0001; // 초기 패턴: 가장 오른쪽 LED 켜짐
        end else begin
```

```
        if (counter_reg == COUNTER_MAX) begin
            counter_reg <= 26'd0;
            // LED 패턴을 왼쪽으로 한 칸 시프트
            // 4'b1000 -> 4'b0001 로 순환
            if (led_pattern_reg == 4'b1000) begin
                led_pattern_reg <= 4'b0001;
            end else begin
                led_pattern_reg <= led_pattern_reg << 1;
            end
        end else begin
            counter_reg <= counter_reg + 1;
        end
    end
  end

  // 출력 할당
  assign leds_o = led_pattern_reg;

endmodule
```

## Verilog 구현: led_shifter_right 모듈

led_shifter_right 모듈은 led_shifter_left 모듈과 정확히 동일한 포트 정의를 가집니다. 내부 로직만 변경되어 LED 불빛이 왼쪽에서 오른쪽으로 이동하도록 구현됩니다. 이는 시프트 연산자 >>를 사용하여 간단히 구현할 수 있습니다.[12]

다음은 led_shifter_right 모듈의 전체 Verilog 소스 코드입니다.

Verilog

```
`timescale 1ns / 1ps

module led_shifter_right (
  input  wire       clk_i,
  input  wire       rst_ni,
  output reg  [3:0]  leds_o
);
```

```
    // 파라미터: 클럭 분주를 위한 카운터 최대값 (125MHz -> 2Hz)
    localparam COUNTER_MAX = 26'd31250000;

    reg [25:0] counter_reg;
    reg [3:0]  led_pattern_reg;

    // 클럭 분주기 및 LED 패턴 시프터 로직
    always @(posedge clk_i or negedge rst_ni) begin
        if (!rst_ni) begin
            counter_reg     <= 26'd0;
            led_pattern_reg <= 4'b1000; // 초기 패턴: 가장 왼쪽 LED 켜짐
        end else begin
            if (counter_reg == COUNTER_MAX) begin
                counter_reg <= 26'd0;
                // LED 패턴을 오른쪽으로 한 칸 시프트
                // 4'b0001 -> 4'b1000 로 순환
                if (led_pattern_reg == 4'b0001) begin
                    led_pattern_reg <= 4'b1000;
                end else begin
                    led_pattern_reg <= led_pattern_reg >> 1;
                end
            end else begin
                counter_reg <= counter_reg + 1;
            end
        end
    end

    // 출력 할당
    assign leds_o = led_pattern_reg;

endmodule
```

## 재구성 파티션을 위한 최상위 래퍼

최상위(top-level) 모듈은 Zynq PS 인스턴스와 재구성될 모듈의 인스턴스를 포함합니다. 본 프로젝트에서는 IP 통합기(IP Integrator)를 사용하여 Zynq PS를 포함하는 블록 디자인(Block Design, BD)을 생성하고, 이 BD의 HDL 래퍼를 최상위 RTL 파일에서 인스턴스화하는 하이브리드 방식을 사용합니다. 이 방식은 PS 설정의 복잡성을 BD가 처리하게 하고, PL 로직은

RTL로 유연하게 기술할 수 있어 효율적입니다.[14]

최상위 파일에서 재구성될 모듈(led_shifter_inst)을 인스턴스화하고, 이 인스턴스를 Vivado DFX 플로우에서 RP로 지정하게 됩니다.

Verilog

```
`timescale 1ns / 1ps

module top_level_wrapper (
    // Zynq PS 고정 I/O
    inout  wire [53:0]  DDR,
    inout  wire         FIXED_IO_ps_clk,
    inout  wire         FIXED_IO_ps_porb,
    inout  wire         FIXED_IO_ps_srstb,
    // LED 출력
    output wire [3:0]   leds_out
);

    // Zynq PS에서 생성된 클럭 및 리셋 신호
    wire FCLK_CLK0;
    wire FCLK_RESET0_N;

    // 재구성 모듈 인스턴스
    // 이 인스턴스가 DFX 플로우에서 재구성 파티션으로 지정됨
    led_shifter_left led_shifter_inst (
        .clk_i(FCLK_CLK0),
        .rst_ni(FCLK_RESET0_N),
        .leds_o(leds_out)
    );
    // 참고: 초기 설계에서는 RM 중 하나(예: led_shifter_left)를
    // 인스턴스화해야 함. Vivado가 이 인스턴스의 포트 정보를
    // 기반으로 RP의 인터페이스를 정의하기 때문임.

