



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 6

I/O 소개

1. 소개

LAB 6-10에서는 RVfpga의 I/O (입/출력) 시스템을 사용하고 확장하여 RISC-V 프로세서가 주변 장치와 상호 작용할 수 있도록 하는 방법을 배웁니다. 다음은 앞으로의 LAB에서 다루는 주제에 대한 개요입니다.

- **-LAB 6:** Nexys A7 보드의 LED, 스위치 및 푸시 버튼에 연결된 GPIO (범용 입력/출력) 핀을 사용하는 방법 알아보기
- **-LAB 7:** 보드에서 사용할 수 있는 7 세그먼트 디스플레이 사용법 배우기
- **-LAB 8:** 타이머 사용 방법 알아보기
- **-LAB 9:** 인터럽트를 사용하여 외부 장치와 인터페이스하는 방법 알아보기
- **-LAB 10:** RVfpga 시스템을 온 보드 SPI 가속도계와 인터페이스하는 방법 알아보기

이 LAB에서는 먼저 범용 I/O 시스템의 주요 기능과 RVfpga 시스템에서 사용되는 기능 (섹션 2)에 대해 설명합니다. 그런 다음 일반 GPIO 컨트롤러의 단순화된 이론적 버전을 설명합니다 (섹션 3). 마지막으로 SweRVofX SoC에서 사용되는 GPIO 컨트롤러에 초점을 맞춥니다. 먼저 상위 수준 사양을 분석하고 기본 LAB을 소개합니다 (섹션 4 및 5). 초급 수준에서 분석하고 Verilator에서 RVfpgaSim를 시뮬레이션하고, 고급 수준 LAB을 소개하면서 LAB을 마칩니다 (섹션 6 및 7).

Labs 7-10에서 이와 동일한 일반 구조를 사용합니다. 시작 섹션에서는 I/O 컨트롤러의 고급 사양 (주요 기능, 레지스터 및 작동, 메모리 맵)을 설명하고 주변 장치 사용을 위한 기본 LAB을 소개합니다. 고급 섹션에서는 컨트롤러의 초급 수준 구현에 대해 설명하고 이를 수정한 다음 테스트하는 프로그램을 작성하는 LAB을 제공합니다.

강사 참고 사항: 코스 수준에 따라 LAB의 복잡성을 선택할 수 있습니다. 예를 들어 1, 2년차 과정 (예: 컴퓨터 기초 또는 컴퓨터 조직)에서는 이 LAB 섹션 5의 기본 LAB이 적합합니다. 그러나 더 고급 과정 (예: 컴퓨터 아키텍처 또는 임베디드 시스템 설계)에서는 기본 및 고급 LAB (이 랩의 섹션 5 ~ 7)을 모두 사용할 수 있습니다.

2. 입력/출력 아키텍처

그림 1은 CPU, 메모리 및 I/O 시스템의 세 가지 주요 블록으로 구성된 Von Neumann 아키텍처의 구조를 보여줍니다. LAB 6-10에서는 CPU와 입출력 (I/O) 기기의 상호 작용에 중점을 둡니다. I/O 장치는 주변 장치 또는 단순히 장치라고도 합니다. 아래에서 각 주요 단위의 역할을 간략하게 설명합니다.

- **CPU:** CPU는 모든 I/O 작업의 시작점입니다. I/O 트랜잭션의 *컨트롤러* (역사적으로 "마스터"라 했지만 이 용어는 더 이상 사용되지 않음)입니다. 직접 메모리 액세스 (DMA) 컨트롤러 (DMAC)도 컨트롤러 역할을 할 수 있지만 이번 LAB에는 포함되지 않습니다.
- **장치 컨트롤러:** *장치 컨트롤러*는 작업을 수행하기 위해 *컨트롤러*의 읽기/쓰기 요청을 기다립니다. 장치 컨트롤러는 I/O 시스템에서 *주변 장치* (이전에는 "슬레이브"라 했지만 이 용어는 더 이상 사용되지 않음)로 작동합니다. 개념적으로 장치 컨트롤러는 컨트롤러에서 액세스할 수 있는 일련의 *레지스터*로 구성됩니다. 이러한 레지스터의 값은 수행할 작업에 대해서 주변 장치에 지시합니다.
- **상호 연결 (버스, 크로스바 등)**은 *컨트롤러*와 *주변 장치* 간의 경로를 설정합니다. 상호 연결은 일반적으로 특정 장치가 전체 시스템의 속도를 저하시키는 것을 방지하는 *브리지*를 통해 연결된 여러 계층으로 구현됩니다.

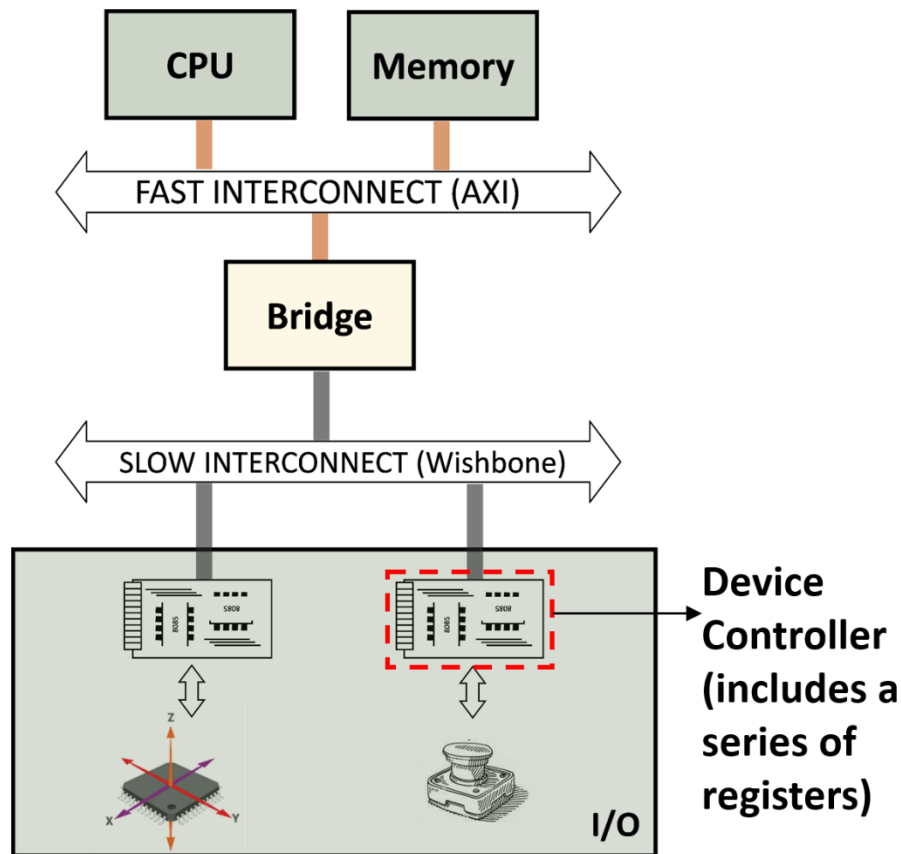


그림 1 일반 컴퓨팅 시스템

그림 2 는 RVfpga 의 I/O 시스템을 보여줍니다. 여기에는 다음과 같은 7 개의 주변 장치가 포함됩니다.

- GPIO1 모듈에 연결된 LED 및 스위치 (단일 주변 장치로 간주)
- 시스템 컨트롤러 모듈에 연결된 7 세그먼트 디스플레이
- SPI1 모듈에 연결된 플래시 메모리
- 가속도계, SPI2 모듈에 연결
- 타이머
- UART
- 부팅 ROM

멀티 플렉서는 7 가지 가능성 중 하나의 주변 장치를 선택하여 CPU 에 연결합니다. 주변 장치는 Wishbone 버스 (회색)를 사용하는 반면 SweRV EH1 Core 는 AXI 브리지 (주황색)를 사용하기 때문에 Wishbone to AXI Bridge 가 필요합니다.

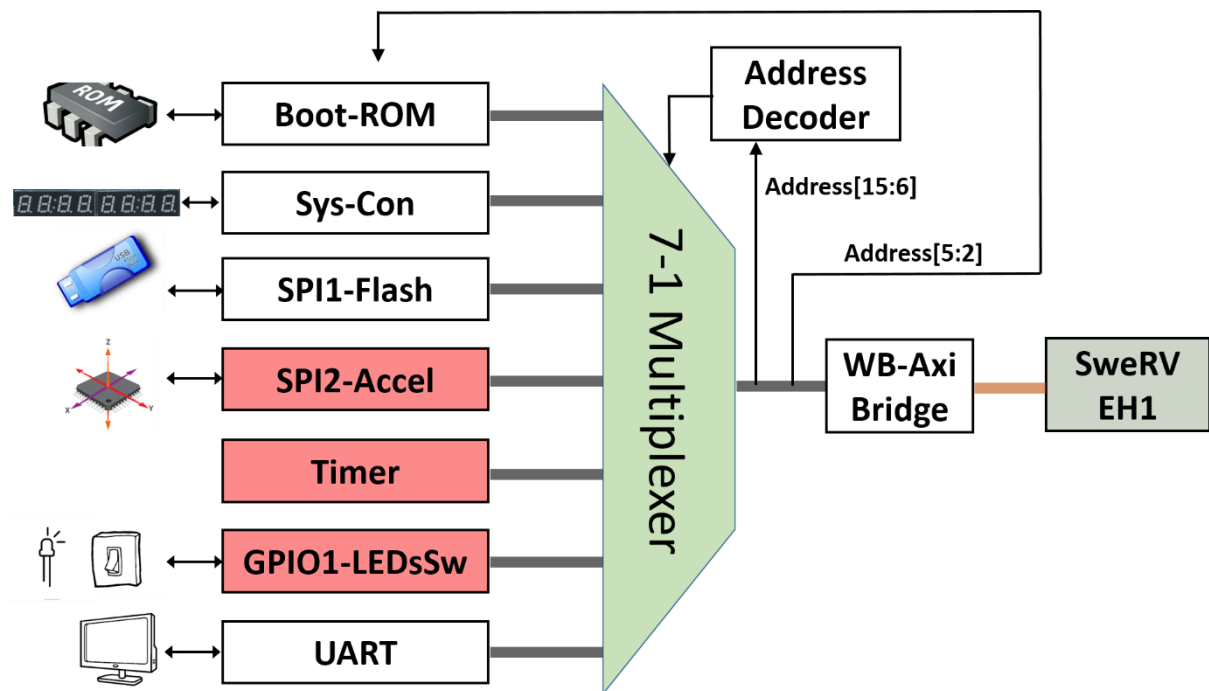


그림 2. RVfpga 시스템의 I/O 시스템

작업: SoC 에서 그림 2 의 각 요소를 찾습니다. 다음 파일 및 디렉토리를 검사해야 합니다.

