

**THE IMAGINATION UNIVERSITY PROGRAMME**

**RVfpga-SoC Lab 2**

**RVfpga-SoC에서 소프트웨어 실행**

**표 1. RVfpga 용어**

|  |  |
| --- | --- |
| **Name** | **Description** |
| **Courses** | |
| **RVfpga** | RVfpgaNexys 및 RVfpgaSim, RISC-V SoC(System-on-Chip)를 사용하여 프로그램을 실행하고 주변 장치를 추가하여 시스템을 확장하고(RVfpga Labs 1-10), 시뮬레이션, 성능 측정, 명령 추가 및 메모리 시스템 수정(RVfpga Labs 11-20)을 실행하여 코어 및 메모리 시스템을 탐색하는 방법을 보여주는 과정입니다. 과정 전반에 걸쳐 사용자는 RISC-V 툴체인(컴파일러 및 디버거) 및 시뮬레이터, Verilator HDL 시뮬레이터 및 Western Digital의 Whisper 명령어 세트 시뮬레이터(ISS)를 사용하는 방법도 보여줍니다. |
| **RVfpga-SoC** | SweRV 코어, 메모리 및 주변 장치와 같은 빌딩 블록을 사용하여 처음부터 SweRVolfX SoC의 하위 집합을 구축하는 방법을 보여주는 과정입니다. 이 과정은 또한 Zephyr 실시간 운영 체제(RTOS)를 SweRVolf에 로드하고 운영 체제 상단에서 Tensorflow Lite의 hello world 예제를 포함한 프로그램을 실행하는 방법을 보여줍니다. |
| **Cores and SoCs** | |
| **SweRV EH1 Core** | Western Digital에서 개발한 오픈 소스 상용 RISC-V 코어 (<https://github.com/chipsalliance/Cores-SweRV>). |
| **SweRV EH1 Core Complex** | 추가된 메모리(ICCM, DCCM 및 명령 캐시), 프로그래밍 가능한 인터럽트 컨트롤러(PIC), 버스 인터페이스 및 디버그 장치가 있는 SweRV EH1 코어 (<https://github.com/chipsalliance/Cores-SweRV>). |
| **SweRVolfX** | RVfpga 과정에서 사용하는 System on Chip으로 SweRVolf의 확장입니다.  **SweRVolf** (<https://github.com/chipsalliance/Cores-SweRVolf>): SweRV EH1 Core Complex를 기반으로 구축된 오픈 소스 SoC입니다. boot ROM, UART 인터페이스, 시스템 컨트롤러, 상호 연결(AXI Interconnect, Wishbone Interconnect 및 AXI-to-Wishbone 브리지) 및 SPI 컨트롤러를 추가합니다.  **SweRVolfX:** SweRVolf에 GPIO, PTC, 추가 SPI 및 8자리 7-세그먼트 디스플레이 컨트롤러, 4가지 새로운 주변 장치를 추가합니다. |
| **RVfpgaNexys** | Nexys A7 보드 및 주변 장치를 대상으로 하는 SweRVolfX SoC입니다. DDR2 **인터페이스, CDC**(클럭 도메인 교차) 장치, BSCAN 로직(JTAG 인터페이스용) 및 클럭 생성기를 추가합니다.  RVfpgaNexys는 SweRVolf Nexys(<https://github.com/chipsalliance/Cores-SweRVolf>)와 동일하지만 SweRVolf Nexys 는 SweRVolf를 기반으로 합니다. |
| **RVfpgaSim** | 시뮬레이션을 위한 테스트 벤치 래퍼(wrapper) 및 AXI 메모리가 있는 SweRVolfX SoC 입니다.  RVfpgaSim은 SweRVolf Sim(<https://github.com/chipsalliance/Cores-SweRVolf>)과 동일하지만 SweRVolf Sim 는 SweRVolf를 기반으로 합니다. |

# 소개

이 실습에서는 Vivado Block 디자인 도구를 사용하여 실습1 (Lab 1) 에서 만든 SweRVolfX 하위 집합에서 C 또는 어셈블리 언어로 작성된 프로그램을 실행하는 방법을 보여줍니다. Verilator를 사용하여 설계를 시뮬레이션하거나 Nexys A7 보드에서 설계를 실행할 수 있습니다. FPGA 보드에 액세스할 수 없는 경우 Verilator를 사용한 시뮬레이션에서만 이 실습을 완료할 수 있습니다.

이 실습을 완료하려면 Block Design의 "**BD.v**" Verilog 파일과 실습 1에서 Vivado의 Block Design을 사용하여 생성한 "**rvfpga.bit**" 비트 파일을 사용합니다.

이 실습에서는 나중에 예제 프로그램의 시뮬레이션 추적(trace)을 만드는 데 사용할 **RVfpgaSim**용 시뮬레이션 바이너리를 생성하는 방법을 보여줍니다. 또한 GTKWave를 사용하여 시뮬레이션 추적을 분석합니다.

선택적 단계로 실습1에서 생성한 비트스트림으로 정의된 **RVfpgaNexys**를 PlatformIO를 사용하여 Nexys A7 보드에 다운로드한 다음, PlatformIO를 사용하여 예제 프로그램을 디버깅하는 방법을 보여줍니다. 이 단계는 선택 사항이지만 학습하길 권장합니다.

# 요구사항

이 실습을 완료하려면 다음 도구를 설치해야 합니다.