    // IP 통합기에서 생성된 Zynq PS 블록 디자인 래퍼 인스턴스
    zynq_bd_wrapper zynq_bd_wrapper_i (
        .DDR(DDR),
        .FCLK_CLK0(FCLK_CLK0),
        .FCLK_RESET0_N(FCLK_RESET0_N),
        .FIXED_IO_ps_clk(FIXED_IO_ps_clk),
```

```
    .FIXED_IO_ps_porb(FIXED_IO_ps_porb),
    .FIXED_IO_ps_srstb(FIXED_IO_ps_srstb)
  );

endmodule
```

# DFX 플로우를 이용한 Vivado 프로젝트 구현

## DFX가 활성화된 PYNQ-Z2용 프로젝트 설정

DFX 프로젝트를 생성하는 과정은 표준 RTL 프로젝트 생성 과정과 유사하지만, DFX 기능을 활성화하는 중요한 단계가 추가됩니다.

1. **Vivado** 실행 및 프로젝트 생성**:** Vivado Design Suite를 실행하고 Create Project를 선택합니다.
2. 프로젝트 유형 선택**:** RTL Project를 선택하고 소스 파일을 추가하지 않은 상태로 진행합니다.
3. 보드**/**부품 선택**:** Boards 탭에서 PYNQ-Z2를 선택하거나, Parts 탭에서 xc7z020clg400-1을 직접 선택합니다.[15]
4. **DFX** 기능 활성화**:** 프로젝트가 생성된 후, Flow Navigator에서 Project Manager -> Settings로 이동하거나, 상단 메뉴에서 Tools -> Enable Dynamic Function eXchange를 선택합니다. 이 작업은 프로젝트에 대해 되돌릴 수 없다는 경고 메시지가 나타나며, Convert를 클릭하여 DFX 프로젝트로 전환합니다.[8]

## DFX를 위한 IP 통합기 및 RTL 기반 설계 구조화

1. 블록 디자인 생성**:** Flow Navigator에서 IP Integrator -> Create Block Design을 클릭하여 새로운 BD를 생성합니다.
2. **ZYNQ7 PS** 추가**:** Diagram 뷰에서 + 버튼을 클릭하고 ZYNQ7 Processing System을 검색하여 추가합니다.
3. **PS** 설정 자동화**:** Run Block Automation을 클릭하여 PYNQ-Z2 보드 프리셋에 맞게 DDR 및 FIXED_IO 설정을 자동으로 구성합니다.
4. **PL** 클럭 및 리셋 활성화**:** Zynq PS 블록을 더블 클릭하여 Re-customize IP 창을 엽니다.

Clock Configuration -> PL Fabric Clocks에서 FCLK_CLK0을 활성화합니다. PS-PL Configuration -> AXI Non Secure Enablement -> GP Master AXI Interface에서 M AXI GP0 인터페이스를 활성화하고, General -> Enable Clock Resets에서 FCLK_RESET0_N을 활성화합니다.

5. **HDL** 래퍼 생성**:** Sources 뷰에서 BD 파일(.bd)을 마우스 오른쪽 버튼으로 클릭하고 Create HDL Wrapper를 선택하여 Vivado가 BD를 관리하도록 합니다.
6. **RTL** 소스 추가**:** 위에서 작성한 led_shifter_left.v, led_shifter_right.v, top_level_wrapper.v 파일을 프로젝트에 추가하고, top_level_wrapper.v를 최상위 모듈로 설정합니다.

## **DFX** 마법사**:** 설정**(Configuration)** 및 실행**(Run)** 정의

설계 합성이 완료된 후, DFX 마법사를 사용하여 재구성 파티션과 모듈, 그리고 구현 전략을 정의합니다.

1. 합성 실행**:** Run Synthesis를 실행합니다.