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v (그림 2 의 요소가 인스턴스화되는 기본 파일).
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh

RVfpga 시작 안내서에 설명된 대로 원본 SweRVolf (<https://github.com/chipsalliance/Cores-SweRVolf>)에는 그림 2 에 표시된 주변 장치 중 일부만 포함되어 있습니다: 특히 부팅 ROM, 시스템 컨트롤러 (7 세그먼트 디스플레이가 없는), SPI 플래시 메모리 및 UART (그림 2 에서 흰색으로 표시). SweRVolf SoC 는 새로운 주변 장치인 SPI 가속도계, 타이머, GPIO 모듈 (그림 2 에서 빨간색으로 표시) 및 7 세그먼트 디스플레이 컨트롤러 (SweRVolf 의 기존 시스템 컨트롤러를 확장)를 이용하여 기존 SweRVolf SoC 를 확장합니다.

각 주변 장치는 프로세서로부터 값을 수신하거나 값을 프로세서로 다시 보냅니다. 메모리 주소는 I/O 값 용으로 예약되어 있으며 레지스터, 메모리 매핑된 I/O 레지스터 또는 장치 컨트롤러 레지스터라고 합니다. 주변 장치로 값을 보내기 위해 CPU 는 지정된 메모리 주소 (즉, 메모리 매핑 레지스터)에 값을 저장합니다. 주변 장치에서 값을 읽기 위해 CPU 는 지정된 메모리 주소에서 값을 로드 합니다. 따라서 CPU 의 간단한 로드/저장 작업으로 장치를 구성하거나 상태를 확인하거나 데이터를 읽고 쓸 수 있습니다.

그림 2 의 멀티플렉서는 주소 [15:6]를 사용하여 요청된 장치 컨트롤러를 선택합니다. 장치 컨트롤러는 주소 [5:2]를 사용하여 장치를 제어하는 데 사용되는 여러 레지스터 중에서 선택합니다.

3. 범용 입력/출력 (GPIO)

범용 I/O (GPIO) 컨트롤러는 프로그래머에게 외부 디지털 핀을 노출합니다. 프로그램에서 주어진 시간에 이러한 핀은 입력 또는 출력으로 구성할 수 있습니다. 이 지정은 핀 단위이며 원하는 경우 프로그램 전체에서 변경할 수 있습니다. GPIO 핀은 LED, 스위치 및 푸시 버튼과 같은 외부 장치에 연결할 수 있습니다.

그림 3 은 하나의 외부 핀을 CPU 에 연결하는 일반 GPIO 모듈의 단순화된 다이어그램을 보여줍니다. 핀은 LED, 스위치 등과 같은 모든 입/출력 장치에 연결할 수 있습니다. 핀은 그림에서 녹색으로 강조 표시된 tri-state 버퍼에 연결됩니다. 이 버퍼를 사용하면 프로그래머가 핀을 입력 또는 출력으로 구성할 수 있습니다. tri-state 버퍼가 활성화되면 핀이 출력 역할을 합니다 (예: LED 구동용). tri-state 버퍼가 비활성화되면 핀이 입력 역할을 합니다 (예: 스위치 값에서 읽기).

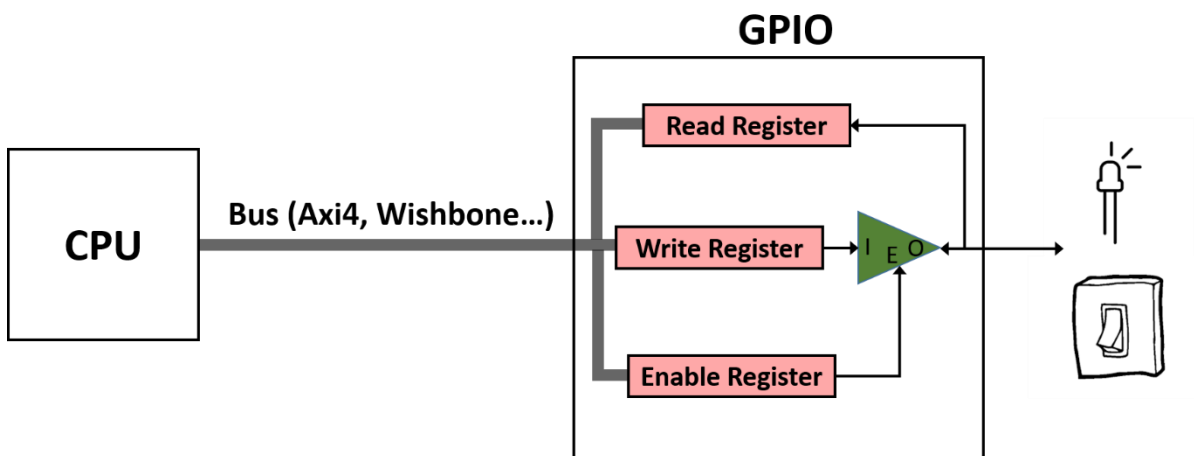


그림 3. GPIO 단순화 회로

tri-state 버퍼는 일반 버퍼로 작동하거나 (활성화된 경우) 부동 출력 (비활성화된 경우)을 가질 수 있습니다. tri-state 버퍼에는 두 개의 입력, E (활성화) 및 I (입력), 하나의 출력, O가 있으며 그 진리표는 표 1에 나와 있습니다. E가 1이면 tri-state는 다음과 같은 일반 버퍼로 작동합니다. 출력 (O)과 입력 (I)이 동일합니다. E가 0이면 입력과 출력 사이에 연결이 없고 출력 (O)이 구동되지 않습니다 (O는 floating). 그림 3에서 핀을 출력으로 구성하기 위해 E는 1이며 CPU가 핀을 구동할 수 있습니다. 핀이 입력으로 설정되면 E는 0이며, 이는 CPU가 핀을 구동하지 못하게 하고 주변 장치가 핀을 구동할 수 있도록 합니다.

표 1. Tri-state 진리표

E	I	O
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

RVfpga 시스템은 메모리 매핑된 I/O를 사용하여 이러한 레지스터에 저장된 값을 읽고 씁니다. 예를 들어, 그림 3의 핀이 스위치에 연결되어 있고 GPIO의 3개의 레지스터가 다음과 같이 매핑되어 있다고 가정합니다.

- Read Register = Address 0x80001400
- Write Register = Address 0x80001404
- Enable Register = Address 0x80001408

스위치 상태를 읽기 위해 다음을 수행합니다.

1. 활성화 레지스터에 0 을 기록하여 핀을 입력으로 설정합니다 (즉, 0x80001408 주소에 0 저장 실행).
2. 0x80001400 주소로 로드 명령을 실행하여 읽기 레지스터를 읽습니다.

4. GPIO 상위 수준 사양

이 섹션에서는 먼저 SweRVolFX's GPIO 의 상위 수준 사양을 분석한 다음 이 주변 장치를 사용하는 한 가지 LAB 을 제안합니다.

A. GPIO 고급 사양

SweRVolFX 에서 사용되는 GPIO 모듈은 OpenCores (<https://opencores.org/projects/gpio>)에서 가져온 것입니다. OpenCores 의 GPIO 모듈 다운로드와 함께 제공되는 gpio_spec.pdf 문서는 모듈의 상위 사양을 설명합니다. 아래 폴더에서 사용할 수 있습니다.

[RVfpgaPath]/RVfpga/src/SweRVolFXSoC/Peripherals/gpio/docs/gpio_spec.pdf, 이 LAB 에서는 GPIO 모듈의 주요 작동 및 기능을 요약합니다. 또한 *gpio_spec.pdf*에서 전체 사양을 얻을 수 있습니다.

GPIO 모듈에는 다음과 같은 주요 기능이 있습니다.

- Wishbone Interconnection 을 사용합니다.
- 주변기기로서만 동작합니다.
- 사용자는 1-32 개의 GPIO 핀을 사용할 수 있습니다.
- 여러 GPIO 모듈 (GPIO 코어라고도 함)을 병렬로 사용하여 32 개 이상의 GPIO 핀에 액세스 할 수 있습니다.
- 모든 GPIO 핀은 다음 기능을 할 수 있습니다:
 - 양방향 (이 경우 외부 양방향 I/O 셀이 필요함).
 - tri-state 또는 오픈 드레인 사용 (이 경우 외부 tri-state 또는 오픈 드레인 I/O 셀이 필요함).
- -입력으로 프로그래밍 된 GPIO 핀:
 - 등록이 가능합니다.
 - CPU 에 인터럽트 요청이 발생할 수 있습니다.

GPIO 코어 사양의 섹션 4 는 GPIO 모듈 내에서 사용 가능한 제어 및 상태 레지스터를 설명합니다. 각 레지스터는 표 2 에 표시된 대로 다른 주소에 할당됩니다. GPIO 레지스터의 기본 주소는 0x80001400 입니다.