* VSCode (설치 가이드(06페이지) 참조)
* PlatformIO (설치 가이드(06페이지) 참조)
* GTKWave (설치 가이드(08페이지) 참조)
* Verilator (설치 가이드(08페이지) 참조)
* Cygwin (For Windows User only) (설치 가이드(13페이지) 참조)

**중요:** RVfpga-SoC Lab을 시작하기 전에 RVfpga-SoC 설치 가이드를 완료하는 것이 좋습니다.

예를 들어, 아직 설치하지 않은 경우 RVfpga-SoC 설치 가이드의 지침에 따라 VScode 및 Verilator를 설치합니다. Imagination의 대학 프로그램 (IUP)에서 다운로드한 RVfpga-SoC 폴더를 컴퓨터에 복사했는지 확인합니다.

# 블록 디자인에서 생성된 SoC 실행

첫 번째 실습에서는 Vivado의 블록 설계 도구를 사용하여 프로세서 코어, 상호 연결 및 주변 장치를 서로 연결하여 SweRVolfX SoC의 하위 집합을 만들었습니다. 그런 다음 블록 디자인은 해당 블록 디자인 모듈 전체의 Verilog 파일을 생성합니다. 우리의 경우 "**BD.v**'' 파일이었습니다.

이제 Block Design의 SoC를 실행할 수 있는 두 가지 옵션과 경로가 있습니다.

* Nexys A7 100T 보드에서 Block Design의 SoC 실행.
* Verilator 시뮬레이터에서 Block Design의 SoC 실행.

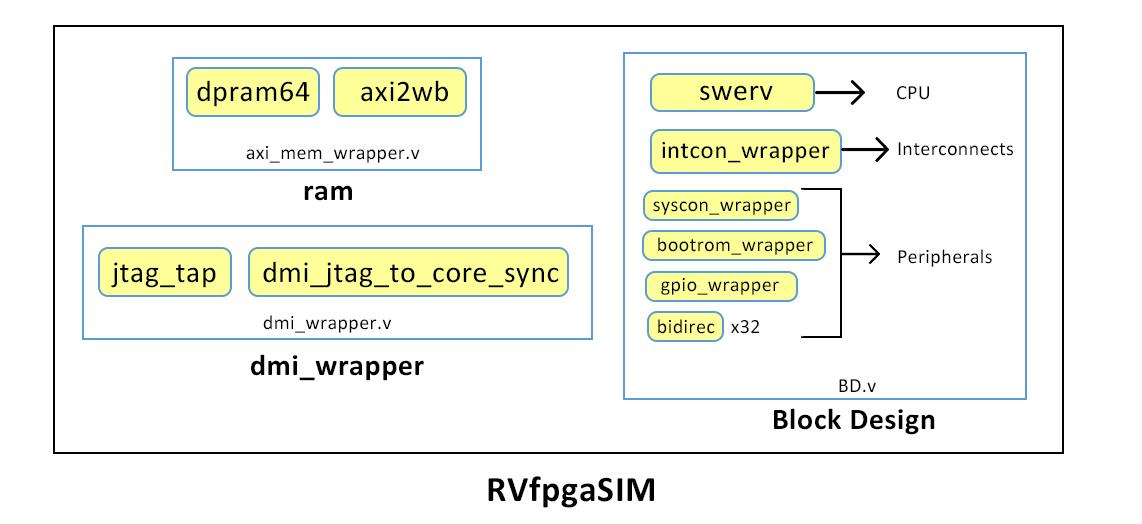
Block Design의 SoC에는 RVfpgaSim(rvfpgasim) 및 RVfpga Nexys(rvfpga)라는 두 가지 최상위 모듈이 아래에 설명되어 있습니다.

1. **RVfpgaSim** (rvfpgasim.v)

RVfpgaSim 모듈은 **시뮬레이션용** RVfpga 시스템의 최상위 모듈로 사용됩니다. 우리는 RVfpga 시스템을 **시뮬레이션**하기 위해 Verilator (RVfpga를 정의하는 Verilog를 시뮬레이션하는 HDL, Hardware Description Language, 시뮬레이터)를 사용합니다.

시뮬레이션에서 SoC를 실행하면 시스템의 내부 신호를 심층적으로 분석할 수 있습니다. 이 실습의 뒷부분에서 RVfpgaSim용 시뮬레이션 바이너리를 생성할 때 "**rvfpgasim.v**"를 최상위 모듈 파일로 사용할 것입니다.

상단 모듈 "**rvfpgasim**" 구조는 그림 1에 나와 있습니다.



**그림 1. RVfpgaSim**

여기에는 세 가지 모듈이 포함됩니다:

* Block Design (BD.v)

Vivado의 블록 디자인을 사용하여 만든 SoC 모듈입니다.

* ram (axi\_mem\_wrapper.v)

메모리 모듈입니다.

