Zynq 기반 시스템을 위한 PYNQ 빌드 및 배포 워크플로우 종합 가이드

Part 1: PYNQ 프레임워크의 핵심 개념

1.1. PYNQ 해부: "Python Productivity for Zynq"에서 하드웨어-소프트웨어 공동 설계 생태계까지

PYNQ는 단순히 이름의 약자를 넘어선 설계 철학을 담고 있습니다. PYNQ(Python Productivity on Zynq)는 AMD가 적응형 컴퓨팅 플랫폼을 더 쉽게 사용할 수 있도록 개발한 오픈 소스 프레임워크입니다.[1] 이 프레임워크의 핵심 목표는 전통적으로 분리되어 있던 두 전문 분야, 즉 하드웨어 기술 언어(HDL)로 작업하는 하드웨어(FPGA) 설계자와 Python과 같은 고급 언어로 작업하는 소프트웨어 개발자 사이의 간극을 메우는 것입니다. 이를 통해 병렬 실행, 고대역폭 I/O, 실시간 처리와 같은 프로그래머블 로직(PL)의 고유한 이점을 활용하는 데 필요한 진입 장벽을 크게 낮춥니다.[3]

PYNQ 시스템은 본질적으로 세 가지 핵심 요소로 구성됩니다. 첫째, 부팅 가능한 Linux 이미지입니다. 이 이미지는 Zynq SoC의 ARM 프로세서 시스템(PS)에서 실행되는 완전한 운영 체제를 제공합니다. 둘째, pynq Python 패키지입니다. 이 패키지는 PL과 상호 작용하는 데 필요한 모든 API와 라이브러리를 포함합니다. 셋째, Jupyter Notebook 환경입니다. 이 웹 기반 대화형 환경은 Zynq의 ARM 프로세서에서 직접 실행되며, 개발자가 원격으로 시스템에 접속하여 코드를 작성하고 실행하며 디버깅할 수 있는 기본 인터페이스 역할을 합니다.[3]

이러한 구성 요소들의 조합은 소프트웨어 개발자가 ASIC 스타일의 설계 도구를 사용하지 않고도 Zynq 디바이스의 강력한 성능을 활용할 수 있게 해줍니다. 즉, 복잡한 하드웨어 설계 지식 없이도 Python 코드 몇 줄로 FPGA의 성능을 끌어낼 수 있는 길을 열어주는 것입니다. 이는 시스템 아키텍트에게는 신속한 프로토타이핑 도구를, 하드웨어 설계자에게는 자신들의 설계를 더 넓은 사용자층에게 제공할 수 있는 통로를 제공합니다.[3]

1.2. 오버레이 패러다임: 하드웨어 라이브러리를 소프트웨어처럼 이해하기

PYNQ 프레임워크의 가장 중요하고 혁신적인 개념은 "오버레이(Overlay)"입니다. 오버레이는 PL에 로드되는 사전 컴파일된 하드웨어 디자인, 즉 비트스트림(bitstream)을 의미합니다. 소프트웨어 개발자의 관점에서 이 하드웨어 디자인은 마치 소프트웨어 라이브러리처럼 취급됩니다.[2]

이해를 돕기 위해 소프트웨어 개발의 비유를 들어보겠습니다. 소프트웨어 엔지니어는 수학 라이브러리에서 math.sqrt() 함수를 호출할 때, 제곱근을 계산하는 복잡한 알고리즘 구현을 알 필요가 없습니다. 단지 함수의 이름과 입출력만 알면 됩니다. PYNQ의 오버레이도 이와 동일한 원리로 작동합니다. PYNQ 사용자는 특정 기능을 수행하도록 설계된 오버레이를 인스턴스화하고, Python API를 통해 하드웨어 가속 기능을 호출할 수 있습니다. 이 과정에서 Verilog나 VHDL 코드를 직접 다룰 필요가 없습니다.[2]

오버레이는 기술적으로 두 가지 핵심 파일로 구성됩니다. 첫 번째는 .bit 파일로 알려진 비트스트림입니다. 이 파일은 PL의 물리적 회로 구성을 정의하는 원시 데이터입니다. 두 번째는 .hwh(Hardware Handoff) 파일입니다. 이 파일은 하드웨어 디자인의 구조에 대한 메타데이터를 담고 있으며, Python 환경이 해당 하드웨어의 IP 코어, 메모리 맵, 인터럽트 등을 인식하고 상호작용할 수 있도록 해주는 "설명서" 역할을 합니다.[7] 이

.hwh 파일의 존재가 PYNQ의 동적인 특성을 가능하게 하는 핵심 요소입니다. Overlay("my_design.bit")와 같은 Python 코드가 실행될 때, PYNQ 런타임은 해당 .hwh 파일을 파싱하여 하드웨어 구조를 동적으로 파악하고, 이를 Python 객체로 매핑합니다. 이 메커니즘 덕분에 소프트웨어 개발자는 하드웨어의 세부 주소 맵을 외울 필요 없이 overlay.my_ip.register_map과 같은 직관적인 속성을 통해 하드웨어 레지스터에 접근할 수 있습니다.[10]

이 패러다임은 "한 번 빌드하고, 여러 번 재사용한다(build once, re-use many times)"는 철학을 실현합니다. 새로운 오버레이를 제작하려면 여전히 FPGA 전문 엔지니어의 역할이 필요하지만, 일단 만들어진 오버레이는 훨씬 더 광범위한 소프트웨어 개발자 커뮤니티에서 손쉽게 재사용될 수 있습니다. 이는 소프트웨어 개발에서 잘 정립된 라이브러리 재사용 모델을 하드웨어 영역으로 확장한 것입니다.[2]

1.3. PYNQ 생태계: 핵심 구성 요소와 상호 작용

PYNQ 프레임워크는 여러 강력한 오픈 소스 기술들을 통합하여 하나의 원활한 개발 환경을

구축합니다. 각 구성 요소는 서로 긴밀하게 상호 작용하며 PYNQ의 생산성을 극대화합니다.

- **Jupyter Notebooks:** PYNQ 개발의 중심에는 Jupyter Notebook이 있습니다. 이 브라우저 기반의 대화형 컴퓨팅 환경은 코드, 설명 텍스트, 수식, 시각화 자료(플롯, 이미지) 및 인터랙티브 위젯을 하나의 문서에 통합할 수 있게 해줍니다.[2] 개발자는 Zynq 보드에서 직접 실행되는 이 환경을 통해 임베디드 시스템 자체에서 코드를 작성하고 즉시 실행 결과를 확인할 수 있습니다. 이는 호스트 PC와 타겟 보드 사이를 오가며 컴파일하고 다운로드해야 했던 전통적인 임베디드 개발 방식에 비해 훨씬 효율적인 워크플로우를 제공합니다.[12]
- **Python APIs:** PYNQ는 프로그래밍 언어로 Python을 채택하여 개발 생산성을 극적으로 향상시킵니다. 임베디드 시스템 프로그래밍은 전통적으로 C/C++가 지배해왔지만, Python은 더 높은 수준의 추상화를 제공하여 복잡한 작업을 더 적은 코드로 빠르게 구현할 수 있게 합니다.[2] PYNQ는 CPython을 사용하기 때문에, Python의 방대한 라이브러리 생태계를 그대로 활용할 수 있을 뿐만 아니라, 성능이 중요한 특정 부분에 대해서는 C로 작성된 최적화된 라이브러리를 호출하는 것도 가능합니다. 이는 생산성과 성능이라는 두 마리 토끼를 모두 잡는 유연한 접근 방식을 제공합니다.[2]
- **Embedded Linux:** PYNQ는 Ubuntu를 기반으로 하는 완전한 기능의 Linux 배포판 위에서 동작합니다. 이는 개발자에게 익숙하고 강력한 운영 환경을 제공합니다. 네트워킹 스택, 파일 시스템 관리, 표준 개발 도구(컴파일러, 셸 스크립트 등)와 같은 Linux의 모든 기능을 활용할 수 있습니다.[14] 이를 통해
  pip를 사용하여 Python 패키지를 설치하거나, git을 사용하여 소스 코드를 직접 보드로 클론하는 등 현대적인 소프트웨어 개발 워크플로우를 임베디드 환경에 그대로 적용할 수 있습니다.