표 2. GPIO 레지스터

Name	Address	Width	Access	Description
RGPIO_IN	0x80001400	1-32	R	GPIO 입력 데이터
RGPIO_OUT	0x80001404	1-32	R/W	GPIO 출력 데이터
RGPIO_OE	0x80001408	1-32	R/W	GPIO 출력 드라이버 활성화

RGPIO_INTE	0x8000140C	1-32	R/W	인터럽트 활성화
RGPIO_PTRIG	0x80001410	1-32	R/W	인터럽트를 트리거하는 이벤트 유형
RGPIO_AUX	0x80001414	1-32	R/W	GPIO 출력에 대한 다중 보조 입력
RGPIO_CTRL	0x80001418	2	R/W	제어 레지스터
RGPIO_INTS	0x8000141C	1-32	R/W	인터럽트 상태
RGPIO_ECLK	0x80001420	1-32	R/W	gpio_eclk 가 RGPIO_IN 을 래치하도록 활성화
RGPIO_NEC	0x80001424	1-32	R/W	gpio_eclk 의 활성 엣지 선택

OpenCores 의 GPIO 모듈은 그림 3 에 표시된 단순화된 버전보다 복잡하지만 그림 3 의 읽기 (입력), 쓰기 (출력) 및 활성화에서 세 개의 레지스터를 식별 할 수 있습니다. OpenCores 의 GPIO 모듈에서 이러한 레지스터는 각각 RGPIO_IN, RGPIO_OUT 및 RGPIO_OE 라고하며 주소 0x80001400, 0x80001404 및 0x80001408 에 각각 매핑됩니다.

작업: GPIO 모듈에서 레지스터 RGPIO_IN, RGPIO_OUT 및 RGPIO_OE 의 선언과 해당 주소의 정의를 찾습니다. GPIO 모듈은 다음 폴더에 있습니다. *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/gpio_top.v*.

RGPIO_IN 레지스터는 범용 입력을 래치합니다. RGPIO_OUT 레지스터는 범용 출력을 구동합니다. RGPIO_OE 는 각 I/O 핀을 입력 또는 출력으로 구성합니다. 활성화 비트 (RGPIO_OE 내)가 설정되면 해당 범용 출력 드라이버가 활성화 되므로 핀을 LED 와 같은 출력 주변 장치에 연결할 수 있습니다. 활성화 비트가 지워지면 출력 드라이버가 tri-state 상태 또는 고 임피던스 모드라고도 하는 오픈 드레인에서 작동하므로 핀을 스위치 또는 푸시 버튼과 같은 입력 주변 장치에 연결할 수 있습니다.

RVfpgaNexys 에서 GPIO 모듈의 처음 16 개의 GPIO 핀 (핀 15:0)은 Nexys A7 보드의 16 개 LED 에 연결됩니다. GPIO 컨트롤러의 마지막 16 개의 GPIO 핀 (핀 31:16)은 16 개의 온 보드 스위치에 연결됩니다.

5. 기본 예제

LAB 1. RISC-V 어셈블리 프로그램과 보드에서 사용할 수 있는 16 개의 LED 중 한 쪽에서 다른 쪽으로 반복적으로 이동하는 4 개의 LED 블록을 보여주는 C 프로그램을 작성합니다. 또한 속도와 방향을 제어하는 두 개의 스위치를 포함합니다. Switch [0]은 속도를 변경하고 Switch [1]은 다음과 같이 방향을 변경합니다.

- Switch [0]이 ON (high)이면 점등된 LED 가 빠르게 움직여야 합니다. 그렇지 않으면 켜진 LED 가 천천히 움직여야 합니다. "빠르게"와 "느리게"가 의미하는 바를 정의 할 수 있지만 두 속도 중 하나를 볼 수 있어야 하며 보기만으로 속도 차이를 감지 할 수 있어야 합니다.
- Switch [1]이 ON (high)이면 점등된 LED 가 오른쪽에서 왼쪽으로 반복해서 이동해야 합니다 (맨 왼쪽 LED 에 도달하면 오른쪽에서 다시 시작). 그렇지 않으면 켜진 LED 가 왼쪽에서 오른쪽으로 반복해서 움직여야 합니다.

아래 그림 4 는 LED 와 스위치가 강조 표시된 Nexys A7 보드를 보여줍니다.

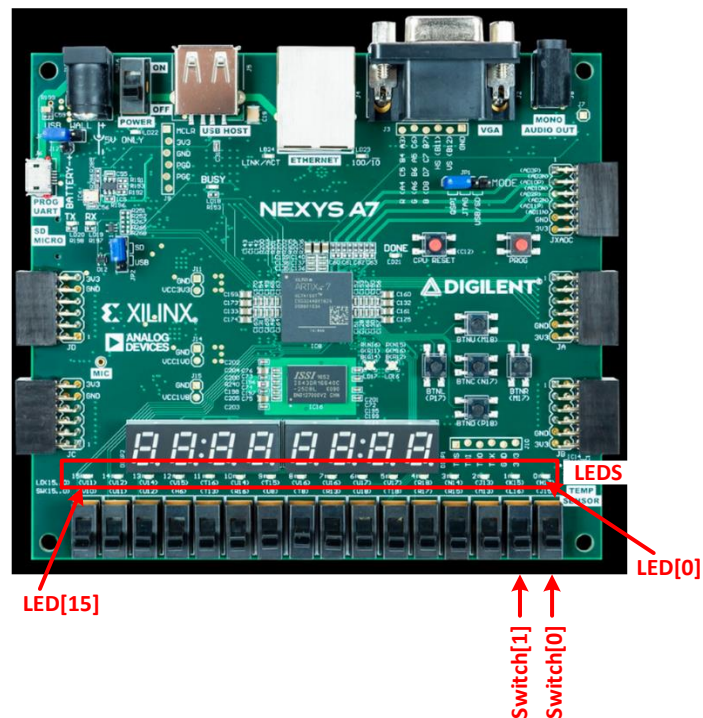


그림 4. Nexys A7 FPGA 보드: LED 및 스위치

힌트: 스위치는 메모리 매핑된 I/O 레지스터의 31:16 핀에 연결되어 있습니다. 따라서 Switch [0]을 읽으려면 RGPIO_OE [16]에 0 을 쓴 다음 RGPIO_IN [16]의 값을 읽어야 합니다. 다른 LED 및 스위치에 액세스하려면 RGPIO_OE 를 적절하게 구성해야 합니다.

6. GPIO 초급 수준의 구현, 시뮬레이션

이 섹션에서는 SweRVofX 에서 사용되는 GPIO 의 초급 수준 세부 정보를 설명합니다. 그런 다음 RVfpgaSim 을 수정하고 Verilator 에서 간단한 어셈블리 예제를 위한 예제 시뮬레이션을 수행합니다. 마지막으로 RVfpgaSim 을 먼저 시뮬레이션한 다음 수정하여 새 GPIO 주변 장치를 추가하고 마지막으로 이 새로운 주변 장치를 사용하는 프로그램을 작성하는 몇 가지 LAB 을 제안합니다.

A. GPIO 초급 수준 구현

이제 메모리 매핑 I/O 를 사용하여 GPIO 핀에 액세스하는 경험이 있으므로 GPIO 의 하위 수준 세부 정보를 살펴 보겠습니다. GPIO 는 그림 5 와 같이 세 가지 주요 부분으로 나눌 수 있습니다. (1) 온 보드 LED/스위치에 대한 RVfpgaNexys 의 외부 연결 (그림 5 의 왼쪽 음영 영역); (2) GPIO 모듈을 SweRVofX SoC 에 통합 (그림 5 의 중간 음영 영역) (3) GPIO 와 SweRV EH1 Core 간의 연결 (그림 5 의 오른쪽 음영 영역).

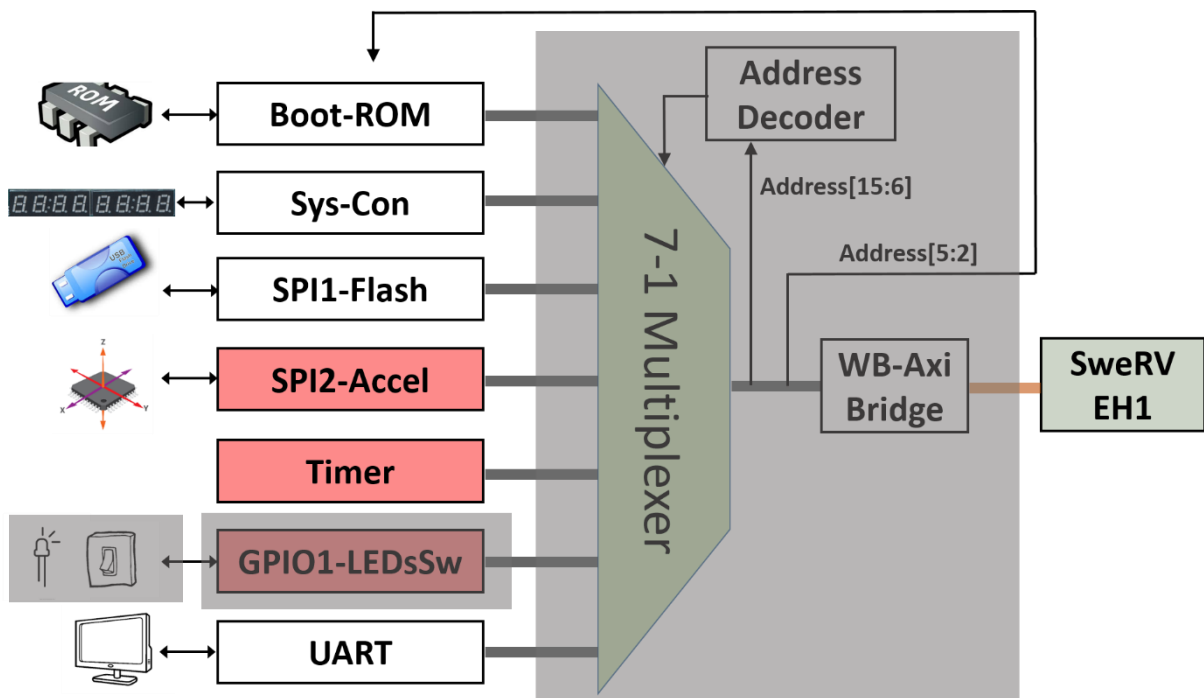


그림 5. 3 단계의 GPIO 분석

i. LED/스위치를 SoC 와 연결

프로젝트의 constraint 파일 (*[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc*)은 입력/출력 SoC 신호와 보드 장치 간의 연결을 정의합니다. 각 보드 장치는 지정된 FPGA 핀과 연결됩니다. 예를 들어, 보드의 가장 오른쪽에 있는 스위치 인 Switch [0]은 PCB의 트레이스를 통해 FPGA 핀 J15에 연결됩니다.