* dmi\_wrapper (dmi\_wrapper.v)

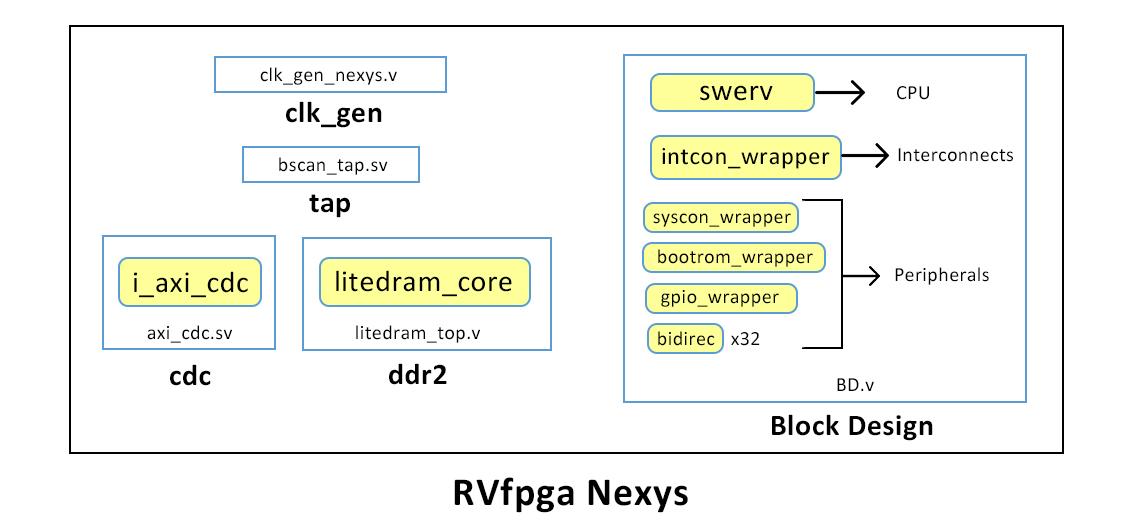
디버깅 모듈 인터페이스입니다.

Lab 1에서는 Vivado의 Block Design을 사용하여 핀을 연결하면서 여러 개의 외부 핀을 연결했습니다. "**Block Design**" 모듈의 이러한 외부 연결은 상위 모듈 "**rvfpgasim**"에서 다른 모듈과 연결됩니다. 예를 들어, “**Block Design**” 모듈의 “**DMI**” 외부 연결은 “**dmi\_wrapper**” 모듈과 연결되고, “**Block Design**” 모듈의 “**RAM**” 외부 연결은 “**ram**” 모듈과 연결됩니다.

1. **RVfpga Nexys** (rvfpga.sv)

RVfpga Nexys 모듈은 Digilent Nexys A7 보드(또는 이전 Nexys 4 DDR 보드)를 대상으로 하는 하드웨어(온보드 구현)용 RVfpga 시스템의 최상위 모듈로 사용됩니다.

상단 모듈 "rvfpga.sv" 구조는 그림 2에 나와 있습니다.



**그림 2. RVfpga Nexys**

RVfpga Nexus에는 다음과 같은 5가지 모듈이 있습니다:

* Block Design(BD.v)

우리가 Block Design을 사용하여 만든 SoC 모듈입니다.

* ddr2 (litedram\_top.v)

DDR 메모리 컨트롤러 모듈입니다.

* clk\_gen (clk\_gen\_nexys.v)

클럭 생성기 모듈입니다.

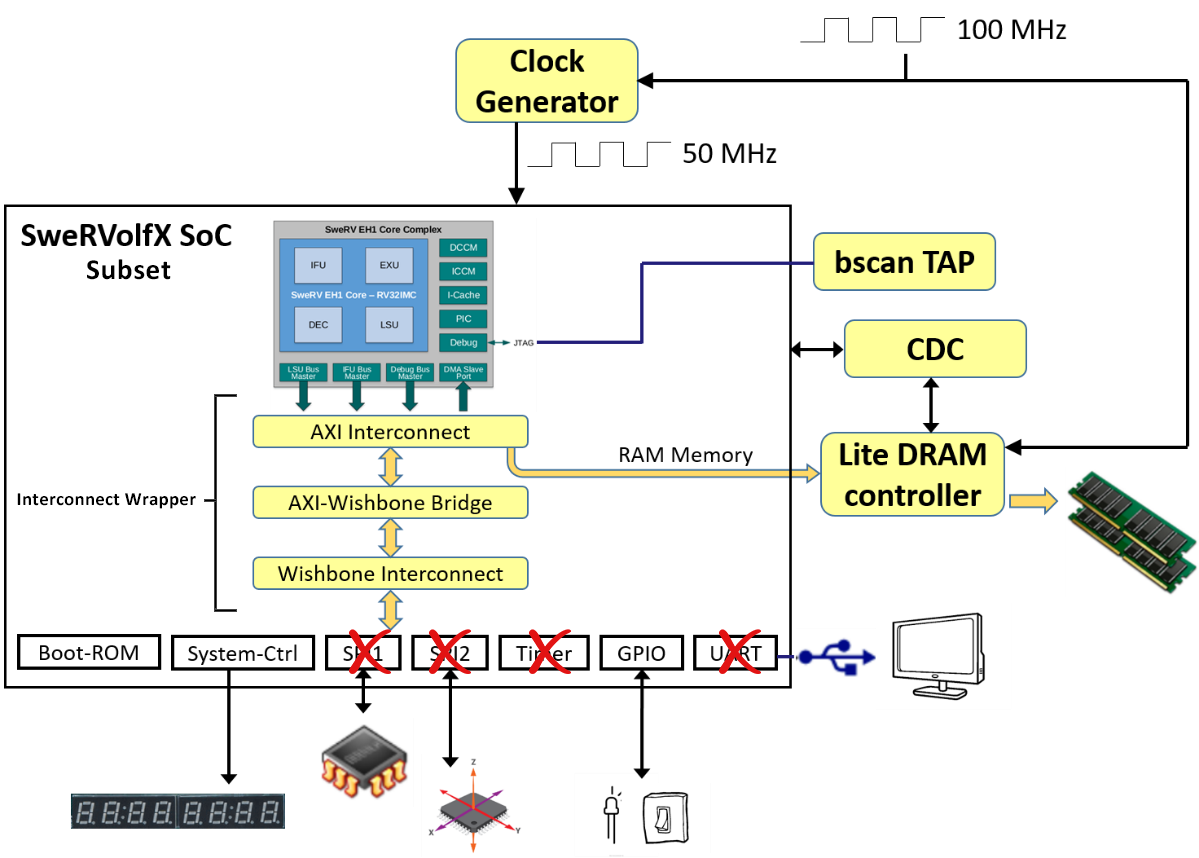
* tap (bscan\_tap.sv)