1.4. 전체 워크플로우: 두 갈래 빌드 프로세스의 개요

PYNQ 시스템을 구축하는 전체 과정은 크게 두 개의 독립적이면서도 상호 연결된 경로로 나눌 수 있습니다. 이 두 경로는 각각 하드웨어와 소프트웨어 개발을 담당하며, 최종적으로 하나의 통합된 시스템으로 수렴합니다.

- 경로 1: 하드웨어 개발 (Vivado)
  이 경로는 하드웨어 설계자의 영역으로, AMD Vivado Design Suite에서 시작됩니다. 설계자는 IP Integrator를 사용하여 Zynq PS를 구성하고, 프로그래머블 로직(PL)에 필요한 커스텀 로직이나 IP 코어를 추가하여 시스템을 설계합니다. 이 과정이 완료되면, 설계자는 디자인을 합성(synthesis), 구현(implementation)하고 비트스트림을 생성합니다. 최종 산출물은 XSA(Xilinx Shell Archive) 파일로, 이 파일은 비트스트림, 하드웨어 핸드오프 파일(.hwh), 그리고 시스템 구성에 필요한 모든 정보를 포함하는 압축 아카이브입니다. 이 단계는 PYNQ에서 사용할 "오버레이"를 생성하는 과정입니다.
- 경로 2: 소프트웨어/OS 개발 (PYNQ GitHub)
  이 경로는 소프트웨어 개발자의 영역으로, PYNQ GitHub 저장소를 클론하는 것에서 시작됩니다. 개발자는 특정 버전의 Ubuntu 호스트 시스템과 AMD 도구(PetaLinux, Vitis

등)로 구성된 빌드 환경을 구축해야 합니다. 이 환경에서 make 명령어를 실행하면, PYNQ의 빌드 시스템이 PetaLinux를 내부적으로 사용하여 부트로더(FSBL, U-Boot), Linux 커널, 그리고 Ubuntu 기반의 루트 파일 시스템을 컴파일합니다. 이 모든 구성 요소는 최종적으로 하나의 부팅 가능한 SD 카드 이미지(.img) 파일로 패키징됩니다.

- 두 경로의 통합:
  이 두 경로는 두 가지 방식으로 통합될 수 있습니다. 첫 번째는, 경로 1에서 생성된 커스텀 하드웨어 디자인(XSA)을 경로 2의 빌드 프로세스에 입력으로 제공하여, 해당 하드웨어를 기본으로 포함하는 맞춤형 PYNQ 이미지를 생성하는 것입니다. 두 번째는, 이미 범용 PYNQ 이미지가 실행 중인 보드에 경로 1에서 생성된 오버레이 파일들(.bit, .hwh)을 전송하여 동적으로 PL을 재구성하는 방식입니다. 엔지니어는 이 두 경로가 어떻게 독립적으로 진행되고 최종적으로 어떻게 결합되는지를 이해하는 것이 PYNQ 개발의 전체 그림을 파악하는 데 매우 중요합니다.

Part 2: AMD Vivado를 이용한 하드웨어 플랫폼 개발

2.1. Vivado 환경 준비**:** 도구 버전**,** 라이선스 및 보드 파일 통합

PYNQ 하드웨어 플랫폼 개발을 시작하기 전에, 성공적인 설계를 위해 필수적인 환경을 구축하는 것이 중요합니다. 이 단계에서의 실수는 이후 디버깅하기 어려운 문제로 이어질 수 있으므로 세심한 주의가 필요합니다.

- 도구 버전 관리**:** PYNQ 프로젝트의 가장 중요한 제약 조건 중 하나는 PYNQ 릴리스 버전과 이에 대응하는 AMD 도구(Vivado, Vitis, PetaLinux) 버전이 엄격하게 일치해야 한다는 점입니다. 예를 들어, PYNQ v2.7은 Vivado 2020.2와 함께 사용해야 합니다.[16] 버전 불일치는 빌드 실패의 가장 흔한 원인이므로, 개발을 시작하기 전에 반드시 목표 PYNQ 버전에 맞는 도구 버전을 확인하고 설치해야 합니다. 이 호환성 정보는 Part 3에서 상세한 표로 제공될 것입니다.
- 라이선스**:** Vivado의 무료 버전인 WebPACK은 PYNQ-Z1/Z2와 같은 다수의 보드를 지원하기에 충분합니다.[3] 하지만, ZCU104와 같은 고성능 보드의 공식 베이스 오버레이에 포함된 HDMI 또는 CMAC과 같은 고급 IP 코어를 사용하거나 재구축하려면 평가 라이선스 또는 정식 라이선스가 필요할 수 있습니다.[14] 라이선스가 필요한 경우, AMD-Xilinx 제품 라이선스 사이트에서 평가 라이선스를 발급받아 Vivado License Manager를 통해 설치해야 합니다.
- 보드 파일 통합**:** 보드 파일은 특정 개발 보드의 하드웨어 구성(FPGA 부품 번호, 주변 장치 연결, DDR 설정 등)을 미리 정의해 놓은 파일들의 집합입니다. 이를 Vivado에 통합하면 프로젝트 생성 시 보드를 선택하는 것만으로 복잡한 초기 설정이 자동으로 완료됩니다.

보드 파일을 통합하는 가장 권장되는 현대적인 방법은 Vivado의 Tcl 콘솔에서 xhub 명령어를 사용하여 Xilinx Board Store에서 직접 다운로드하고 설치하는 것입니다.[18] 과거에는 수동으로 파일을 복사하는 방법도 사용되었으나 [20], xhub를 사용하면 버전 관리가 용이하고 실수를 줄일 수 있습니다.[21]

2.2. 기본 **Zynq** 시스템 생성

Vivado 환경이 준비되면, IP Integrator를 사용하여 PYNQ 시스템의 하드웨어 기반을 구축할 수 있습니다. IP Integrator는 시스템을 블록 다이어그램 형태로 시각적으로 설계할 수 있는 강력한 도구입니다.[21]