Nexys A7 보드에는 16 개의 LED와 16 개의 스위치가 있습니다. 16 개의 LED를 SoC의 최상위 모듈 (RVfpganexys라고 함, *[RVfpgaPath]/RVfpga/src/rvfpganexys.sv* 파일 내에서 사용 가능)과 연결하는 신호를 *o_led [15:0]*이라고 하고 16 개를 연결하는 신호를 최상위 모듈이 있는 스위치를 *i_sw [15:0]*이라고 합니다. 그림 6은 Xilinx 설계 constraint 파일 (.xdc) 파일인 *rvfpganexys.xdc* (*[RVfpgaPath]/RVfpga/src*에서 사용 가능)의 섹션을 보여줍니다. 여기서 신호와 FPGA 핀 사이의 32 개 연결이 정의되어 있습니다.

```

26 set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { i_sw[0] }]
27 set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { i_sw[1] }]
28 set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[2] }]
29 set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { i_sw[3] }]
30 set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { i_sw[4] }]
31 set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { i_sw[5] }]
32 set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { i_sw[6] }]
33 set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[7] }]
34 set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS18 } [get_ports { i_sw[8] }]
35 set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { i_sw[9] }]
36 set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { i_sw[10] }]
37 set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[11] }]
38 set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMOS33 } [get_ports { i_sw[12] }]
39 set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { i_sw[13] }]
40 set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { i_sw[14] }]
41 set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { i_sw[15] }]
42
43 set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports { o_led[0] }]
44 set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports { o_led[1] }]
45 set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports { o_led[2] }]
46 set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports { o_led[3] }]
47 set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports { o_led[4] }]
48 set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports { o_led[5] }]
49 set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports { o_led[6] }]
50 set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { o_led[7] }]
51 set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMOS33 } [get_ports { o_led[8] }]
52 set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMOS33 } [get_ports { o_led[9] }]
53 set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMOS33 } [get_ports { o_led[10] }]
54 set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMOS33 } [get_ports { o_led[11] }]
55 set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMOS33 } [get_ports { o_led[12] }]
56 set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports { o_led[13] }]
57 set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMOS33 } [get_ports { o_led[14] }]
58 set_property -dict { PACKAGE_PIN V11 IOSTANDARD LVCMOS33 } [get_ports { o_led[15] }]
59

```

그림 6. 온 보드 스위치가 있는 *i_sw* [15:0] 및 온보드 LED 가있는 *o_led* [15:0] 연결 (파일 *rvfpganexys.xdc*).

상단 모듈 (rvfpganexys)의 48-49 행은 SoC (그림 7 의 왼쪽 부분)에 연결된 두 신호를 보여주고, 해당 모듈의 끝은 swervolf_core 모듈 (그림 7 의 오른쪽 부분)과의 연결을 보여줍니다. *i_sw* 및 *o_led* 신호는 swervolf_core 모듈의 GPIO 에 연결된 32 비트 입력/출력 신호인 *io_data* 신호 (257 행)에 병합됩니다 (나중에 그림 8 참조). 또한 *o_led* 신호는 중간 신호 *gpio_out* (라인 266)을 통해 래치됩니다.

```

25 module rvfpganexys
26     #(parameter bootrom_file = "boot_main.mem")
27     (input wire      clk,
28      input wire      rstn,
29      output wire [12:0] ddram_a,
30      output wire [2:0] ddram_ba,
31      output wire      ddram_ras_n,
32      output wire      ddram_cas_n,
33      output wire      ddram_we_n,
34      output wire      ddram_cs_n,
35      output wire [1:0] ddram_dm,
36      inout wire [15:0] ddram_dq,
37      inout wire [1:0] ddram_dqs_p,
38      inout wire [1:0] ddram_dqs_n,
39      output wire      ddram_clk_p,
40      output wire      ddram_clk_n,
41      output wire      ddram_cke,
42      output wire      ddram_odt,
43      output wire      o_flash_cs_n,
44      output wire      o_flash_mosi,
45      input wire      i_flash_miso,
46      input wire      i_uart_rx,
47      output wire      o_uart_tx,
48      inout wire [15:0] i_sw,
49      output reg [15:0] o_led,
50

```

```

256     .i_ram_init_error (litedram init error),
257     .io_data           ((i_sw[15:0],gpio_out[15:0])),
258     .AN (AN),
259     .Digits Bits ({CA,CB,CC,CD,CE,CF,CG}),
260     .o_accel_sclk      (accel_sclk),
261     .o_accel_cs_n      (o_accel_cs_n),
262     .o_accel_mosi      (o_accel_mosi),
263     .i_accel_miso      (i_accel_miso));
264
265     always @(posedge clk core) begin
266         o_led[15:0] <= gpio_out[15:0];
267     end
268

```

그림 7. 상위 모듈 (rvfpganexys.sv)을 사용하여 LED 및 스위치 연결

작업: constraint 파일에서 SweRVolf SoC 모듈 (*io_data*에 병합 됨)로 이 두 신호 (*i_sw* 및 *o_led*)를 따릅니다. 다음 파일을 확인해야 합니다.

```

[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc
[RVfpgaPath]/RVfpga/src/rvfpganexys.sv
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v

```

이전 섹션에서 우리는 RVfpgaNexys 에서 GPIO 모듈의 16 개의 첫 번째 GPIO 핀 (15 ~ 0)이 16 개의 온 보드 LED 에 연결되고 GPIO 컨트롤러의 마지막 GPIO 핀 (31 ~ 16)이 16 개의 온 보드 스위치로 연결되어 있다고 말했습니다. 이것이 이 섹션과 그림 8 에 설명된 구현과 일치합니까?