jtag 디버그 모듈입니다. 자세한 내용은 이 링크를 참조하십시오. [link](https://github.com/chipsalliance/Cores-SweRVolf/issues/29)

* cdc (axi\_cdc.sv)

클럭 도메인 크로싱 모듈입니다.

"**rvfpga.sv**" 상단 모듈에서 "**Block Design**" 모듈의 "**RAM**" 외부 연결은 "**ddr2**" 모듈과 연결됩니다. "**Block Design**"의 "**DMI**" 외부 연결은 "**bscan\_tap.sv**" 모듈과 연결됩니다. "**clk**" 외부 연결은 "**clk\_gen**" 모듈과 연결됩니다(그림 3 참조).



**그림 3. RVfpga Nexys(SweRVolfX SoC 하위 집합)**

# Verilator에서 프로그램 실행

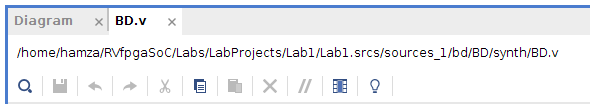
이 섹션에서는 Verilator를 사용하여 **RVfpgaSim**에서 첫 번째 프로그램(*AL\_Operations*)을 실행하는 프로세스를 안내합니다.

|  |
| --- |
| **참고:** 모든 RVfpga Verilog 소스 모듈은 블록 디자인 생성 "BD.v" 파일과 함께 작동하기 위해 "BD\_" 접두사를 붙여야 합니다. 이 "BD.v" 파일은 Verilator에서 시뮬레이션을 위해 최상위 모듈인 "rvfpgasim"에서 사용됩니다. 소스 모듈은 "BD.v" 파일에서 인스턴스화되며 사용된 모든 모듈에 "BD\_" 접두사가 붙도록 요구합니다. 다음 경로에 있는 별도의 "src" 폴더에서 이미 수행했습니다. [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/src. |

먼저 "**Block Design**" 모듈 파일인 **BD.v**를 최상위 모듈 파일 "rvfpgasim.v"를 포함하여 다른 모든 소스 파일이 포함된 폴더로 이동해야 합니다.

이전 실습에서 HDL 래퍼를 만들 때 전체 경로가 제공되었습니다(그림 4 참조). 이 경로를 탐색한 다음 파일을 복사해야 합니다.

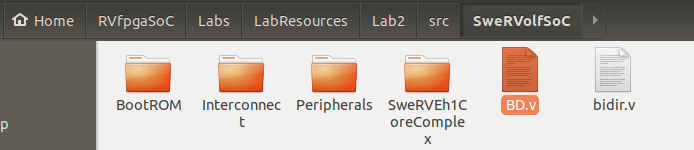
[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabProjects/Lab1/Lab1.srcs/sources\_1/bd/BD/synth/BD.v

****

**그림 4. “Block Design” 모듈의 “BD.v” Verilog 파일 경로**

**1단계.** (그림 4)에 주어진 경로에서 "BD.v" 파일을 복사하고 다음 경로에 "**BD.v**" 파일을 붙여 넣습니다(그림 5 참조).

[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/src/SweRVolfSoC/

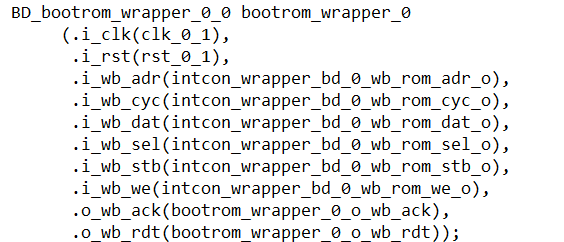
**

**그림 5. "BD.v" "SweRVolfSoC" 디렉토리에 붙여 넣기**

**2단계.** "BD.v" 파일을 열고 다음 모듈 이름이 **"\_0\_0"**으로 끝나는지 확인합니다(그림 6 참조).

* BD\_bootrom\_wrapper\_0\_0
* BD\_gpio\_wrapper\_0\_0
* BD\_intcon\_wrapper\_bd\_0\_0
* BD\_swerv\_wrapper\_verilog\_0\_0
* BD\_syscon\_wrapper\_0\_0

|  |
| --- |
| **참고:** 이것은 시뮬레이션을 위한 모듈 이름의 일관성을 유지하기 위해 수행됩니다. 일치하지 않으면 다음 단계에서 RvfpgaSim용 시뮬레이션 바이너리를 생성하는 동안 오류가 발생합니다. |



**그림 6. “BD.v”**

이제 블록 디자인의 SoC에서 *AL\_Operations* 프로그램을 실행하는 프로세스를 시작하겠습니다.