- 프로젝트 생성 및 **IP Integrator** 실행**:**
  1. Vivado를 실행하고 "Create Project"를 선택하여 새 RTL 프로젝트를 생성합니다.[23]
  2. 프로젝트 유형 화면에서 "RTL Project"를 선택하고, 소스는 나중에 추가하도록 설정합니다.
  3. 부품 선택 화면에서 "Boards" 탭으로 이동하여 이전에 설치한 PYNQ 보드(예: PYNQ-Z2)를 선택합니다.[25]
  4. 프로젝트가 생성되면 Flow Navigator에서 "IP INTEGRATOR" 아래의 "Create Block Design"을 클릭하여 새로운 블록 다이어그램을 엽니다.[23]
- **Zynq Processing System (PS)** 구성**:**
  1. 블록 다이어그램에서 마우스 오른쪽 버튼을 클릭하거나 '+' 아이콘을 눌러 "Add IP"를 선택하고, "ZYNQ7 Processing System" (또는 Zynq UltraScale+ MPSoC)을 검색하여 추가합니다.[24]
  2. IP가 추가되면 다이어그램 상단에 녹색 배너로 "Run Block Automation" 링크가 나타납니다. 이 단계는 매우 중요합니다**.** 이 링크를 클릭하면 Vivado가 보드 파일을 참조하여 DDR 메모리 컨트롤러, 이더넷, USB, SD 카드 인터페이스와 같은 고정 I/O(FIXED_IO) 등 PS의 모든 필수 설정을 자동으로 구성합니다.[24] 이 자동화 기능을 사용하지 않고 수동으로 DDR 타이밍이나 MIO 핀 할당을 설정하는 것은 매우 복잡하고 오류가 발생하기 쉬우며, 시스템 부팅 실패의 주요 원인이 됩니다. 자동화 기능은 전문가에 의해 검증된 설정을 적용하여 이러한 위험을 근본적으로 제거합니다.
- PS-PL 클럭 및 AXI 인터페이스 구성:
  PS 구성이 완료되면, PL과의 통신을 위한 인터페이스를 활성화해야 합니다.
  1. Zynq PS 블록을 더블 클릭하여 "Re-customize IP" 창을 엽니다.
  2. "Clock Configuration" 페이지로 이동하여 "PL Fabric Clocks"를 확장하고, FCLK_CLK0와 같은 클럭을 활성화한 후 원하는 주파수(예: 100MHz)를 설정합니다. 이 클럭은 PL 내부 로직의 동작 기준이 됩니다.[24]
  3. "PS-PL Configuration" 페이지로 이동하여 "GP Master AXI Interface" 또는 "HP Slave AXI Interface"를 확장합니다. PS가 PL의 IP를 제어하기 위해서는 M AXI GP0 인터페이스를, PL이 PS의 DDR 메모리에 직접 접근하기 위해서는 S AXI HP0

인터페이스를 활성화해야 합니다.[24]

2.3. 프로그래머블 로직(PL)에 커스텀 로직 통합

기본 Zynq 시스템이 구성되면, PL 영역에 실제 기능을 수행할 IP 코어를 추가할 차례입니다.

- 표준 IP 추가 및 연결 자동화:
  가장 간단한 예로, AXI GPIO IP를 블록 다이어그램에 추가해 보겠습니다. IP를 추가하면 다시 "Run Connection Automation"이라는 녹색 배너가 나타납니다. 이를 실행하면 Vivado가 자동으로 AXI Interconnect IP를 추가하고, Zynq PS의 M_AXI_GP0 포트와 AXI GPIO의 슬레이브 포트를 올바르게 연결하며, 클럭과 리셋 신호까지 연결해 줍니다. 또한, Address Editor에서 자동으로 메모리 주소를 할당해 줍니다. 이 기능은 복잡한 AXI 버스 프로토콜의 연결을 단순화하여 설계 시간을 단축하고 오류를 방지합니다.29
- 커스텀 AXI-Lite IP 생성 및 패키징:
  PYNQ의 진정한 힘은 사용자가 직접 설계한 로직을 가속기로 활용하는 데 있습니다. 이를 위해 AXI(Advanced eXtensible Interface) 버스 프로토콜을 이해하는 것이 중요합니다. AXI는 Zynq PS와 PL 간의 표준 통신 규약입니다. Vivado는 커스텀 IP를 쉽게 생성할 수 있도록 "Create and Package New IP" 마법사를 제공합니다.
  1. Tools -> Create and Package New IP를 선택합니다.
  2. "Create a new AXI4 peripheral" 옵션을 선택합니다.[31]
  3. IP의 이름, 버전, 설명을 입력합니다.
  4. 인터페이스 유형으로 "AXI4-Lite", 모드는 "Slave"로 설정하고, 필요한 레지스터 개수(예: 4개)를 지정합니다. AXI-Lite는 간단한 제어 및 상태 레지스터 접근에 적합한 경량화된 프로토콜입니다.[31]
  5. 마법사를 완료하면 Vivado는 AXI-Lite 슬레이브 로직과 레지스터가 포함된 VHDL/Verilog 템플릿 코드를 생성해 줍니다. 개발자는 이 템플릿에 자신의 로직(예: 간단한 덧셈기)을 추가하여 IP를 완성할 수 있습니다.
  6. IP 패키징이 완료되면, 이 커스텀 IP는 IP 카탈로그에 추가되어 다른 표준 IP와 동일한 방식으로 블록 다이어그램에 인스턴스화하고 연결할 수 있습니다.[31] 이 과정은 엔지니어가 기존 IP 사용을 넘어 자신만의 하드웨어 가속기를 PYNQ 시스템에 통합하는 능력을 갖추게 해줍니다.

2.4. 하드웨어 플랫폼 아티팩트 생성

블록 다이어그램 설계가 완료되면, PYNQ와 PetaLinux 같은 다운스트림 도구에서 사용할 수 있는 최종 결과물을 생성해야 합니다.

- 검증, 래퍼 생성, 합성 및 구현:
  1. "Validate Design" (F6 키)을 실행하여 블록 다이어그램에 연결 오류가 없는지 확인합니다.[19]
  2. Sources 창에서 블록 다이어그램 파일(.bd)을 마우스 오른쪽 버튼으로 클릭하고 "Create HDL Wrapper"를 선택합니다. Vivado가 블록 다이어그램을 인스턴스화하는 최상위 VHDL/Verilog 파일을 자동으로 생성하게 둡니다.[26]
  3. Flow Navigator에서 "Run Synthesis"와 "Run Implementation"을 차례로 실행합니다. 이 과정은 HDL 코드를 논리 게이트로 변환하고 FPGA의 물리적 자원에 배치 및 라우팅하는 과정입니다.[19]
- 비트스트림 생성:
  구현이 성공적으로 완료되면 "Generate Bitstream"을 실행합니다. 이 단계의 최종 산출물은 FPGA 패브릭을 직접 구성하는 바이너리 데이터인 .bit 파일입니다.35
- XSA(Extensible Hardware Archive) 내보내기:
  File -> Export -> Export Hardware 메뉴를 통해 하드웨어 플랫폼을 내보냅니다.26 이때 "Include bitstream" 옵션을 반드시 선택해야 합니다.36 이 과정은 .xsa 파일을 생성하는데, 이 파일은 PYNQ 빌드 프로세스의 핵심 입력물입니다. XSA는 단순히 비트스트림만 담고 있는 것이 아니라, 하드웨어 설계를 설명하는 데 필요한 모든 정보를 포함하는 zip 형식의 아카이브입니다.[38]
- 핵심 산출물의 역할:
  - **.bit (Bitstream):** PL을 위한 "실행 파일"입니다. 이 파일이 PL에 로드되면 물리적인 회로가 구성됩니다.[9]
  - **.hwh (Hardware Handoff):** 하드웨어를 위한 "메타데이터" 또는 "API 명세서"입니다. 이 XML 기반 파일은 시스템에 포함된 모든 IP, 메모리 맵 주소, 인터럽트 연결 정보 등을 상세히 기술합니다. PYNQ의 Python 라이브러리는 XSA 아카이브 내부에서 이 파일을 찾아 파싱함으로써, 소프트웨어가 하드웨어의 구조를 이해하고 제어할 수 있게 됩니다.[8]
  - **.tcl (Tcl Script):** write_bd_tcl 명령을 통해 블록 다이어그램을 Tcl 스크립트로 내보낼 수 있습니다. 이 파일은 PYNQ 런타임에서는 사용되지 않지만, Git과 같은 버전 관리 시스템으로 Vivado 프로젝트를 관리하고, 명령줄에서 프로젝트를 자동으로 재생성하는 데 필수적입니다.[7]

Part 3: 소스에서 **PYNQ SD** 카드 이미지 빌드하기

3.1. 빌드 환경 구축

PYNQ SD 카드 이미지를 소스 코드로부터 직접 빌드하는 과정은 매우 강력하지만, 그만큼 엄격한 환경 요구사항을 따릅니다. 이 단계는 전체 프로세스에서 가장 많은 시간이 소요되고 잠재적인 문제가 발생하기 쉬운 부분이므로, 정확한 환경 설정이 성공의 열쇠입니다.