2. 재구성 파티션 정의**:** 합성이 완료되면 Open Synthesized Design을 클릭합니다. Netlist 뷰에서 top_level_wrapper 아래의 led_shifter_inst 인스턴스를 마우스 오른쪽 버튼으로 클릭하고 Create Partition Definition을 선택합니다. Partition Definition name에 led_shifter_pd와 같이 이름을 지정하고 OK를 클릭합니다.[8]
3. **DFX** 마법사 실행**:** 상단 메뉴에서 Tools -> Dynamic Function eXchange Wizard를 실행합니다.[9]
    - **Reconfigurable Modules** 편집**:** led_shifter_pd 파티션이 선택된 것을 확인하고 + 버튼을 클릭하여 RM을 추가합니다. led_shifter_left와 led_shifter_right를 각각 RM으로 추가합니다. Module name을 지정하고 Source file에서 해당 Verilog 파일을 선택합니다.[9]
    - **Configurations** 편집**:** + 버튼을 두 번 클릭하여 두 개의 설정을 생성합니다. 첫 번째 설정의 이름을 config_left로 변경하고 RM으로 led_shifter_left를 할당합니다. 두 번째 설정의 이름을 config_right로 변경하고 RM으로 led_shifter_right를 할당합니다.[9]
    - **Configuration Runs** 생성**:** Automatically create configuration runs 옵션을 선택합니다. Vivado는 config_left를 부모 실행(impl_1)으로, config_right를 자식 실행(child_0_impl_1)으로 자동으로 구성합니다. 이 부모-자식 관계는 자식 실행이 부모의 정적 라우팅 결과를 재사용하여 구현 시간을 단축하는 DFX 플로우의 핵심입니다.[9]
    - 마법사를 완료하면 Design Runs 뷰에 impl_1과 child_0_impl_1이 생성된 것을 확인할 수 있습니다.

## 물리적 제약 조건 및 플로어플래닝

DFX 설계에서는 재구성 파티션(RP)이 위치할 물리적 영역을 지정하기 위해 Pblock이라는 제약 조건을 사용해야 합니다.

1. 부모 실행 구현**:** Design Runs 뷰에서 impl_1을 마우스 오른쪽 버튼으로 클릭하고 Launch Runs를 선택하여 구현을 시작합니다.
2. 구현된 설계 열기**:** impl_1 실행이 완료되면 Open Implemented Design을 클릭합니다.
3. **Pblock** 생성**:** Device 뷰에서 RP로 지정할 영역을 마우스로 드래그하여 선택합니다. 선택된 영역에서 마우스 오른쪽 버튼을 클릭하고 Create Pblock을 선택합니다.
4. **Pblock**에 **RP** 할당**:** Pblock Properties 창에서 Cells 탭으로 이동하여 + 버튼을 클릭합니다. Choose Physical Objects 창에서 led_shifter_inst를 찾아 추가합니다.
5. **XDC** 제약 조건 파일 추가**:** 프로젝트에 새로운 제약 조건 파일(.xdc)을 추가하고 PYNQ-Z2 보드의 LED 핀 정보를 입력합니다. 이 정보는 보드마다 고유하므로 정확한 값을 사용하는 것이 매우 중요합니다.

표 **1: PYNQ-Z2 LED** 핀 제약 조건 **(XDC)**

이 표는 Verilog 모듈의 4비트 출력을 PYNQ-Z2 보드의 물리적 LED에 연결하는 데 필요한 정확한 XDC 제약 조건을 제공합니다. 이 정보는 프로젝트의 정상 동작을 위해 필수적입니다.[17]

| 포트 이름 (Verilog) | 패키지 핀 | I/O 표준 |
|---|---|---|
| leds_out | R14 | LVCMOS33 |
| leds_out | P14 | LVCMOS33 |
| leds_out | N16 | LVCMOS33 |
| leds_out | M14 | LVCMOS33 |

코드 스니펫