먼저 PlatformIO를 사용하여 시뮬레이션 추적을 생성한 다음 슈퍼스칼라 프로세서의 양방향 명령인 클럭을 추가하고, 레지스터 x28 (즉, 레지스터 t3) 신호를 시뮬레이션 파형에 추가하고, GTKWave로 명령 및 레지스터 신호가 프로그램이 실행되면서 변경되는 것을 확인합니다.

이렇게 하려면 아래 단계를 완료하십시오.

**3단계. RvfpgaSim용 시뮬레이션 바이너리 생성**

*[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM* 디렉토리에는 RVfpgaSim용 시뮬레이터 바이너리를 생성하기 위한 *script* (*swervolf\_0.7.vc*) 와 *Makefile* 이 있습니다. *script* 에는 Verilator가 무엇보다도 SoC의 소스를 찾을 수 있는 위치를 알 수 있는 정보가 포함되어 있습니다. 이 경우 *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/src* 에서 확인할 수 있습니다.

다음으로, 나중에 RVfpga에서 실행되는 프로그램 *AL-Operations*의 시뮬레이션 추적을 만드는 데 사용할 RVfpgaSim용 바이너리를 생성합니다.

터미널 창에서 다음 명령을 실행하여 시뮬레이터 바이너리를 생성합니다.

* cd [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM
* make clean
* make

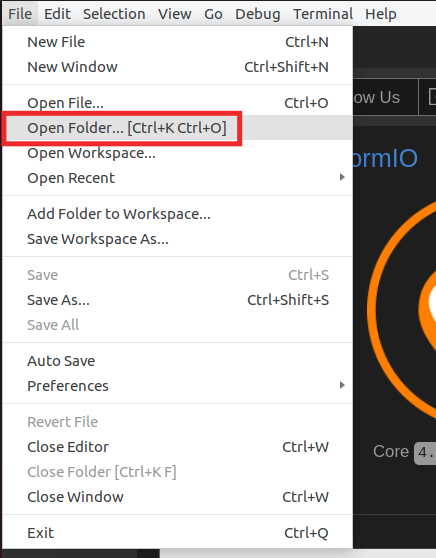
파일 ***Vrvfpgasim***(RVfpga 시뮬레이션 바이너리)은 *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM* 디렉토리 내에 생성되어야 합니다.

**Windows:** Windows를 사용하는 경우 Cygwin 터미널 내에서 이와 동일한 단계를 수행해야 합니다(자세한 지침은 RVfpga-SoC 시작 안내서 부록 B 참조). *C:* Windows 폴더는 Cygwin의 */cygdrive/c*에서 찾을 수 있습니다. 이 섹션의 다른 모든 지침은 Linux에 대해 설명된 지침과 동일합니다.

**4단계. PlatformIO에서 시뮬레이션 추적 생성**

시뮬레이터 바이너리(*Vrvfpgasim*)가 생성되면 AL\_Operations 프로그램의 시뮬레이션 추적(*trace.vcd*)을 생성하기 위해 PlatformIO 내에서 이를 사용할 것입니다.

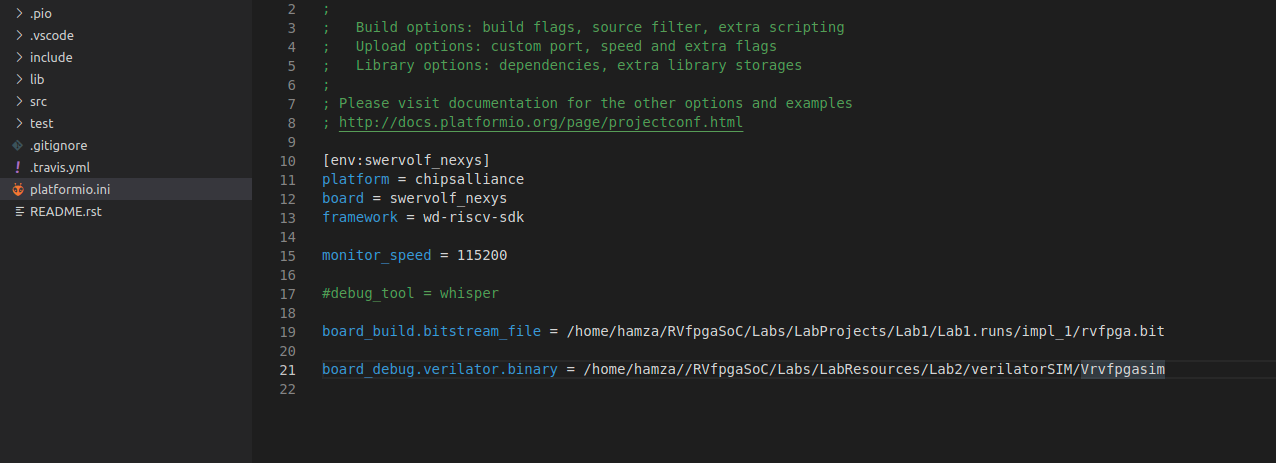
1. 컴퓨터에서 VSCode를 연 다음 PlatformIO를 엽니다.
2. 상단 표시줄에서 *파일→폴더 열기*(그림 7)를 클릭하고 *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/* 디렉토리로 이동합니다.