ii. SoC 에 GPIO 모듈 통합

swervolf_core 모듈 (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v*)의 299-354 행에서 GPIO 모듈이 인스턴스화되고 SoC 에 통합됩니다 (그림 8 참조).

```

299 // GPIO - Leds and Switches
300 wire [31:0] en_gpio;
301 wire      gpio_irq;
302 wire [31:0] i_gpio;
303 wire [31:0] o_gpio;
304
305 bidirec gpio0 (.oe(en_gpio[0]), .inp(o_gpio[0]), .outp(i_gpio[0]), .bidir(io_data[0]));
306 bidirec gpio1 (.oe(en_gpio[1]), .inp(o_gpio[1]), .outp(i_gpio[1]), .bidir(io_data[1]));
307 bidirec gpio2 (.oe(en_gpio[2]), .inp(o_gpio[2]), .outp(i_gpio[2]), .bidir(io_data[2]));
308 bidirec gpio3 (.oe(en_gpio[3]), .inp(o_gpio[3]), .outp(i_gpio[3]), .bidir(io_data[3]));
309 bidirec gpio4 (.oe(en_gpio[4]), .inp(o_gpio[4]), .outp(i_gpio[4]), .bidir(io_data[4]));
310 bidirec gpio5 (.oe(en_gpio[5]), .inp(o_gpio[5]), .outp(i_gpio[5]), .bidir(io_data[5]));
311 bidirec gpio6 (.oe(en_gpio[6]), .inp(o_gpio[6]), .outp(i_gpio[6]), .bidir(io_data[6]));
312 bidirec gpio7 (.oe(en_gpio[7]), .inp(o_gpio[7]), .outp(i_gpio[7]), .bidir(io_data[7]));
313 bidirec gpio8 (.oe(en_gpio[8]), .inp(o_gpio[8]), .outp(i_gpio[8]), .bidir(io_data[8]));
314 bidirec gpio9 (.oe(en_gpio[9]), .inp(o_gpio[9]), .outp(i_gpio[9]), .bidir(io_data[9]));
315 bidirec gpio10 (.oe(en_gpio[10]), .inp(o_gpio[10]), .outp(i_gpio[10]), .bidir(io_data[10]));
316 bidirec gpio11 (.oe(en_gpio[11]), .inp(o_gpio[11]), .outp(i_gpio[11]), .bidir(io_data[11]));
317 bidirec gpio12 (.oe(en_gpio[12]), .inp(o_gpio[12]), .outp(i_gpio[12]), .bidir(io_data[12]));
318 bidirec gpio13 (.oe(en_gpio[13]), .inp(o_gpio[13]), .outp(i_gpio[13]), .bidir(io_data[13]));
319 bidirec gpio14 (.oe(en_gpio[14]), .inp(o_gpio[14]), .outp(i_gpio[14]), .bidir(io_data[14]));
320 bidirec gpio15 (.oe(en_gpio[15]), .inp(o_gpio[15]), .outp(i_gpio[15]), .bidir(io_data[15]));
321 bidirec gpio16 (.oe(en_gpio[16]), .inp(o_gpio[16]), .outp(i_gpio[16]), .bidir(io_data[16]));
322 bidirec gpio17 (.oe(en_gpio[17]), .inp(o_gpio[17]), .outp(i_gpio[17]), .bidir(io_data[17]));
323 bidirec gpio18 (.oe(en_gpio[18]), .inp(o_gpio[18]), .outp(i_gpio[18]), .bidir(io_data[18]));
324 bidirec gpio19 (.oe(en_gpio[19]), .inp(o_gpio[19]), .outp(i_gpio[19]), .bidir(io_data[19]));
325 bidirec gpio20 (.oe(en_gpio[20]), .inp(o_gpio[20]), .outp(i_gpio[20]), .bidir(io_data[20]));
326 bidirec gpio21 (.oe(en_gpio[21]), .inp(o_gpio[21]), .outp(i_gpio[21]), .bidir(io_data[21]));
327 bidirec gpio22 (.oe(en_gpio[22]), .inp(o_gpio[22]), .outp(i_gpio[22]), .bidir(io_data[22]));
328 bidirec gpio23 (.oe(en_gpio[23]), .inp(o_gpio[23]), .outp(i_gpio[23]), .bidir(io_data[23]));
329 bidirec gpio24 (.oe(en_gpio[24]), .inp(o_gpio[24]), .outp(i_gpio[24]), .bidir(io_data[24]));
330 bidirec gpio25 (.oe(en_gpio[25]), .inp(o_gpio[25]), .outp(i_gpio[25]), .bidir(io_data[25]));
331 bidirec gpio26 (.oe(en_gpio[26]), .inp(o_gpio[26]), .outp(i_gpio[26]), .bidir(io_data[26]));
332 bidirec gpio27 (.oe(en_gpio[27]), .inp(o_gpio[27]), .outp(i_gpio[27]), .bidir(io_data[27]));
333 bidirec gpio28 (.oe(en_gpio[28]), .inp(o_gpio[28]), .outp(i_gpio[28]), .bidir(io_data[28]));
334 bidirec gpio29 (.oe(en_gpio[29]), .inp(o_gpio[29]), .outp(i_gpio[29]), .bidir(io_data[29]));
335 bidirec gpio30 (.oe(en_gpio[30]), .inp(o_gpio[30]), .outp(i_gpio[30]), .bidir(io_data[30]));
336 bidirec gpio31 (.oe(en_gpio[31]), .inp(o_gpio[31]), .outp(i_gpio[31]), .bidir(io_data[31]));
337
338 gpio_top gpio_module(
339     .wb_clk_i      (clk),
340     .wb_rst_i      (wb_rst),
341     .wb_cyc_i      (wb_m2s_gpio_cyc),
342     .wb_adr_i      ({2'b0,wb_m2s_gpio_adr[5:2],2'b0}),
343     .wb_dat_i      (wb_m2s_gpio_dat),
344     .wb_sel_i      (4'b1111),
345     .wb_we_i       (wb_m2s_gpio_we),
346     .wb_stb_i      (wb_m2s_gpio_stb),
347     .wb_dat_o      (wb_s2m_gpio_dat),
348     .wb_ack_o      (wb_s2m_gpio_ack),
349     .wb_err_o      (wb_s2m_gpio_err),
350     .wb_inta_o     (gpio_irq),
351     // External GPIO Interface
352     .ext_pad_i     (i_gpio[31:0]),
353     .ext_pad_o     (o_gpio[31:0]),
354     .ext_padoe_o   (en_gpio));
355

```

그림 8. GPIO 모듈 통합 (*swervolf_core.v*).

모듈의 인터페이스는 두 개의 블록으로 나눌 수 있습니다: SweRV EH1 Core 가 컨트롤러/주변 장치 모델을 사용하여 GPIO 와 통신할 수 있는 Wishbone 신호와 (표 3), 외부 I/O 신호 (표 4).

표 3. Wishbone 신호

Port	Width	Direction	Description
wb_cyc_i	1	Inputs	유효한 버스 사이클 표시 (코어 선택)
wb_adr_i	15	Inputs	주소 입력
wb_dat_i	32	Inputs	데이터 입력
wb_dat_o	32	Outputs	데이터 출력
wb_sel_i	4	Inputs	데이터 버스의 유효한 바이트를 나타냅니다 (유효 사이클 동안 0xf 여야 함).
wb_ack_o	1	Output	승인 출력 (정상적인 트랜잭션 종료를 나타냄)
wb_err_o	1	Output	오류 확인 출력 (비정상적인 트랜잭션 종료를 나타냄)
wb_rty_o	1	Output	미사용
wb_we_i	1	Input	high 가 될 때 트랜잭션 쓰기
wb_stb_i	1	Input	유효한 데이터 전송 사이클을 나타냅니다.
wb_inta_o	1	Output	인터럽트 출력

표 4. 외부 I/O 신호

Port	Width	Direction	Description
in_pad_i	1-32	Inputs	GPIO 입력
out_pad_o	1-32	Outputs	GPIO 출력
oen_padoen_o	1-32	Outputs	GPIO 출력 드라이버 활성화 (tri-state 또는 오픈 드레인 드라이버 용)

그림 8 의 342 행에 표시된 것처럼, Wishbone 버스 신호 *wb_m2s_gpio_adr [5:2]*에서 코어가 제공하는 주소의 비트 5:2 는 사용 가능한 메모리 매핑 레지스터 10 개 중 하나를 선택하는 데 사용됩니다. 이 4 비트는 *wb_adr_i* 신호를 통해 GPIO Core 에 제공됩니다 (그림 8 에도 표시됨).

입력 *ext_pad_i*는 GPIO 읽기 레지스터 (RGPIO_IN)에 직접 연결됩니다. 마찬가지로 출력 *ext_pad_o*는 GPIO 쓰기 레지스터 (RGPIO_OUT)에 직접 연결됩니다. 이 두 신호는 32 개의 tri-state 버퍼 모듈 (그림 8, 라인 305-336)을 통해 LED 및 스위치 (*i_gpio*, *o_gpio*, *io_data*)에 연결됩니다. 이렇게 하면 모든 32 핀을 입력 또는 출력으로 구성할 수 있습니다. 우리의 경우 하단 16 개 핀인 15:0 핀이 LED 에 연결되어 있으므로 (그림 7) 출력으로 구성해야 합니다. 상위 16 개 핀 (31:16)은 스위치에 연결되므로 (그림 7) 입력으로 구성해야 합니다. swervolf_core 모듈의 끝부분의 (634-640 행). 다음 모듈을 포함하여 32 개의 tristate 버퍼를 구현합니다.

```
module bidirec (input wire oe, input wire inp, output wire outp, inout wire bidir);
    assign bidir = oe ? inp : 1'bZ ;
    assign outp = bidir;
endmodule
```

TASKS: GPIO 핀 (*io_data*)은 tri-state 버퍼를 통해 GPIO 모듈에 연결됩니다 (그림 8 참조). 활성화 신호의 두 가지 가능한 상태 (*oe* = 0 및 *oe* = 1)에 대해 tri-state 버퍼를 분석합니다.

GPIO 모듈과 온보드 LED/스위치 간의 연결을 고려할 때 프로그래머는 *en_gpio* 에 어떤 값을 할당해야 합니까?

iii. GPIO 와 SweRV EH1 코어 간의 연결

그림 2 에 표시된 것처럼 장치 컨트롤러는 멀티플렉서 및 브리지를 통해 SweRV EH1 Core 에 연결됩니다.

멀티플렉서는 CPU 에서 생성한 주소에 따라 N 개의 가능한 주변 장치 (이 경우 N = 7) 중 하나를 선택합니다.

브리지는 장치 컨트롤러에서 사용하는 Wishbone 신호를 SweRV Core 에서 사용하는 AXI4 신호로 변환하고 그 반대로도 변환합니다 ([RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/AxiToWb/axi2wb.v 파일에서 구현 됨).

7:1 멀티플렉서 (그림 9)는

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v 파일에서 구현되며 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh 파일의 04-205 행에서 인스턴스화 됩니다. 마지막 파일은 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v 위치에 있는 swervolf_core 모듈의 168 행에 포함되어 있습니다.

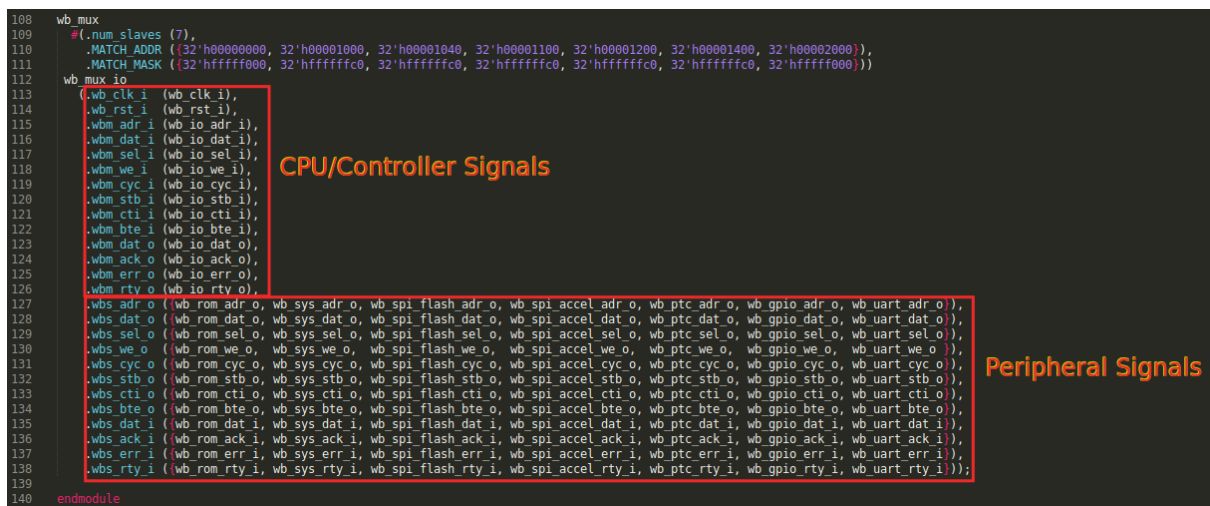


그림 9. 7-1 멀티플렉서는 CPU (wb_intercon.v)에 연결할 주변 장치를 선택합니다.