- 호스트 시스템 요구사항**:** PYNQ 빌드 프로세스는 특정 버전의 Ubuntu LTS(Long-Term Support) 배포판에서만 검증되었습니다. 예를 들어, PYNQ v2.7은 Ubuntu 18.04 또는 20.04를 요구하며, PYNQ v3.1은 Ubuntu 22.04를 기반으로 합니다. 또한, 호스트 시스템에는 PYNQ 릴리스 버전에 맞는 특정 버전의 Vivado, Vitis, 그리고 PetaLinux가 설치되어 있어야 합니다.[15]
- 표 1: PYNQ 릴리스별 필수 AMD 도구 및 호스트 OS 버전
  새로운 개발 환경을 구성하는 엔지니어가 가장 먼저 확인해야 할 사항은 바로 이 호환성 매트릭스입니다. PYNQ, PetaLinux, AMD 개발 도구, 그리고 호스트 OS 간의 강한 종속성은 빌드 실패의 주된 원인이기 때문에, 아래 표를 기준으로 환경을 정확히 일치시키는 것이 매우 중요합니다.

PYNQ 릴리스 버전	필수 AMD 도구 버전	지원되는 호스트 OS
v2.5	2019.1	Ubuntu 18.04
v2.6	2020.1	Ubuntu 18.04
v2.7	2020.2	Ubuntu 18.04, 20.04
v3.0.1	2022.1	Ubuntu 20.04
v3.1	2024.1	Ubuntu 22.04

주: 위 표는 주요 릴리스를 기반으로 하며, 최신 정보는 PYNQ 공식 문서를 참조해야 합니다.[14]

- 권장 접근 방식: Docker 기반 빌드 흐름:
  이러한 복잡한 의존성 문제를 해결하기 위한 가장 현대적이고 권장되는 방법은 Docker를 사용하는 것입니다. Docker는 빌드에 필요한 모든 소프트웨어와 라이브러리를 격리된 컨테이너 환경에 패키징하여, 호스트 OS의 종류나 설정에 관계없이 일관되고 재현 가능한 빌드 환경을 제공합니다.14

  Docker 기반 빌드 절차는 다음과 같습니다:
  1. 호스트 시스템에 Docker를 설치합니다.
  2. 필수 버전의 AMD 도구(Vivado, Vitis, PetaLinux)를 호스트 시스템에 설치합니다. 이 도구들은 컨테이너 내부에 설치되는 것이 아니라, 호스트에 설치된 경로를 컨테이너와 공유하게 됩니다.[14]
  3. PYNQ 저장소를 클론한 후, sdbuild 디렉토리에서 제공된 Dockerfile을 사용하여 PYNQ

빌드용 Docker 이미지를 생성합니다.
4. 생성된 Docker 이미지를 실행하여 컨테이너를 시작합니다. 이때 -v 옵션을 사용하여 호스트에 설치된 AMD 도구 경로와 PYNQ 소스 코드 경로를 컨테이너 내부의 작업 공간에 마운트합니다.[14]

- 네이티브 Ubuntu 호스트에서의 수동 설정:
  Docker를 사용하지 않는 경우, 네이티브 Ubuntu 환경에 직접 빌드 환경을 구성할 수도 있습니다. 이 경우, PYNQ 저장소의 sdbuild/scripts/setup_host.sh 스크립트를 실행하여 빌드에 필요한 모든 의존성 패키지를 설치해야 합니다.16 그 후, 터미널 세션마다 AMD 도구들의 환경 설정 스크립트(
  settings64.sh)를 순서대로 source 명령어를 사용하여 로드해야 합니다.[14]

3.2. PYNQ 소스 저장소 및 빌드 시스템

- 저장소 클론**:** 빌드를 시작하려면 PYNQ 공식 GitHub 저장소를 클론해야 합니다. 이때 반드시 --recursive 옵션을 사용하여 모든 서브모듈을 함께 클론해야 합니다. 서브모듈에는 PYNQ 동작에 필수적인 라이브러리들이 포함되어 있어, 이 옵션을 누락하면 빌드가 실패할 수 있습니다.[14]

  ```
  git clone --recursive https://github.com/Xilinx/PYNQ.git
  ```
- 디렉토리 구조**:** 클론된 저장소에서 주목해야 할 두 가지 핵심 디렉토리가 있습니다.
  - sdbuild: SD 카드 이미지 빌드와 관련된 모든 파일이 위치하는 곳입니다. 메인 Makefile, 빌드 스크립트, 패키지 설정 등이 포함되어 있습니다.[45]
  - boards: 공식적으로 지원되는 보드 및 커스텀 보드를 위한 파일들이 위치합니다. 각 보드 디렉토리에는 해당 보드의 베이스 오버레이 소스 코드와 빌드 설정을 담은 .spec 파일이 들어 있습니다.[45]
- **PetaLinux**와 **Yocto** 기반**:** PYNQ의 sdbuild 시스템은 겉보기에는 간단한 Makefile처럼 보이지만, 그 내부에서는 PetaLinux라는 강력한 임베디드 Linux 개발 도구를 호출합니다. PetaLinux는 다시 Yocto Project라는 업계 표준 빌드 프레임워크를 기반으로 동작합니다.[15] 이 사실을 이해하는 것은 매우 중요합니다. 사용자가
  make 명령을 실행하면, 실제로는 수백, 수천 개의 소프트웨어 패키지를 소스 코드로부터 컴파일하고, 커널을 구성하며, 루트 파일 시스템을 생성하는 거대한 자동화 프로세스가 시작되는 것입니다. 이 배경 지식은 빌드 과정에서 문제가 발생했을 때, 단순한 Makefile 오류가 아니라 복잡한 임베디드 Linux 빌드 시스템의 문제일 수 있음을 인지하고 더 깊이 있는 디버깅을 가능하게 합니다.

3.3. 빌드 프로세스 실행

빌드 환경이 준비되면 make 명령어를 통해 이미지 생성을 시작할 수 있습니다.