**그림 7. AL\_Operations.S 예제 열기**

1. *AL\_Operations* 디렉토리를 선택하고(열지 말고 선택하기만 하면 됨) OK를 클릭합니다. 예제는 PlatformIO에서 열립니다.
2. *platformio.ini* 파일을 엽니다. 다음 줄을 편집하여 첫 번째 단계(*Vrvfpgasim*)에서 생성된 RVfpga 시뮬레이션 바이너리의 경로를 설정합니다(그림 8 참조).

board\_debug.verilator.binary = [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM/Vrvfpgasim

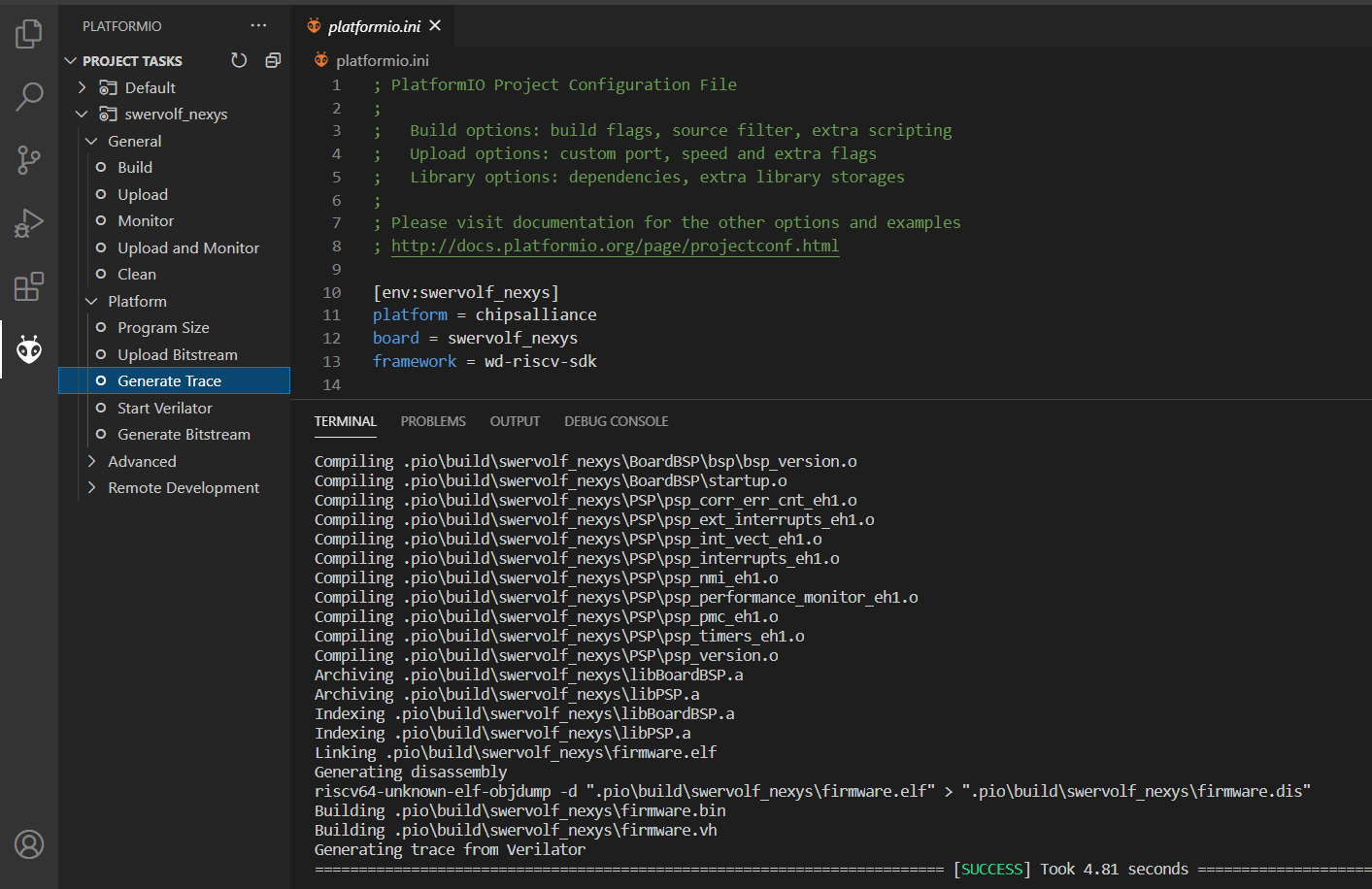
****

**그림 8. PlatformIO 초기화 파일: platformio.ini**

**Windows:** Windows에서 RVfpga 시뮬레이션 실행 파일을 Vrvfpgasim.exe. 라고 합니다. (아래 참조)

board\_debug.verilator.binary = [RVfpgaSoCPath]\RVfpgaSoC\Labs\LabResources\Lab2\verilatorSIM\Vrvfpgasim.exe

1. 왼쪽 메뉴 리본에서 PlatformIO 아이콘을 클릭하여 시뮬레이션을 실행한 다음, Project Task → env:swervolf\_nexys → Paltform을 확장하고 그림 9와 같이 추적 생성(Generate Trace)을 클릭합니다.