```
# PYNQ-Z2 On-board LEDs
set_property -dict { PACKAGE_PIN R14   IOSTANDARD LVCMOS33 } [get_ports { leds_out }];
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { leds_out }];
set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { leds_out }];
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { leds_out }];
```

6. 설계 저장 및 재실행**:** 제약 조건을 저장한 후, 모든 구현 실행을 다시 시작합니다.

## 전체 및 부분 비트스트림 생성

모든 구현 실행이 성공적으로 완료되면, 최종 산출물인 비트스트림을 생성합니다.

1. 비트스트림 생성**:** Flow Navigator에서 Program and Debug -> Generate Bitstream을 클릭합니다. Vivado는 impl_1에 대한 전체 비트스트림과 child_0_impl_1에 대한 부분 비트스트림을 모두 생성합니다.
2. 산출물 확인**:** 생성된 파일들은 각 실행의 디렉터리 내에 위치합니다. PYNQ 보드에서 DFX를 실행하기 위해 필요한 핵심 파일들은 다음과 같습니다.

표 **2: DFX** 프로젝트 생성 파일 요약

이 표는 DFX 빌드 프로세스의 결과물을 명확히 설명하고, PYNQ 보드로 복사해야 할 필수 파일들을 식별하여 배포 과정에서의 실수를 방지하는 체크리스트 역할을 합니다.[2]

| 파일 이름 패턴 | 위치 | 유형 | PYNQ에서의 용도 및 목적지 |
|---|---|---|---|
| top_level_wrapper.bit | .../project.runs/impl_1/ | 전체 비트스트림 | 초기 하드웨어 디자인 (정적 로직 + led_shifter_left). 가장 먼저 로드됨. |
| top_level_wrapper.hwh | .../project.srcs/.../hw_handoff/ | 하드웨어 핸드오프 | 전체 디자인의 메타데이터. PYNQ에서 .bit 파일과 동일한 디렉터리에 위치해야 함. |
| led_shifter_inst_led_shifter_right_partial.bit | .../project.runs/child_0_impl_1/ | 부분 비트스트림 | led_shifter_right 모듈의 로직만 포함. 동적으로 로드됨. |

PYNQ 프레임워크는 하드웨어의 메타데이터를 파싱하기 위해 .hwh 파일에 크게 의존합니다.[19] 표준 DFX 플로우는 부분 비트스트림에 대한 별도의 .hwh 파일을 생성하지 않습니다. 따라서 재구성될 모듈의 메타데이터를 PYNQ가 인식하게 하려면, 해당 모듈을 포함하는 전체

비트스트림과 그에 상응하는 .hwh 파일을 생성해야 하는 경우가 있습니다. 본 프로젝트와 같이 RM 내부의 IP에 직접 접근할 필요가 없는 간단한 경우에는 전체 비트스트림의 .hwh 파일 하나만으로 충분할 수 있지만, 복잡한 설계에서는 각 설정(configuration)에 대한 전체 비트스트림과 .hwh 파일을 모두 생성하는 것이 안전한 접근 방식입니다.

# **PYNQ**에서의 소프트웨어 제어 및 동적 재구성

## **PYNQ** 환경 및 프로젝트 파일 준비

Vivado에서 생성된 필수 파일들을 PYNQ 보드로 전송해야 합니다.

1. 파일 전송**:** scp나 Jupyter Notebook의 업로드 기능을 사용하여 표 2에서 식별된 파일들을 PYNQ 보드의 특정 디렉터리(예: /home/xilinx/pynq/dfx_led_shifter/)로 복사합니다.
   - top_level_wrapper.bit
   - top_level_wrapper.hwh
   - led_shifter_inst_led_shifter_right_partial.bit
2. 파일 이름 변경 **(**선택 사항**):** 가독성을 위해 파일 이름을 간결하게 변경할 수 있습니다. 예를 들어, top_level_wrapper.bit -> base.bit, led_shifter_inst_led_shifter_right_partial.bit -> shift_right.bit 등으로 변경합니다.

## **pynq** 라이브러리를 사용한 **DFX** 제어용 **Python** 스크립트

PYNQ 프레임워크는 pynq.Overlay 클래스를 통해 PL과의 상호작용을 추상화합니다. 이 클래스는 전체 및 부분 비트스트림 로드를 모두 처리합니다.[19]

- 전체 비트스트림 로드**:** Overlay 클래스를 인스턴스화하면 지정된 전체 비트스트림이 PL에 자동으로 다운로드됩니다. 이 과정에서 정적 로직과 초기 RM(led_shifter_left)이 프로그래밍됩니다.[19]
- 부분 비트스트림 로드**:** Overlay 객체의 download() 메서드를 사용하여 부분 재구성을 수행합니다. 이때, 인자로 부분 비트스트림 파일의 경로를 전달합니다.[19] 이 단일 함수 호출은 내부적으로 복잡한 재구성 시퀀스를 처리합니다. PYNQ의 리눅스 FPGA 관리자 프레임워크는 PCAP에 비트스트림을 쓰고, 관련 디바이스 트리 오버레이를 관리하는 저수준 작업을 자동으로 수행합니다.[21] 이는 설계자가 복잡한 하드웨어 제어 시퀀스를 직접 구현할 필요 없이 소프트웨어 수준에서 DFX를 쉽게 활용할 수 있게 해주는 강력한

추상화입니다.[10]

## 동적 재구성 실행 및 검증

다음 Python 코드는 PYNQ의 Jupyter Notebook 환경에서 실행되어 동적 재구성을 시연합니다.

1. 초기 오버레이 로드: Overlay('base.bit')를 실행하면 PL이 프로그래밍되고, PYNQ 보드의 LED가 오른쪽에서 왼쪽으로 이동하기 시작합니다.
2. 동적 재구성: overlay.download('shift_right.bit')를 실행하면, 시스템이 멈추거나 리셋되지 않고 즉시 LED의 점멸 패턴이 왼쪽에서 오른쪽으로 이동하는 것으로 변경됩니다. 이는 PL의 특정 부분만 성공적으로 교체되었음을 시각적으로 증명합니다.

## 전체 **Python** 소스 코드 및 상세 설명

다음은 PYNQ-Z2에서 DFX를 시연하기 위한 완전한 Python 스크립트입니다.

Python