- 공식 지원 보드 빌드: 모든 공식 지원 보드의 이미지를 한 번에 빌드하려면 sdbuild 디렉토리에서 make 명령만 실행하면 됩니다. 특정 보드 하나만 빌드하고 싶다면 BOARDS= 변수를 사용하여 빌드 시간을 단축할 수 있습니다.
  make BOARDS=Pynq-Z2.43
- 커스텀 보드 빌드**:** Part 2에서 생성한 XSA 파일을 사용하여 커스텀 보드용 PYNQ 이미지를 빌드하는 것은 PYNQ의 활용 범위를 크게 확장하는 핵심 기술입니다.
  1. 보드 디렉토리 생성**:** boards 디렉토리 내에 새로운 커스텀 보드 이름으로 디렉토리를 생성하거나, 별도의 경로를 BOARDDIR= 변수로 지정합니다.[15]
  2. 보드 명세 파일 **(.spec)** 작성**:** 생성한 디렉토리 안에 <BOARD_NAME>.spec 파일을 생성하고 다음 내용을 작성합니다.[14]
     - ARCH_<BOARD_NAME> := arm (Zynq-7000의 경우) 또는 aarch64 (Zynq UltraScale+의 경우)
     - BSP_<BOARD_NAME> := <path_to_bsp_file> 또는 비워둠
     - BITSTREAM_<BOARD_NAME> := <path_to_bitstream_file>
  3. 커스텀 **XSA** 통합**:** Part 2에서 생성한 .xsa 파일을 PetaLinux를 사용하여 BSP(Board Support Package)로 변환한 후, 이 BSP 파일의 경로를 .spec 파일의 BSP_ 변수에 지정하는 것이 일반적인 방법입니다.[39] 또는,
     .spec 파일의 BSP_ 변수를 비워두고 특정 규칙에 따라 XSA 파일을 위치시키면 PYNQ 빌드 시스템이 자동으로 BSP를 생성하도록 할 수도 있습니다.[15] PetaLinux는 XSA 파일 내부의 하드웨어 정보를 분석하여 디바이스 트리 소스, 부트로더 설정 등 Linux 부팅에 필요한 모든 구성 요소를 자동으로 생성합니다.
- 빌드 로그 해석 및 문제 해결**:** PYNQ 이미지 빌드는 수 시간이 걸릴 수 있는 긴 과정입니다.[51] 빌드 중 발생하는 오류는 대부분 환경 설정 문제에서 기인합니다.
  make 실행 초기에 checkenv 단계에서 실패한다면, AMD 도구의 경로 설정이나 버전이 잘못되었을 가능성이 높습니다.[52] 특정 IP와 관련된 라이선스 문제가 발생할 수도 있습니다.[17] 빌드가 실패하면,
  sdbuild/build/<BOARD_NAME>/petalinux_project/build.log와 같은 PetaLinux의 상세 로그 파일을 확인하는 것이 문제 해결의 지름길입니다. make의 최상위 출력 메시지만으로는 근본 원인을 파악하기 어렵기 때문입니다.

3.4. 빌드 결과물 분석

빌드가 성공적으로 완료되면 sdbuild/output 디렉토리에 최종 결과물인 .img 파일이 생성됩니다.[43] 이 이미지 파일은 단순한 파일이 아니라, 디스크 파티션 구조를 그대로 담고 있는 디스크 이미지입니다. 일반적으로 이 이미지는 두 개의 파티션으로 구성됩니다:

1. **BOOT** 파티션**:** FAT32 파일 시스템으로 포맷되어 있으며, 부팅에 필수적인 파일들을 포함합니다.
    - BOOT.BIN: FSBL(First Stage Boot Loader), PL 비트스트림(선택 사항), U-Boot 부트로더를 하나로 묶은 바이너리 파일입니다.
    - image.ub: Linux 커널과 램디스크를 포함하는 U-Boot 형식의 이미지 파일입니다.
    - system.dtb: 시스템의 하드웨어 구성을 설명하는 컴파일된 디바이스 트리 블롭(Device Tree Blob)입니다.[53]
2. 루트**(rootfs)** 파티션**:** ext4 파일 시스템으로 포맷되어 있으며, Ubuntu 기반의 전체 Linux 운영 체제, PYNQ Python 패키지, 라이브러리, 예제 노트북 등 모든 사용자 공간 소프트웨어를 포함합니다.

이 구조를 이해하면 SD 카드를 PC에 연결했을 때 왜 BOOT 파티션만 보이는지, 그리고 실제 운영 체제는 어디에 저장되는지를 명확히 알 수 있습니다.

Part 4: 배포**,** 검증 및 애플리케이션

4.1. 보드 준비 및 부팅

빌드된 SD 카드 이미지를 실제 하드웨어에 배포하고 시스템을 부팅하는 단계입니다.

- SD 카드 이미지 플래싱:
  생성된 .img 파일을 마이크로 SD 카드에 기록해야 합니다. 이 과정을 "플래싱"이라고 합니다.
    - 권장 도구**:** Windows, macOS, Linux에서 모두 사용 가능한 BalenaEtcher와 같은 그래픽 사용자 인터페이스(GUI) 도구를 사용하는 것이 가장 안전하고 쉽습니다. 사용자는 이미지 파일과 대상 SD 카드를 선택하고 "Flash!" 버튼을 클릭하기만 하면 됩니다. 이 도구들은 잘못된 드라이브(예: 시스템 하드 드라이브)를 선택하는 실수를 방지하는 안전장치를 포함하고 있습니다.[55]
    - 명령줄 도구**:** 고급 사용자는 Linux나 macOS에서 dd 명령어를 사용할 수 있습니다. 이 방법은 매우 강력하지만, 대상 장치(of=/dev/sdX)를 잘못 지정하면 시스템 데이터를 영구적으로 손상시킬 수 있으므로 극도의 주의가 필요합니다. df -h나 lsblk 명령으로 SD 카드의 정확한 장치 이름을 확인한 후 사용해야 합니다.[58]
- 보드 점퍼 설정:
  SD 카드를 보드에 삽입한 후, Zynq 칩이 SD 카드로부터 부팅하도록 부트 모드 점퍼를 올바르게 설정해야 합니다. 대부분의 PYNQ 보드에는 "SD", "QSPI", "JTAG" 모드를 선택하는 점퍼가 있으며, 이를 "SD" 위치로 설정해야 합니다. 이 설정은 보드 전원이

켜지는 순간에만 읽히므로, 점퍼 변경 후에는 반드시 전원을 껐다가 다시 켜야 합니다.53

- 부팅 순서 확인:
  보드에 전원을 인가하면 부팅 과정이 시작됩니다. 이 과정을 시각적으로 확인하는 것은 시스템이 정상적으로 동작하는지 판단하는 첫 번째 단계입니다. 일반적으로 다음과 같은 순서로 LED가 점등됩니다:
  1. 전원 인가 시 즉시 전원 LED(주로 빨간색)가 켜집니다.
  2. 잠시 후, Zynq 칩의 초기화 및 부트로더가 실행되면 특정 상태 LED가 켜지거나 깜박입니다.
  3. PL(FPGA)에 베이스 오버레이의 비트스트림이 성공적으로 로드되면 "DONE" LED(주로 녹색 또는 파란색)가 켜집니다.
  4. Linux 커널 부팅이 완료되고 PYNQ 서비스가 시작되면, 사용자 LED 여러 개가 특정 패턴으로 깜박이며 시스템이 준비되었음을 알립니다.[59] 이 LED 패턴을 통해 부팅의 어느 단계까지 진행되었는지 대략적으로 파악할 수 있습니다.

4.2. PYNQ 환경에 연결하기

보드가 성공적으로 부팅되면, 개발 PC에서 보드에 접속하여 Jupyter Notebook 환경을 사용할 수 있습니다.