**그림 9. Verilator에서 추적 생성**

대안으로 PlatformIO 터미널 창에서 추적을 생성할 수 있습니다. 이를 위해 PlatformIO 창 하단에 있는 버튼**** (PlatformIO: New Terminal 버튼)을 클릭하여 새 터미널 창을 연 후 PlatformIO 터미널에 다음 명령을 입력(또는 복사)합니다.

pio run --target generate\_trace

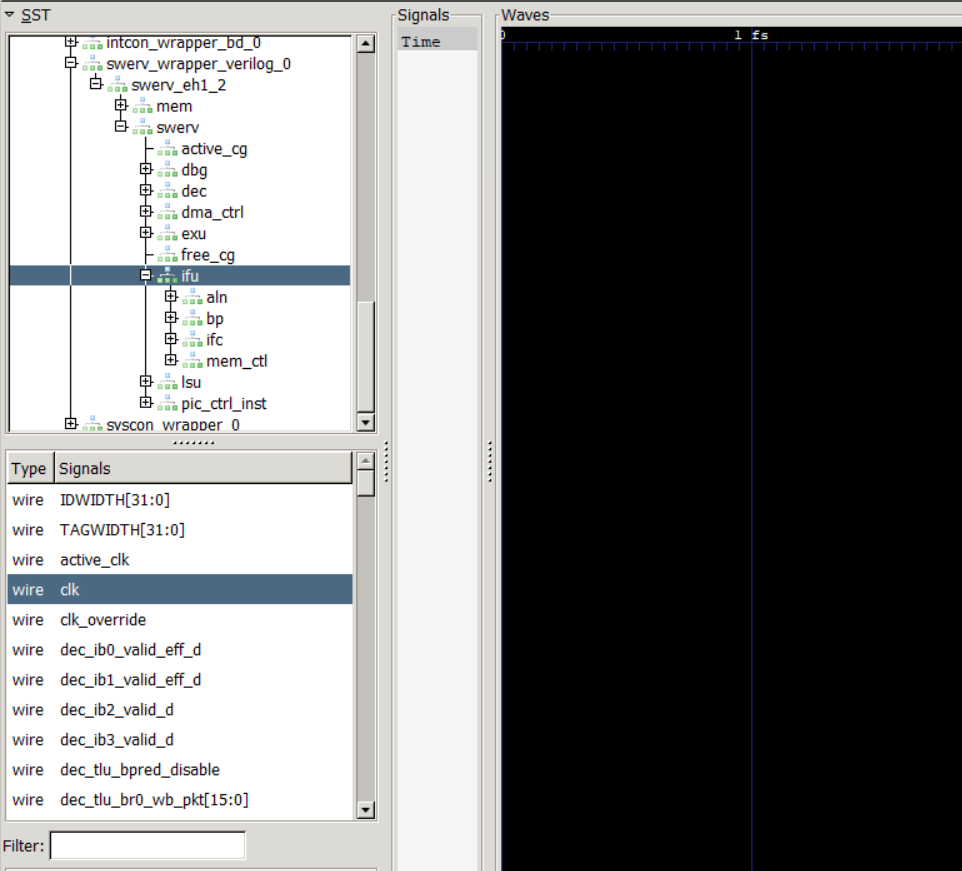
1. 이전 단계 다음, 몇 초 후에 [*RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/AL\_Operations/.pio/build/swervolf\_nexys* 내부에 *trace.vcd* 파일이 생성되어야 하며 *GTKWave*로 열 수 있습니다. Ubuntu 터미널을 열고 다음을 입력합니다.

gtkwave [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/AL\_Operations/.pio/build/swervolf\_nexys/trace.vcd

**WINDOWS:** 다운로드 받은 폴더 *gtkwave64*에는 *bin* 폴더 안에 *gtkwave.exe*라는 애플리케이션이 포함되어 있습니다. 해당 응용 프로그램을 두 번 클릭하여 GTKWave를 시작합니다. 애플리케이션 상단에서 **파일 – 새 탭 열기**를 클릭하고 *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/AL\_Operations/.pio/build/swervolf\_nexys* 폴더에 생성된 trace.vcd 파일을 엽니다.

**5단계. GTKWave에서 시뮬레이션 분석**

1. 이제 클럭, 명령어 및 레지스터 신호를 추가합니다. *GTKWave*의 왼쪽 상단 창에서 SoC 계층을 확장하여 그래프에 신호를 추가할 수 있습니다. 계층 구조를 **TOP → rvfpgasim → swervolf → swerv\_wrapper\_verilog\_0 → swerv\_eh1\_2 → swerv**로 확장하고, 모듈 **ifu**(그림 10과 같이 강조 표시됨)를 클릭하고, signal clk(코어에 사용되는 클럭)를 선택하고, 오른쪽의 흰색 신호 패널 또는 검은색 웨이브 패널에 끌어다 놓습니다.



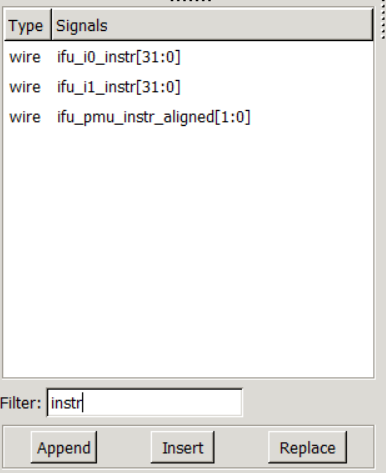
**그림 10. 그래프에 신호 clk 추가**

1. 확대/축소 맞춤을 수행한 다음 시계 신호 변경을 볼 수 있도록 여러 번 확대합니다(그림 11).