```python
# 필요한 라이브러리 임포트
from pynq import Overlay
import time

# 1. 초기 전체 비트스트림 로드
# 이 비트스트림은 정적 로직과 'led_shifter_left' RM을 포함합니다.
# 로드가 완료되면 PL이 프로그래밍되고 LED가 왼쪽으로 이동하기 시작합니다.
print("초기 오버레이(base.bit)를 로드합니다...")
overlay = Overlay('base.bit')
print("로드 완료. LED가 왼쪽으로 5초간 점멸합니다.")

# 왼쪽으로 이동하는 패턴을 5초간 관찰
time.sleep(5)

# 2. 부분 비트스트림을 이용한 동적 재구성
# download() 메서드에 부분 비트스트림 파일 경로를 전달합니다.
# PYNQ 프레임워크가 PCAP를 통해 해당 RM을 재구성 파티션에 로드합니다.
```

```
print("\n부분 비트스트림(shift_right.bit)을 로드하여 동적 재구성을 수행합니다...")
overlay.download('shift_right.bit')
print("재구성 완료. LED가 오른쪽으로 5초간 점멸합니다.")

# 오른쪽으로 이동하는 패턴을 5초간 관찰
time.sleep(5)

print("\n데모가 완료되었습니다.")
```

이 스크립트는 DFX의 핵심 가치, 즉 실행 중인 시스템의 하드웨어 기능을 중단 없이 동적으로 변경하는 능력을 명확하게 보여줍니다.

# 디버깅 및 고급 고려사항

## **DFX** 설계의 일반적인 함정 및 문제 해결

DFX 설계는 강력하지만, 몇 가지 일반적인 문제에 직면할 수 있습니다.

- 플로어플래닝 문제**:** Pblock의 크기가 너무 작거나, 위치가 부적절하거나, 형태가 복잡하면 라우팅 실패로 이어질 수 있습니다. Vivado의 Report DRC에서 DFX 관련 규칙 검사를 실행하여 문제를 조기에 발견하는 것이 중요합니다.[4]
- 인터페이스 불일치**:** RM 간의 인터페이스가 일치하지 않으면 구현 단계에서 치명적인 오류가 발생합니다. 모든 RM의 포트 선언이 RP의 인터페이스와 정확히 일치하는지 반복해서 확인해야 합니다.[5]
- **PYNQ** 파일 오류**:** Jupyter 환경에서 FileNotFoundError가 발생하면 .bit 또는 .hwh 파일이 올바른 경로에 있는지 확인해야 합니다. PYNQ가 재구성 영역을 인식하지 못하는 AttributeError가 발생하면, .hwh 파일이 누락되었거나 내용이 부정확할 가능성이 높습니다.
- **JTAG** 연결 문제**:** 하드웨어 디버깅 시 Vivado가 보드를 인식하지 못하는 경우, JTAG 케이블 연결, 보드 전원, 드라이버 설치 상태 등 기본적인 연결 상태를 먼저 점검해야 합니다.[22]

## **Vivado** 하드웨어 관리자를 이용한 재구성 검증

LED 점멸 패턴을 시각적으로 확인하는 것 외에, Vivado 하드웨어 관리자(Hardware Manager)와 ILA(Integrated Logic Analyzer) IP를 사용하여 재구성을 보다 체계적으로 검증할 수 있습니다.

ILA 코어를 정적 영역에 배치하고, 프로브를 정적 영역과 RP 사이의 인터페이스 신호(예: leds_o)에 연결합니다. 이를 통해 Python 스크립트로 부분 재구성을 트리거했을 때, ILA의 파형 뷰에서 leds_o 신호의 동작 패턴이 실시간으로 변경되는 것을 관찰할 수 있습니다. 이는 로직이 성공적으로 교체되었음을 객관적인 데이터로 증명하는 강력한 디버깅 방법입니다.[24]

## 결론 및 프로젝트 확장 방안

본 보고서는 PYNQ-Z2 보드에서 DFX 기술을 사용하여 두 개의 간단한 LED 점멸 로직을 동적으로 교체하는 전체 과정을 상세히 다루었습니다. Verilog를 사용한 재구성 모듈 설계부터 Vivado DFX 플로우를 통한 구현, 그리고 PYNQ Python 라이브러리를 이용한 소프트웨어 제어에 이르기까지, DFX 프로젝트의 전체 생명주기를 살펴보았습니다. 이를 통해 Zynq SoC의 PS-PL 아키텍처가 어떻게 DFX를 효율적으로 지원하며, PYNQ 프레임워크가 이 강력한 기능을 얼마나 쉽게 활용할 수 있도록 추상화하는지 확인할 수 있었습니다.

본 프로젝트에서 습득한 개념을 바탕으로 다음과 같은 확장을 시도해볼 수 있습니다.

- 복잡한 **RM** 추가**:** 단순한 LED 시프터 대신, 서로 다른 DSP 필터나 이미지 처리 알고리즘(예: Sobel 필터, 가우시안 블러)을 RM으로 구현하여 실시간으로 교체하는 시스템을 구축합니다.
- 외부 입력 기반 재구성**:** PYNQ 보드의 푸시 버튼 입력을 받아 특정 버튼이 눌렸을 때 DFX 재구성이 일어나도록 제어 로직을 수정합니다.