멀티플렉서는 라인 110-111 의 주소에 따라 CPU (wb_mio_* 신호 – 그림 9 의 115-126 행)를 Wishbone 버스 (그림 9 의 127-138 행)에 연결하여 읽거나 쓸 주변 장치를 선택합니다. 예를 들어 CPU 에서 생성 한 주소가 0x80001400-0x8000143F 범위에 있으면 GPIO 주변 장치가 선택되므로 wb_mio_* 신호는 wb_gpio_* 신호와 연결됩니다.

그림 10 은 멀티플렉서의 Verilog 구현을 보여줍니다

([RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon_1.2.2-r1/wb_mux.v 파일에서 사용 가능).

작업: 멀티플렉서 구현을 자세히 분석하십시오. GPIO 관련 신호 (wb_gpio_*)에 집중할 수 있습니다. 다음 파일을 검사해야 합니다.

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon_1.2.2-r1/wb_mux.v

SoC 의 이 부분을 이해하는 것은 이번 LAB 뿐 아니라 향후 LAB 에서도 중요합니다. 다음 섹션에서 수행되는

시뮬레이션은 멀티플렉서와 관련된 새로운 신호를 추가하여 시뮬레이션을 확장하는 경우 이해하는 데 도움이 될 수 있습니다.

```

82 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
83 // Master/slave connection
84 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
85
86 //Use parameter instead of localparam to work around a bug in Xilinx ISE
87 parameter slave_sel_bits = num_slaves > 1 ? $clog2(num_slaves) : 1;
88
89 reg wbm_err;
90 wire [slave_sel_bits-1:0] slave_sel;
91 wire [num_slaves-1:0] match;
92
93 genvar idx;
94
95 generate
96   for(idx=0; idx<num_slaves ; idx=idx+1) begin : addr_match
97     assign match[idx] = (wbm_adr_i & MATCH_MASK[idx*aw+:aw]) == MATCH_ADDR[idx*aw+:aw];
98   end
99 endgenerate
100
101 //
102 // Find First 1 - Start from MSB and count downwards, returns 0 when no bit set
103 //
104 function [slave_sel_bits-1:0] ff1;
105   input [num_slaves-1:0] in;
106   integer i;
107
108   begin
109     ff1 = 0;
110     for (i = num_slaves-1; i >= 0; i=i-1) begin
111       if (in[i])
112         ff1 = i;
113     end
114   end
115 endfunction
116
117 assign slave_sel = ff1(match);
118
119 always @(posedge wb_clk_i)
120   wbm_err <= wbm_cyc_i & !(match);
121
122 assign wbs_adr_o = {num_slaves{wbm_adr_i}};
123 assign wbs_dat_o = {num_slaves{wbm_dat_i}};
124 assign wbs_sel_o = {num_slaves{wbm_sel_i}};
125 assign wbs_we_o = {num_slaves{wbm_we_i}};
126
127 /* verilator lint_off WIDTH */
128 assign wbs_cyc_o = match & (wbm_cyc_i << slave_sel);
129 /* verilator lint_on WIDTH */
130 assign wbs_stb_o = {num_slaves{wbm_stb_i}};
131
132 assign wbs_cti_o = {num_slaves{wbm_cti_i}};
133 assign wbs_bte_o = {num_slaves{wbm_bte_i}};
134
135 assign wbm_dat_o = wbs_dat_i[slave_sel*dw+:dw];
136 assign wbm_ack_o = wbs_ack_i[slave_sel];
137 assign wbm_err_o = wbs_err_i[slave_sel] | wbm_err;
138 assign wbm_rty_o = wbs_rty_i[slave_sel];
139
140
141
142 endmodule

```

그림 10. Wishbone 멀티플렉서 (파일 *wb_mux.v*).

B. Verilator 시뮬레이션

이 섹션에서는 먼저 새 입력 신호를 추가하여 RVfpgaSim 시뮬레이터를 수정합니다. 그런 다음 Verilator 를 사용하여 RVfpgaSim 을 다시 컴파일하고 시뮬레이터가 간단한 프로그램을 실행할 때 이 새로운 신호를 분석합니다.

i. RVfpgaSIM 수정 및 재 컴파일

시뮬레이션에서는 실제 LED 나 스위치가 없습니다. 따라서 테스트 벤치

(*[RVfpgaPath]/RVfpga/src/rvfpgasim.v*)에서 해당 신호 (*i_sw*)에 상수 값 0xFE34 (그림 11의 왼쪽 부분)를 할당하여 스위치 구동을 시뮬레이션 합니다. 그러면 스위치가 SweRVolfX Core에 대한 입력으로 제공됩니다 (그림 11의 오른쪽 부분).

```
80    wire [15:0] i_sw;
81    assign i_sw = 16'hFE34;    248    .io_data      ({i_sw,16'bz}));
```

그림 11. *rvfpgasim.v*의 SweRVolfX Core에 할당되고 전달된 신호 *i_sw*.

시작 안내서에서 testbench (*rvfpgasim.v*)는 RVfpgaSim (그림 12의 왼쪽 부분)에 대한 입력 신호 (*clk*, *rst* 등)를 수신하고 *swervolf_core* 모듈 (그림 12의 오른쪽 부분)을 인스턴스화 합니다.

```
28    (input wire clk,
29    input wire rst,
30    input wire i_jtag_tck,
31    input wire i_jtag_tms,
32    input wire i_jtag_tdi,
33    input wire i_jtag_trst_n,
34    output wire o_jtag_tdo,
35    output wire o_uart_tx,
36    output wire o_gpio)

190    swervolf_core
191    #(.bootrom_file (bootrom_file))
192    swervolf
193    (.clk (clk),
194    .rstn (!rst),
195    .dmi_reg_rdata (dmi_reg_rdata),
```

그림 12. RVfpgaSim 및 SweRVolfX 인스턴스화를 위한 입력 신호 (파일 *rvfpgasim.v*).

어떤 상황에서는 시뮬레이터에 새로운 입력/출력 신호를 추가할 수 있습니다. 예를 들어, 다음으로 가장 오른쪽 스위치에 대한 값을 제공하는 *i_sw0*이라는 RVfpgaSim에 입력 신호를 포함하는 방법을 설명합니다.

다음 단계를 따르십시오.

1. *[RVfpgaPath]/RVfpga/src/rvfpgasim.v* 파일 수정:

- a. *i_sw0*이라는 새로운 1-bit 입력 신호를 포함합니다. 그림 13을 참조하십시오.

```
28    (input wire clk,
29    input wire rst,
30    input wire i_jtag_tck,
31    input wire i_jtag_tms,
32    input wire i_jtag_tdi,
33    input wire i_jtag_trst_n,
34    output wire o_jtag_tdo,
35    output wire o_uart_tx,
36    output wire o_gpio,
37    input wire i_sw0)
```

그림 13. 새로운 *i_sw0* 입력 신호.

- b. 이 신호를 맨 오른쪽 스위치로 제공하십시오. 나머지 스위치 값을 0xFE34로 할당합니다 (bit 0 제외). 그림 14를 참조하십시오.

```

80
81     wire [15:0] i_sw;
82     // assign i_sw = 16'hFE34;
83     assign i_sw = {15'b111111100011010,i_sw0};
84

```

그림 14. i_sw0 을 맨 오른쪽 스위치로 제공합니다.

2. [RVfpgaPath]/RVfpga/src/verilatorSIM/tb.cpp 파일 수정: Verilator 용 C ++ 메인 파일입니다. 이 파일의 마지막 부분에서 반복적으로 클럭 펄스를 구성하는 while loop(그림 15 참조)를 찾을 수 있습니다. SoC 에 대한 클럭 신호는 각 반복에서 이진 값을 반전하여 (1→0 또는 0→1)이 루프 내에서 생성됩니다 (178 행). 또한 시뮬레이션 시간은 가변 main_time (180 라인)으로 계산되며 나노 초 단위로 측정됩니다 (클럭 사이클은 20ns 이므로 클럭 펄스는 10ns). 마지막으로, 시뮬레이션 시간이 timeout (라인 173-176 의 빨간색 부분) 값에 도달하면 시뮬레이션이 완료됩니다.

```

136     top->clk = 1;
137     top->rst = 1;
138     while (!(done || Verilated::gotFinish())) {
139         if (main_time == 100) {
140             printf("Releasing reset\n");
141             top->rst = 0;
142         }
143         if (main_time == 200)
144             top->i_jtag_trst_n = true;
145
146         top->eval();
147         if (tfp)
148             tfp->dump(main_time);
149         if (baud_rate) do_uart(&uart_context, top->o_uart_tx);
150         if (jtag && (main_time > 300)) {
151             int ret = jtag->doJTAG(main_time/20, //doJtag requires t to only increment by one
152                 &top->i_jtag_tms,
153                 &top->i_jtag_tdi,
154                 &top->i_jtag_tck,
155                 top->o_jtag_tdo);
156             if (ret != VerilatorJtagServer::SUCCESS) {
157                 if (ret == VerilatorJtagServer::CLIENT_DISCONNECTED) {
158                     printf("Ending simulation. Reason: jtag_vpi client disconnected.\n");
159                     done = true;
160                 }
161                 else {
162                     printf("Ending simulation. Reason: jtag_vpi error encountered.\n");
163                     done = true;
164                 }
165             }
166         }
167         if (gpio0 != top->o_gpio) {
168             printf("%lu: gpio0 is %s\n", main_time, top->o_gpio ? "on" : "off");
169             gpio0 = top->o_gpio;
170         }
171         if (timeout && (main_time >= timeout)) {
172             printf("Timeout: Exiting at time %lu\n", main_time);
173             done = true;
174         }
175         top->clk = !top->clk;
176         main_time+=10;
177     }

```

그림 15. 시뮬레이션을위한 While loop

루프에 들어가기 전에 새 i_sw0 신호에 이진 값 0 을 할당하고 (그림 16 의 왼쪽 부분) 루프 내에서 30 us 시간에 1 로 변경합니다 (그림 16 의 오른쪽 부분 참조).