- 네트워크 구성**:**
  - 직접 연결 **(**정적 **IP):** 가장 간단한 방법은 PYNQ 보드의 이더넷 포트와 개발 PC의 이더넷 포트를 케이블로 직접 연결하는 것입니다. 이 경우, PYNQ 보드는 기본적으로 192.168.2.99라는 정적 IP 주소를 가집니다. 개발 PC의 이더넷 어댑터 IP 주소를 동일한 서브넷의 다른 주소(예: 192.168.2.1)로 수동 설정해야 통신이 가능합니다.[12] 이 방식은 초기 설정 및 테스트에 편리하지만, 보드가 인터넷에 연결되지 않아 외부 패키지를 설치할 수 없는 단점이 있습니다.
  - 라우터 연결 **(DHCP):** 보드를 DHCP 서버 기능이 있는 네트워크 라우터에 연결하면, 보드는 자동으로 IP 주소를 할당받습니다. 할당된 IP 주소는 라우터의 관리 페이지에 있는 연결된 장치 목록에서 확인하거나, pynq라는 기본 호스트 이름을 사용하여 접근할 수 있습니다. 예를 들어, 웹 브라우저에서 http://pynq:9090 또는 http://pynq.local:9090으로 접속을 시도할 수 있습니다.[60] 이 방식은 보드가 인터넷에 연결될 수 있게 해주므로,
    pip로 Python 패키지를 설치하거나 git으로 저장소를 클론하는 등 본격적인 개발에 필수적입니다.
- Jupyter Notebook 인터페이스 접속:
  네트워크 연결이 확인되면, 개발 PC의 웹 브라우저를 열고 http://<보드_IP_주소>:9090 (또는 최신 이미지의 경우 /lab 접미사 추가)으로 접속합니다. 로그인 화면이 나타나면 기본 사용자 이름(xilinx)과 비밀번호(xilinx)를 입력하여 Jupyter 환경에 로그인할 수 있습니다.12
- 대체 접속 방법: 직렬 터미널 (UART):
  네트워크 설정 문제로 Jupyter에 접속할 수 없을 때, 직렬 터미널은 매우 유용한 디버깅

도구입니다.

1. 보드의 "UART"라고 표시된 마이크로 USB 포트와 개발 PC를 케이블로 연결합니다.
2. PC의 장치 관리자(Windows)나 ls /dev/tty* (Linux/macOS) 명령으로 가상 COM 포트 번호나 장치 파일 이름을 확인합니다.
3. PuTTY(Windows)나 screen(Linux/macOS)과 같은 터미널 에뮬레이터를 사용하여 해당 포트에 접속합니다. 통신 설정은 일반적으로 **115200 baud, 8 data bits, no parity, 1 stop bit (8N1)** 입니다.[12] 터미널에 연결되면 Linux 부팅 메시지를 실시간으로 확인하고, 로그인 프롬프트가 나타나면 셸 명령어를 실행하여 ifconfig 등으로 네트워크 상태를 직접 점검할 수 있습니다.

4.3. 커스텀 오버레이 로드 및 상호 작용

이제 Part 2에서 설계하고 Part 3에서 이미지에 통합(또는 별도 준비)한 커스텀 하드웨어 오버레이를 Python 코드로 제어할 차례입니다.

- 오버레이 파일 전송:
  만약 커스텀 오버레이가 SD 카드 이미지에 포함되지 않았다면, Part 2에서 생성한 .bit와 .hwh 파일을 보드로 전송해야 합니다. 두 파일은 반드시 같은 이름(확장자 제외)으로 같은 디렉토리에 위치해야 합니다. 파일 전송은 SCP, WinSCP와 같은 도구를 사용하거나 Jupyter Notebook의 업로드 기능을 통해 할 수 있습니다.30
- 실용적인 Python 예제:
  Jupyter Notebook에서 새 노트를 생성하고 다음 코드를 실행하여 오버레이를 로드하고 상호 작용할 수 있습니다.
  Python
  ```
  from pynq import Overlay

  #.bit 파일의 경로를 지정하여 오버레이를 로드합니다.
  # 이 시점에서 비트스트림이 PL에 다운로드되고,.hwh 파일이 파싱됩니다.
  overlay = Overlay('/home/xilinx/pynq/overlays/my_design/my_design.bit')
  ```
  8
  오버레이가 성공적으로 로드되면, PYNQ는 .hwh 파일의 정보를 바탕으로 블록 다이어그램에 있던 IP들을 Python 객체의 속성으로 동적으로 생성합니다. overlay? 또는 help(overlay) 명령을 실행하면 사용 가능한 IP 목록을 확인할 수 있습니다.[11]
- 레지스터 수준 상호 작용:
  Part 2에서 생성했던 커스텀 AXI-Lite IP(my_custom_ip_0)에 접근하여 레지스터를 직접 읽고 쓰는 방법은 다음과 같습니다.
  Python
  ```
  # IP 이름으로 속성에 접근하여 드라이버 객체를 가져옵니다.
  my_ip = overlay.my_custom_ip_0
  ```

```
# register_map 속성을 통해 레지스터에 이름으로 접근하여 값을 씁니다.
my_ip.register_map.reg0 = 0x12345678
my_ip.register_map.reg1 = 42

# 레지스터 값을 읽습니다.
val = my_ip.register_map.reg0.read()
print(f"Read value: {hex(val)}")

# 또는 read/write 메서드를 사용하여 주소 오프셋으로 직접 접근할 수도 있습니다.
# 0x00은 첫 번째 레지스터(reg0)의 주소 오프셋입니다.
my_ip.write(0x00, 0xABCD)
val_read = my_ip.read(0x00)
print(f"Read value via offset: {hex(val_read)}")
```

11

이 코드는 Vivado에서 설계한 하드웨어 레지스터와 Python 소프트웨어 간의 직접적인 연결을 명확하게 보여주며, 하드웨어-소프트웨어 공동 설계의 핵심을 체험하게 해줍니다.

- Python 드라이버 개발:
  매번 주소 오프셋을 사용하여 레지스터에 접근하는 것은 번거롭고 오류가 발생하기 쉽습니다. 더 나은 방법은 pynq.DefaultIP 클래스를 상속받는 커스텀 Python 클래스를 작성하여 하드웨어에 대한 더 높은 수준의 API를 만드는 것입니다. 이 클래스 내에 add(a, b)와 같은 메서드를 정의하고, 내부적으로 레지스터 읽기/쓰기를 캡슐화할 수 있습니다. 이렇게 하면 다른 소프트웨어 개발자들이 하드웨어의 세부 사항을 몰라도 쉽게 오버레이를 사용할 수 있게 됩니다.11
- 표 2: PYNQ 일반적인 문제 해결 가이드
  개발 과정에서 흔히 겪는 문제들을 해결할 수 있도록 다음의 문제 해결 가이드를 제공합니다.

증상	가능한 원인	해결 방안		
보드가 부팅되지 않음 (전원 **LED**만 켜짐)	1. 부트 모드 점퍼 설정 오류 2. SD 카드 이미지 손상 또는 플래싱 오류 3. SD 카드 불량	1. 보드 전원을 끄고 부트 모드 점퍼가 'SD'로 설정되었는지 확인 후 다시 켜기.[60]	2. SD 카드 이미지를 신뢰할 수 있는 도구(예: BalenaEtcher)로 다시 플래싱하기.55	3. 다른 SD 카드로 시도하기.

DONE LED가 켜지지 않음	1. BOOT.BIN에 비트스트림이 포함되지 않았거나 손상됨 2. 하드웨어 디자인(Vivado)의 클럭 또는 리셋 문제	1. 베이스 오버레이가 포함된 공식 이미지를 사용하거나, 커스텀 이미지 빌드 시 비트스트림이 올바르게 패키징되었는지 확인. 2. 직렬 터미널로 부팅 로그를 확인하여 오류 메시지 검색.	
Jupyter Notebook에 접속할 수 없음	1. 잘못된 IP 주소 또는 호스트 이름 사용 2. PC와 보드 간의 네트워크 연결 문제 (케이블, 방화벽 등) 3. PYNQ 서비스 시작 실패	1. 직접 연결 시 PC의 IP를 192.168.2.x로 설정했는지 확인. 라우터 연결 시 라우터 관리 페이지에서 보드의 IP 주소 확인.[63]	2. 이더넷 케이블 연결 상태 확인. PC의 방화벽이 포트 9090을 차단하는지 확인. 3. 직렬 터미널로 접속하여 ifconfig로 IP 주소를 확인하고, systemctl status pynq-notebook.service로 서비스 상태 점검.12
오버레이 로드 실패 **(Overlay... not found)**	1. .bit 파일 경로가 잘못됨 2. .hwh 파일이 없거나 이름이 .bit 파일과	1. Jupyter에서 파일 경로가 올바른지 확인. 절대 경로 사용 권장.	