- 다중 재구성 파티션**:** 하나의 설계 안에 여러 개의 독립적인 RP를 생성하고, 각각 다른 RM들로 동시에 또는 순차적으로 재구성하는 고급 DFX 설계를 구현합니다.
- 블록 디자인 컨테이너**(BDC)** 활용**:** IP 통합기 내에서 계층적 블록을 재구성 파티션으로 직접 지정할 수 있는 BDC(Block Design Container) 기능을 탐색합니다. 이는 순수 RTL 기반 DFX 플로우의 대안으로, 모든 설계를 블록 디자인 환경 내에서 관리할 수 있는 장점을 제공합니다.[3]

참고 자료


1. Dynamic Function eXchange (DFX) - AMD, 9월 25, 2025에 액세스, https://www.amd.com/en/products/adaptive-socs-and-fpgas/technologies/dynamic-function-exchange.html
2. Dynamic Function eXchange (DFX) — Kria SOM DFX Examples 1.0 documentation - GitHub Pages, 9월 25, 2025에 액세스, https://xilinx.github.io/kria-apps-docs/dfx.html
3. Dynamic Function eXchange (DFX) Using the Vivado Design Suite - BLT Inc., 9월 25, 2025에 액세스,

https://bltinc.com/xilinx-training-courses/dynamic-function-exchange/
4. Partial Reconfiguration in Zynq, 9월 25, 2025에 액세스, https://picture.iczhiku.com/resource/eetop/WhldtqahYuEaAvvv.pdf
5. Dynamic Function eXchange - Technology Blogs, 9월 25, 2025에 액세스, https://blog.abbey1.org.uk/index.php/technology/dynamic-function-exchange
6. Introduction to Partial Reconfiguration Methodology - BME-MIT, 9월 25, 2025에 액세스, https://home.mit.bme.hu/~szanto/education/vimima15/heterogen_pr_intro.pdf
7. Dynamic Function eXchange through ICAP for Zynq Devices - 2025.1 English - UG909, 9월 25, 2025에 액세스, https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration/Dynamic-Function-eXchange-through-ICAP-for-Zynq-Devices
8. 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/
9. Vivado Design Suite Tutorial: Dynamic Function eXchange, 9월 25, 2025에 액세스, https://static.eetrend.com/files/2021-12/wen_zhang_/100556482-232279-ug947-vivado-partial-reconfiguration-tutorial.pdf
10. Xilinx Partial Reconfiguration: Reset and decoupling - 01signal.com, 9월 25, 2025에 액세스, https://www.01signal.com/vendor-specific/xilinx/partial-reconfiguration/part3-decoupling-and-reset/
11. Static and Dynamic regions in partial reconfiguration of FPGA - Reddit, 9월 25, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/hm9fo6/static_and_dynamic_regions_in_partial/
12. Verilog Operators - Alchitry, 9월 25, 2025에 액세스, https://alchitry.com/tutorials/verilog/verilog-operators/
13. Shift Operator - Verilog Example - Nandland, 9월 25, 2025에 액세스, https://nandland.com/shift-operator/