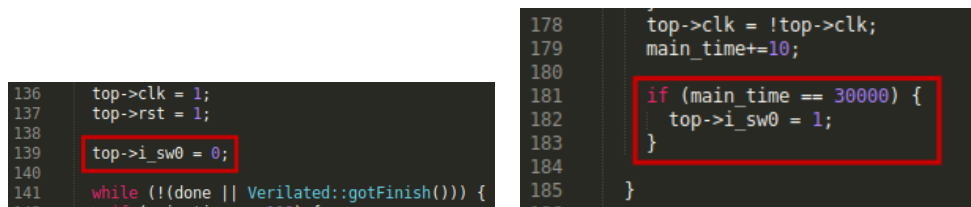


그림 16. 신호 `i_sw0` 에 값 할당.

- 이러한 모든 변경을 수행한 후에는 다음 명령을 실행하여 RVfpgaSim 을 다시 컴파일하십시오 (GSG 에 설명되어 있음).

```
cd [RVfpgaPath]/RVfpga/verilatorSIM
make clean
make
```

새 파일 *Vrvfpgasim* (RVfpgaSim 시뮬레이션 바이너리)은 *[RVfpgaPath]/RVfpga/verilatorSIM* 디렉토리 내에 생성되어야 합니다.

WINDOWS: Cygwin 터미널에서 마지막 단계 (4 단계)를 수행해야 합니다 (자세한 지침은 시작 안내서의 섹션 6 및 부록 C 참조). C: Windows 폴더는 Cygwin (*/cygdrive/c*)에서 찾을 수 있습니다.

MacOS: 자세한 지침은 시작 안내서의 부록 D 를 참조하십시오.

ii. 프로그램 *LedsSwitches.S*의 시뮬레이션 분석

이 섹션에서는 RVfpga 시작 안내서의 *LedsSwitches.S* 예제 프로그램 (그림 17)을 시뮬레이션 합니다. 이 프로그램은 스위치의 값을 읽고 Nexys A7 보드의 LED 에 해당 값을 씁니다. 32 개의 입력/출력 핀이 설정에 따라 입력 또는 출력으로 구성되도록 활성화 레지스터를 구성해야 합니다. 특히 GPIO 의 하위 16 개 핀은 LED 에 연결되므로 CPU 에 대한 출력 핀입니다 (Enable = 1). GPIO 의 상위 16 개 핀은 CPU 에 대한 입력 핀이 스위치에 연결됩니다 (Enable = 0). 스위치는 읽기 레지스터의 상위 16 비트를 차지하므로 값을 LED 에 쓰기 전에 오른쪽으로 이동해야 합니다.

```
#define GPIO_SWS      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408

.globl main
main:

li x28, 0xFFFF
li x29, GPIO_INOUT
sw x28, 0(x29)           # Write the Enable Register

next:
    li a1, GPIO_SWS      # Read the Switches
    lw t0, 0(a1)

    li a0, GPIO_LEDs
    srl t0, t0, 16
    sw t0, 0(a0)         # Write the LEDs
```

```
beq zero, zero, next
.end
```

그림 17. SweRVofx SoC 에서 실행하기 위한 LedsSwitches.s

시뮬레이션을 실행하려면 다음 단계를 따르십시오.

1. 컴퓨터에서 VSCode/PlatformIO 를 엽니다.
2. 상단 표시 줄에서 *File* → *Open Folder ...* (그림 18)를 클릭하고 *[RVfpgaPath]/RVfpga/examples/* 디렉토리로 이동 합니다.

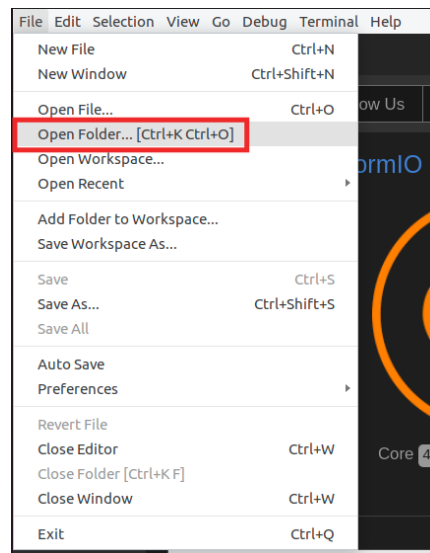


그림 18. LedsSwitches.S 예제 열기

3. *LedsSwitches* 디렉토리를 선택하고 (열지 말고 선택만 하십시오) 확인을 클릭하십시오. 예제는 PlatformIO 에서 열립니다.
4. *platformio.ini* 파일을 열고 위에서 생성한 RVfpgaSim 시뮬레이션 바이너리 (그림 19)의 경로 (이전 섹션의 3 단계)가 올바른지 확인합니다. GSG 에서 다음과 같이 표시되어야 합니다.

```
board_debug.verilator.binary =  
[RVfpgaPath] /RVfpga/verilatorSIM/Vrvfpgasim
```

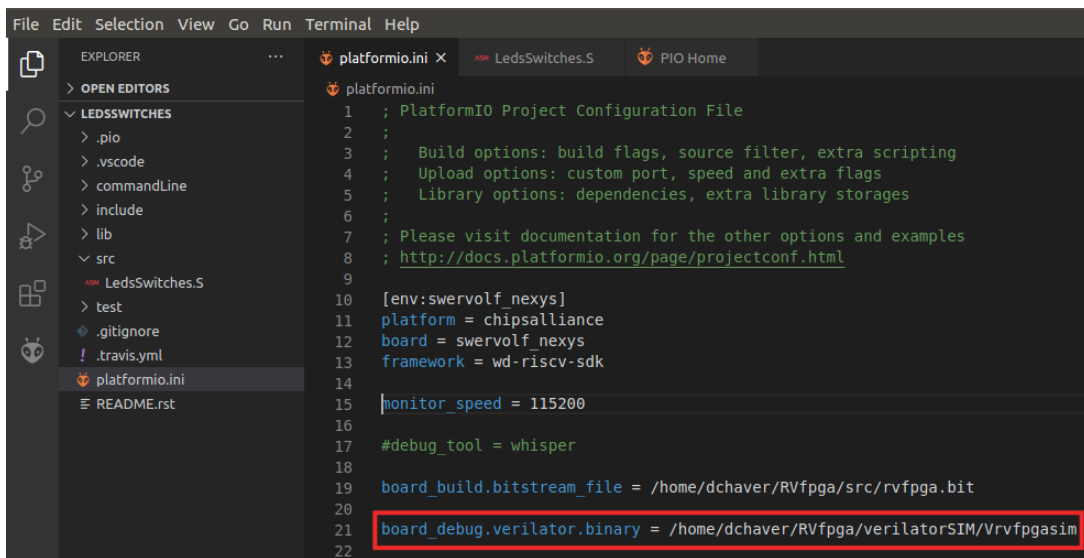



그림 19. Platformio 초기화 파일: platformio.ini

Windows: RVfpgaSim 시뮬레이션 실행 파일을 Vrvfpgasim.exe 라고 합니다.

```
board_debug.verilator.binary = [RVfpgaPath]\RVfpga\verilatorSIM\Vrvfpgasim.exe
```

5. 왼쪽 메뉴 리본에서 PlatformIO 아이콘  을 클릭하여 시뮬레이션을 실행한 다음 Project Tasks → env : swervolf_nexys → Platform 을 확장하고 Generate Trace 를 클릭합니다.
trace.vcd 파일은 [RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys 내에 생성되어 있어야 하며, PlatformIO 터미널에 다음 명령을 입력하여 GTKWave로 열 수 있습니다.

```
gtkwave [RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys/trace.vcd
```

WINDOWS: 다운로드한 gtwave64 폴더에는 bin 폴더 안에 gtwave.exe 라는 애플리케이션이 포함되어 있습니다. 해당 응용 프로그램을 두 번 클릭하여 GTKWave 를 시작합니다. 애플리케이션 상단에서 **File – Open New Tab** 을 클릭하고 [RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys 폴더에 생성된 trace.vcd 파일을 엽니다.

6. 다음 신호를 trace 에 포함합니다 (이러한 각 신호를 찾으려면 모듈 rvfpgasim-swervolf로 이동).
 - 클럭 신호 추가: **clk**
 - GPIO 입력 신호 추가: **i_gpio**
 - GPIO 출력 신호 추가: **o_gpio**

그래프 (그림 20)에서 16 개의 스위치 (**i_gpio** 신호의 최상위 비트 16 개)의 값이 약간 지연된 상태로 16 개의 LED (신호 **o_gpio**의 최하위 비트 16 개)에 복사되는 것을 볼 수 있습니다. 또한 시간 30us 에서 맨 오른쪽 스위치 (0>1)가 변경되고 이로 인해 맨 오른쪽 LED 도 나중에 변경됩니다.