	일치하지 않음	2. .bit 파일과 동일한 이름, 동일한 위치에 .hwh 파일이 있는지 확인.8
pip install 실패	보드가 인터넷에 연결되지 않음	보드를 직접 연결 방식이 아닌, 인터넷 게이트웨이가 있는 라우터에 연결하여 DHCP로 IP를 할당받도록 네트워크 구성 변경.[60]

결론

본 가이드는 PYNQ 프레임워크를 사용하여 Zynq 기반 시스템을 구축하는 전 과정을 체계적으로 안내하기 위해 작성되었습니다. PYNQ는 Python의 높은 생산성과 Zynq SoC의 하드웨어 가속 성능을 결합하여, 전통적인 임베디드 시스템 개발의 패러다임을 혁신하는 강력한 도구입니다.

보고서를 통해 PYNQ의 핵심인 오버레이 개념이 단순한 하드웨어 로딩을 넘어, .hwh 메타데이터를 통해 하드웨어를 동적으로 소프트웨어 객체화하는 정교한 추상화 메커니즘임을 확인했습니다. 또한, 성공적인 PYNQ 시스템 개발은 두 가지 핵심 축, 즉 **Vivado**를 통한 하드웨어 플랫폼 설계와 **PetaLinux** 기반의 **SD** 카드 이미지 빌드가 정확하게 맞물려야 함을 강조했습니다. 특히, Vivado의 자동화 기능 활용, PYNQ 버전과 AMD 도구 버전의 엄격한 일치, 그리고 Docker를 통한 재현 가능한 빌드 환경 구축은 프로젝트 실패율을 줄이는 데 결정적인 역할을 합니다.

궁극적으로, 이 가이드는 단순히 명령어의 나열을 넘어 각 단계의 '왜'를 설명함으로써 엔지니니어들이 문제 해결 능력을 갖추고, 단순한 사용자에서 나아가 자신만의 커스텀 하드웨어 가속기를 PYNQ 생태계에 통합할 수 있는 시스템 설계자로 성장하는 데 기여하고자 합니다. 제시된 체계적인 절차와 문제 해결 가이드를 통해 엔지니어들은 PYNQ 개발 워크플로우를 자신감 있게 수행하고, 하드웨어와 소프트웨어의 경계를 넘나드는 혁신적인 임베디드 애플리케이션을 구현할 수 있을 것입니다.

참고 자료

1. www.avnet.com, 9월 18, 2025에 액세스, https://www.avnet.com/apac/products/cp/xilinx-pynq/reference-materials/productivity-of-embedded-designs-with-pynq/#:~:text=PYNQ%20is%20defined%20as%3A%20Python,on%20Xilinx%20Zynq%20SoC%20devices.
2. Productivity of Embedded Designs with PYNQ - Avnet, 9월 18, 2025에 액세스, https://www.avnet.com/apac/products/cp/xilinx-pynq/reference-materials/productivity-of-embedded-designs-with-pynq/
3. PYNQ | Python Productivity for AMD Adaptive Computing platforms, 9월 18, 2025에 액세스, http://www.pynq.io/
4. PYNQ Framework on TityraCore Zynq® 7000 | Numato Lab Help Center, 9월 18, 2025에 액세스, https://numato.com/blog/pynq-framework-on-tityracore-zynq-7000/
5. PYNQ: Python Productivity for Zynq - Dustin Richmond, 9월 18, 2025에 액세스, https://www.dustinrichmond.com/pynq/
6. PYNQ Introduction — Python productivity for Zynq (Pynq), 9월 18, 2025에 액세스, https://pynq.readthedocs.io/
7. Overlay Design - Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/v2.4/overlay_design_methodology/overlay_design.html
8. Loading an Overlay — Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/latest/pynq_overlays/loading_an_overlay.html
9. Blog #5: CNN HW Accelerator for Handwriting Recognition - Integrating the HW Accelerator as a PYNQ Overlay - element14 Community, 9월 18, 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
10. PYNQ, Partial Reconfiguration, Part 2 - DJ Park, 9월 18, 2025에 액세스, https://dj-park.github.io/posts/2022/1/PYNQ_PR_2/
11. Overlay Tutorial — Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/latest/overlay_design_methodology/overlay_tutorial.html
12. Connecting to Jupyter Notebook — Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/v2.7.0/getting_started/pynq_sdcard_getting_started.html
13. Jupyter Notebooks — Python productivity for Zynq (Pynq) v1.0, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/v2.3/jupyter_notebooks.html
14. PYNQ SD Card image — Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/latest/pynq_sd_card.html
15. sdbuild/README.md · 7f4a3b92e09714d02545ffc8432124d5d9f1ad0d · paulrr2 / PYNQ - GitLab at Illinois, 9월 18, 2025에 액세스, https://gitlab-03.engr.illinois.edu/paulrr2/PYNQ/-/blob/7f4a3b92e09714d02545ffc

8432124d5d9f1ad0d/sdbuild/README.md
16. PYNQ SD Card image — Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/v2.7.0/pynq_sd_card.html
17. How to Build PYNQ v2.6 for Ultra96 - FPGA Developer, 9월 18, 2025에 액세스, https://www.fpgadeveloper.com/how-to-build-pynq-v2.6-for-ultra96/
18. please tell me which version of vivado have pynq z2 board..... - Adaptive Support - AMD, 9월 18, 2025에 액세스, https://adaptivesupport.amd.com/s/question/0D54U00006MndxBSAR/please-tell-me-which-version-of-vivado-have-pynq-z2-board?language=en_US
19. Vivado Tutorial • ECEn 625: Compilation Strategies for High-Performance Systems, 9월 18, 2025에 액세스, https://byu-cpe.github.io/ecen625/vivado-tutorial/
20. Add Pynq-Z2 board to Vivado - element14 Community, 9월 18, 2025에 액세스, https://community.element14.com/technologies/fpga-group/b/blog/posts/add-pynq-z2-board-to-vivado
21. Introduction to PYNQ - 6.S965: Digital Systems Laboratory II, 9월 18, 2025에 액세스, https://rfsoc.mit.edu/6S965/F24/assignments/week01/pynq_01
22. xupsh/pynq-supported-board-file - GitHub, 9월 18, 2025에 액세스, https://github.com/xupsh/pynq-supported-board-file
23. Getting Started with Zynq and the Vivado IP Integrator - Digilent Reference, 9월 18, 2025에 액세스, https://digilent.com/reference/vivado/getting_started_with_zynq/start
24. Creating a Base System for the Zynq in Vivado - FPGA Developer, 9월 18, 2025에 액세스, https://www.fpgadeveloper.com/2014/07/creating-a-base-system-for-the-zynq-in-vivado.html/
25. Getting started with a Zynq development board. Base PS Design. - controlpaths.com, 9월 18, 2025에 액세스, https://www.controlpaths.com/2023/10/14/getting-started-with-zynq/
26. Vivado Block Design Tutorial • BYU Computing Bootcamp, 9월 18, 2025에 액세스, https://byu-cpe.github.io/ComputingBootCamp/tutorials/vivado/
27. Step 1: Generating a Zynq UltraScale+ MPSoC PS Xilinx Support Archive - 2025.1 English, 9월 18, 2025에 액세스, https://docs.amd.com/r/en-US/ug936-vivado-tutorial-programming-debugging/Step-1-Generating-a-Zynq-UltraScale-MPSoC-PS-Xilinx-Support-Archive
28. PYNQ v2.7, Tutorial - DJ Park, 9월 18, 2025에 액세스, https://dj-park.github.io/posts/2021/12/PYNQ_TUTORIAL/
29. Adding Custom IP to the System | High Level Systhesis Design Flow, 9월 18, 2025에 액세스, https://xilinx.github.io/xup_embedded_system_design_flow/lab3.html
30. PYNQ Tutorial 1: GPIO - HackMD, 9월 18, 2025에 액세스, https://hackmd.io/@ween168/SyNPIkUb1x
31. Creating a custom IP block in Vivado - FPGA Developer, 9월 18, 2025에 액세스, https://www.fpgadeveloper.com/2014/08/creating-a-custom-ip-block-in-vivado.html/