14. Tutorial to understand Zynq block design implementation for new students [XILINX] - Reddit, 9월 25, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/hs8dkp/tutorial_to_understand_zynq_block_design/
15. Xilinx Design Constraints | FPGA Design with Vivado - GitHub Pages, 9월 25, 2025에 액세스, https://xilinx.github.io/xup_fpga_vivado_flow/lab5.html
16. DFX BDC Project Flow in IP Integrator for Zynq UltraScale+ - 2025.1 ..., 9월 25, 2025에 액세스, https://docs.amd.com/r/en-US/ug947-vivado-partial-reconfiguration-tutorial/DFX-BDC-Project-Flow-in-IP-Integrator-for-Zynq-UltraScale
17. XDC files for Arty and Pynq boards - GitHub Gist, 9월 25, 2025에 액세스, https://gist.github.com/fcayci/eb913f5f17cbf107d624448a7282b631
18. counter example part 2 · GitHub, 9월 25, 2025에 액세스, https://gist.github.com/fcayci/8669e6ea54147980730eed95de1ccca8

19. Partial Reconfiguration — Python productivity for Zynq (Pynq) v1.0, 9월 25, 2025에 액세스, https://pynq.readthedocs.io/en/v2.4/pynq_overlays/partial_reconfiguration.html
20. PYNQ, Partial Reconfiguration, Part 2 - DJ Park, 9월 25, 2025에 액세스, https://dj-park.github.io/posts/2022/1/PYNQ_PR_2/
21. pynq.bitstream Module — Python productivity for Zynq (Pynq) - Read the Docs, 9월 25, 2025에 액세스, https://pynq.readthedocs.io/en/v2.6.1/pynq_package/pynq.bitstream.html
22. Can't Connect to Zynq Via JTAG : r/FPGA - Reddit, 9월 25, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/zmujot/cant_connect_to_zynq_via_jtag/
23. Vivado Design Suite User Guide Programming and Debugging, 9월 25, 2025에 액세스, https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug908-vivado-programming-debugging.pdf
24. Debugging Dynamic Function eXchange (DFX) Designs in Vivado Hardware Manager - 2025.1 English - UG908, 9월 25, 2025에 액세스, https://docs.amd.com/r/en-US/ug908-vivado-programming-debugging/Debugging-Dynamic-Function-eXchange-DFX-Designs-in-Vivado-Hardware-Manager
25. Debugging Versal Device DFX Designs - 2025.1 English - UG909, 9월 25, 2025에 액세스, https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration/Debugging-Versal-Device-DFX-Designs
26. [PP-11] Lab 8 - hardware debugging of Zynq designs - element14 Community, 9월 25, 2025에 액세스, https://community.element14.com/challenges-projects/design-challenges/pathtoprogrammable/b/blog/posts/pp-11-lab-8---hardware-debugging-of-zynq-designs