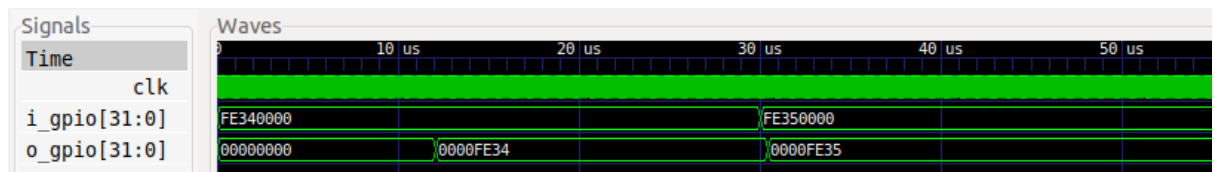


그림 20. LedsSwitches 프로그램 시뮬레이션

7. 고급 연습

연습 2. 이전 섹션의 시뮬레이션을 더 자세히 분석하십시오. 그림 21 은 GPIO 레지스터 (활성화, 읽기 및 쓰기)에 액세스하는 세 가지 명령이 강조 표시된 .elf *LedsSwitches* 프로그램 (그림 17)의 디어셈블리 버전을 보여줍니다. 시작 가이드에서 폴드 내에서 컴파일시 생성되는 *firmware.dis* 파일을 열어 PlatformIO 에서 .elf 프로그램의 디어셈블리 버전을 쉽게 볼 수 있음을 기억하십시오.

[RVfpgaPath] /RVfpga/examples/LedsSwitches/.pio/build/swervolf-nexys/ (그림 21 참조).

```

> PSP
> src
  ≡ .sconsign36.dblite
  ≡ firmware.bin
  ≡ firmware.elf
  ≡ LedsSwitches.map
  ≡ libBoardBSP.a
  ≡ libPSP.a
  ≡ project.checksum
> libdeps
> .vscode
> commandLine
> include
> lib
  ≡ src
65 Disassembly of section .text:
66
67 00000090 <main>:
68 90: 00010e37      lui t3,0x10
69 94: fffe0e13      addi t3,t3,-1 # ffff <_sp+0xcebf>
70 98: 80001eb7      lui t4,0x80001
71 9c: 408e8e93      addi t4,t4,1032 # 80001408 <OVERLAY_END_OF_OVERLAYS+0xa0001408>
72 a0: 01cea023      sw t3,0(t4)
73
74 000000a4 <next>:
75 a4: 800015b7      lui a1,0x80001
76 a8: 40058593      addi a1,a1,1024 # 80001400 <OVERLAY_END_OF_OVERLAYS+0xa0001400>
77 ac: 0005a283      lw t0,0(a1)
78 b0: 80001537      lui a0,0x80001
79 b4: 40450513      addi a0,a0,1028 # 80001404 <OVERLAY_END_OF_OVERLAYS+0xa0001404>
80 b8: 0102d293      srli t0,t0,0x10
81 bc: 00552023      sw t0,0(a0)
82 c0: fe0002e3      beqz zero,a4 <next>

```

그림 21. LedsSwitches.S 프로그램의 디어셈블리 버전

RVfpgaSim 에서 이 프로그램을 시뮬레이션하고 그림 21 (sw, lw 및 sw)에서 빨간색으로 강조 표시된 세 가지 메모리 명령을 각각 실행하는 동안 GPIO 신호를 분석합니다. 이것은 섹션 A 에 설명된 GPIO 하위 수준 구현을 이해하는 데 도움이 됩니다.

섹션 B 의 시뮬레이션에서 시작하여 다음 신호에 대한 값을 추가하고 분석할 수 있습니다 (각 신호를 찾기 위해 참조된 모듈로 이동).

- rvfpgasim → swervolf → swerv_eh1 → swerv → ifu
 - 클럭: *clk*.
 - 가져온 명령어: *ifu_i0_instr* 및 *ifu_i1_instr*.
- rvfpgasim – swervolf
 - 32 비트 입/출력 핀: *i_gpio* 및 *o_gpio*.
 - CPU 에서 제공하는 주소: *wb_m2s_io_adr*.
- rvfpgasim – swervolf – gpio_module
 - GPIO 외부 인터페이스: *ext_pad_i*, *ext_pad_o* 및 *ext_padoe_o*.
- rvfpgasim – swervolf – wb_intercon0

- 그림 2의 멀티플렉서에 대한 출력 주소 및 데이터 신호: ***wb_io_adr_i, wb_io_dat_i, wb_io_dat_o.***
- 그림 2: ***wb_gpio_adr_i, wb_gpio_dat_i, wb_gpio_dat_o***의 멀티플렉서에 대한 입력 GPIO 데이터 신호.
- 그림 2의 멀티플렉서에 대한 Selection 신호: ***wb_*_cyc_o.***
- rvfpgasim – swervolf – wb_intercon0 – wb_mux_io
 - 그림 2의 멀티플렉서에 대한 match 신호: ***match***
- rvfpgasim – swervolf – swerv_eh1 – swerv – dec – arf – gpr_banks(0) – gpr(5) – gprff
 - t0에 대한 레지스터 값: ***dout.***

연습 3. 5개의 온 보드 푸시 버튼을 지원하도록 **RVfpgaNexys**를 확장합니다. 푸시 버튼은 그림 22에 나와 있습니다. 5개의 버튼은 위치에 따라 이름이 지정됩니다. 위, 아래, 왼쪽, 오른쪽 및 가운데 – BTNU, BTND, BTNL, BTNR, BTNC.

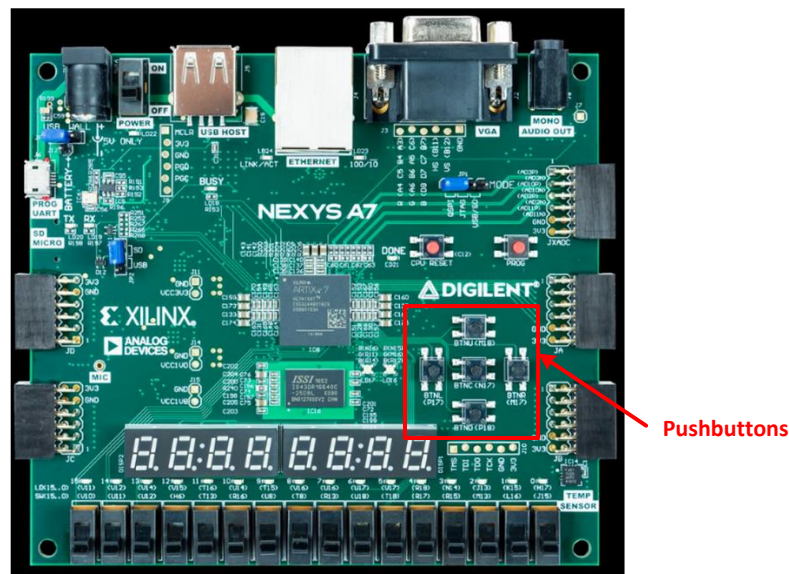


그림 22. Nexys A7 FPGA 보드의 푸시 버튼

- a. 우리가 사용하는 GPIO 모듈의 최대 크기 (gpio_top)는 우리가 가지고있는 I/O 핀의 수인 32 (LED 16개 + 스위치 16개)라는 점을 감안할 때 SweRVolfX에 있는 GPIO 모듈의 또 다른 인스턴스화를 (5개의 새로운 tri-state 버퍼 및 필요한 모든 신호) 포함할 필요가 있습니다.
- b. 새 GPIO 컨트롤러에 의해 노출된 레지스터 매핑을 위해 0x80001800 (사용 가능한)에서 시작하는 주소를 사용합니다. 새 주변 장치를 포함하려면 멀티플렉서 (그림 9)를 수정해야 합니다.
- c. 또한 5개의 푸시 버튼이 다음 FPGA 핀에 연결되어 있다는 점을 고려하여 제약 조건 파일을 수정해야 합니다.
 - i. BTNC는 PIN N17에 연결됩니다.
 - ii. BTNU는 PIN M18에 연결됩니다.
 - iii. BTNL은 PIN P17에 연결됩니다.
 - iv. BTNR은 PIN M17에 연결됩니다.
 - v. BTND는 PIN P18에 연결됩니다.

연습 4. 5 개의 온보드 푸시 버튼에 대해 **RVfpgaNexys** 에서 다른 컨트롤러를 설계합니다.

- a. 연습 3 과 달리 이번의 경우에는 그림 3 에 있는 체계에 따라 Verilog 또는 SystemVerilog 에서 자체 GPIO 컨트롤러를 구현해야 합니다. 실제로 해당 회로를 단순화하고 **읽기 레지스터**만 포함할 수도 있습니다. (즉, tri-state 버퍼 또는 **쓰기 레지스터**를 필요가 없음)
- b. 푸시 버튼은 GPIO 컨트롤러에서 사용하지 않는 주소에 매핑할 수 있으므로 이전 연습에서 컨트롤러를 제거할 필요가 없습니다.
- c. 시스템 컨트롤러 주변 장치에 새 컨트롤러를 포함합니다. 사용되지 않는 주소 범위 0x8000101C-0x8000101F 를 사용할 수 있습니다. 시스템 컨트롤러에 포함된 레지스터는 CPU 가 생성한 주소 (i_wb_adr)를 기반으로 Wishbone 버스 (o_wb_rdt)의 데이터 신호에 직접 연결하여 CPU 로 읽어 들입니다. 진행 방법을 이해하는 데 도움이 되도록 **swervolf_syscon** 모듈
(*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v*)의 234-266 행을 확인하십시오.

연습 5. 1 부터 시작하여 점점 증가하는 이진 카운트를 LED 에 표시하는 C 프로그램과 RISC-V 어셈블리 프로그램을 작성합니다. 사용자가 직접 눈으로 확인할 수 있도록 각 증가된 값을 표시하는 사이에 지연을(간격) 시키는 빈 루프를 포함하여야 합니다. 연습 3 에서 구현된 OpenCores 주변 장치를 통해 BTNC 를 읽고 이를 사용하여 카운트 속도를 변경하고, 연습 4 에서 구현된 임시 주변 장치를 통해 BTNU 를 읽고, 이를 누를 때마다 카운트를 다시 시작하는 데 사용합니다.