32. Building Custom AXI IP, 9월 18, 2025에 액세스, https://www.xilinx.com/content/dam/xilinx/training/embedded/embd/Building-Custom-AXI-IP.pdf
33. PYNQ Tutorial 3: Custom IP core and the Block Design | regymm's blog, 9월 18, 2025에 액세스, https://www.ustcpetergu.com/MyBlog/tutorial/2020/03/07/pynq-3-ip-bd.html
34. Path to Programmable III Training Blog #04: My first Custom IP in Vivado, 9월 18, 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
35. Step 1: Create the Vivado Hardware Design and Generate XSA — Vitis™ Tutorials 2021.1 documentation - GitHub Pages, 9월 18, 2025에 액세스, https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/Vitis_Platform_Creation/Introduction/02-Edge-AI-ZCU104/step1.html
36. Exporting the XSA - 2025.1 English - XD260, 9월 18, 2025에 액세스, https://docs.amd.com/r/en-US/Vitis-Tutorials-Embedded-Software/Exporting-the-XSA
37. Export a Fixed Post-Synthesis Hardware Platform - Digilent Reference, 9월 18, 2025에 액세스, https://digilent.com/reference/programmable-logic/guides/vivado-export-fixed-platform
38. Registers Addressing for .HWH file - Support - PYNQ, 9월 18, 2025에 액세스, https://discuss.pynq.io/t/registers-addressing-for-hwh-file/1667
39. Customizing PYNQ for Pynq-Z1 - Support, 9월 18, 2025에 액세스, https://discuss.pynq.io/t/customizing-pynq-for-pynq-z1/1882
40. Ubuntu 24.04 LTS support - PYNQ, 9월 18, 2025에 액세스, https://discuss.pynq.io/t/ubuntu-24-04-lts-support/7947
41. Generating PYNQ SD card image with Ubuntu 20.04 - Support, 9월 18, 2025에 액세스, https://discuss.pynq.io/t/generating-pynq-sd-card-image-with-ubuntu-20-04/3251
42. Remote Image Build Guide — Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_remote/image_build.html
43. PYNQ Edition! Building PYNQ Images | by Adam Taylor | Medium, 9월 18, 2025에 액세스, https://medium.com/@aptaylorceng/pynq-edition-building-pynq-images-4f0ec30e172f
44. How to Build PYNQ v2.5 for Ultra96 - FPGA Developer, 9월 18, 2025에 액세스, https://www.fpgadeveloper.com/how-to-build-pynq-v2.5-for-ultra96/
45. Tutorial: Rebuilding the PYNQ base overlay (PYNQ v2.6), 9월 18, 2025에 액세스, https://discuss.pynq.io/t/tutorial-rebuilding-the-pynq-base-overlay-pynq-v2-6/1993
46. PYNQ Edition! Building PYNQ Images - Hackster.io, 9월 18, 2025에 액세스,

https://www.hackster.io/news/pynq-edition-building-pynq-images-4f0ec30e172f
47. Help building from source for Pynq - Support, 9월 18, 2025에 액세스, https://discuss.pynq.io/t/help-building-from-source-for-pynq/149
48. nick-petrovsky/PYNQ-ZYBO - GitHub, 9월 18, 2025에 액세스, https://github.com/nick-petrovsky/PYNQ-ZYBO
49. PYNQ build failed for custom board - Support, 9월 18, 2025에 액세스, https://discuss.pynq.io/t/pynq-build-failed-for-custom-board/2977
50. PYNQ Build Image for a Custom Board : r/FPGA - Reddit, 9월 18, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/1lsep3h/pynq_build_image_for_a_custom_board/
51. Help making a PYNQ Image. : r/FPGA - Reddit, 9월 18, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/m8x1wu/help_making_a_pynq_image/
52. How to build PYNQ image in Ubuntu 20.04 - Support, 9월 18, 2025에 액세스, https://discuss.pynq.io/t/how-to-build-pynq-image-in-ubuntu-20-04/3157
53. PYNQ-Z1 Reference Manual - Digilent, 9월 18, 2025에 액세스, https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual
54. Implementing the DPU on a SD card image - YOLO on PYNQ-Z2 - GitBook, 9월 18, 2025에 액세스, https://andre-araujo.gitbook.io/yolo-on-pynq-z2/dpu-implementation/implementing-the-dpu-on-a-sd-card-image
55. Flash the SD Card (only once) - TU/e PYNQ Pages, 9월 18, 2025에 액세스, https://pynq.tue.nl/general/flash-sd-card/
56. Flash with Balena Etcher - Raspberry Pi Forums, 9월 18, 2025에 액세스, https://forums.raspberrypi.com/viewtopic.php?t=279692
57. Balena Etcher (flashing the OS) - Knowledge Base - Unipi technology, 9월 18, 2025에 액세스, https://kb.unipi.technology/en:sw:05-tools:balenaetcher
58. Writing an SD Card Image — Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/latest/appendix/sdcard.html
59. Getting Started — NengoPYNQ 0.3.3.dev0 docs, 9월 18, 2025에 액세스, https://www.nengo.ai/nengo-pynq/getting-started.html
60. Getting Started — Python productivity for Zynq (Pynq) v1.0 - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/v2.1/getting_started.html
61. Getting started with your PYNQ-ZU, 9월 18, 2025에 액세스, https://xilinx.github.io/PYNQ-ZU/getting_started.html
62. Getting Started with PYNQ using EDGE ZYNQ SoC FPGA kit - Invent Logics, 9월 18, 2025에 액세스, https://allaboutfpga.com/getting-started-with-pynq-using-edge-zynq-soc-fpga-kit/
63. PYNQ-Z2 can't access Jupyter Notebook. : r/FPGA - Reddit, 9월 18, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/11o4h08/pynqz2_cant_access_jupyter_notebook/
64. Getting Started — Python productivity for Zynq (Pynq) - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/v2.5/getting_started.html

65. Adam Taylor's MicroZed Chronicles Part 158: Creating our own PYNQ hardware overlay, 9월 18, 2025에 액세스, https://adaptivesupport.amd.com/s/article/735388
66. PYNQ Edition! Creating Custom Overlays - Hackster.io, 9월 18, 2025에 액세스, https://www.hackster.io/news/pynq-edition-creating-custom-overlays-8543f45eccb1
67. Overlay Tutorial — Python productivity for Zynq (Pynq) v1.0 - Read the Docs, 9월 18, 2025에 액세스, https://pynq.readthedocs.io/en/v2.0/overlay_design_methodology/overlay_tutorial.html
68. Need some docs tutorials on pynq drivers - Support, 9월 18, 2025에 액세스, https://discuss.pynq.io/t/need-some-docs-tutorials-on-pynq-drivers/3456