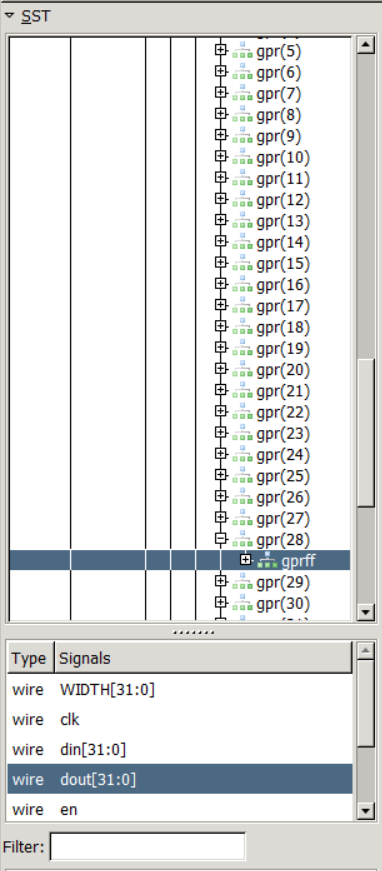
**그림 10. 확대**

1. 이제 양방향 수퍼스칼라 RISC-V 코어의 각 방향을 실행하는 명령을 보여주는 신호를 추가합니다. 동일한 모듈(**ifu**)에서 *ifu\_i0\_instr[31:0]* 및 *ifu\_i1\_instr[31:0]*(그림 12) 신호를 찾아 검정색 Waves 창으로 끌어옵니다. 접두사 *ifu*는 명령어 패치 단위를 나타내고, *i0*은 수퍼스칼라 방식 0을 나타내고, *i1*은 수퍼스칼라 방식 1을 나타냅니다. *instr[31:0]*은 32비트 명령어를 나타냅니다.
2. 검색 필터를 사용하여 신호를 빠르게 찾을 수 있습니다(그림 12 참조).



**그림 12. 타이밍 파형에 *ifu\_i0\_instr[31:0]* 및 *ifu\_i1\_instr[31:0]* 신호 추가**

1. 이제 레지스터 t3 (즉, 레지스터 번호 28, x28)의 값을 유지하는 신호를 추가합니다. **swerv** 아래의 계층 구조를 **dec → arf → gpr\_banks(0) → gpr(28)**으로 확장하고 **gprff** 모듈을 클릭하고(아래 그림과 같이 강조 표시됨) signal *dout[31:0]*을 선택하고, 신호 doout[31:0](AL\_Operations.S 예제에서 사용된 레지스터 x28의 내용을 보여줌)을 선택하고 검은색 Waves 창으로 끌어 놓습니다(그림 13).



**그림 13. 그래프에 신호 doout[31:0] 추가**

1. GTKWave에서 신호를 표시하는 또 다른 방법은 *.tcl* 파일을 사용하는 것입니다. *gtkwave\_signals.tcl* 파일은 *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/* 에 있습니다. 해당 파일을 열고 분석하십시오. 각 라인에는 그래프에 표시하려는 각 신호의 경로와 이름이 표시됩니다.

gtkwave::addSignalsFromList rvfpgasim.**clk**

gtkwave::addSignalsFromList rvfpgasim.swervolf.swerv\_wrapper\_verilog\_0.swerv\_eh1\_2.swerv.ifu.**ifu\_i0\_instr**

gtkwave::addSignalsFromList rvfpgasim.swervolf.swerv\_wrapper\_verilog\_0.swerv\_eh1\_2.swerv.ifu.**ifu\_i1\_instr**

gtkwave::addSignalsFromList rvfpgasim.swervolf.swerv\_wrapper\_verilog\_0.swerv\_eh1\_2.swerv.dec.arf.gpr\_banks(0).gpr(28).gprff.**dout**

GTKWave에서 *.tcl* 파일을 사용하려면 *File – Read Tcl Script File* 을 클릭하고 *RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/gtkwave+signals.tcl* 파일을 선택하면 됩니다.

그림 14는 AL\_Operations.S 프로그램과 이에 상응하는 기계 명령어를 보여줍니다.

|  |
| --- |
| **# RISC-V assembly # comment (t3 = x28) # machine code**  **li t3, 0x0 # t3 = 0 # 0x00000E13**    **REPEAT:**  **addi t3, t3, 6 # t3 = t3 + 6 # 0x006E0E13**  **addi t3, t3, -1 # t3 = t3 – 1 # 0xFFFE0E13**  **andi t3, t3, 3 # t3 = t3 AND 3 # 0x003E7E13**  **beq zero, zero, REPEAT # Repeat the loop # 0xFE000CE3**  **nop # nop # 0x00000013** |

**그림 14. 동등한 기계 코드가 있는 AL\_Operations.S**

이제 프로그램이 실행됨에 따라 신호가 변경되는 것을 확인하십시오. 프로그램이 실행될 때 명령어와 t3 (레지스터 x28)이 그림 15에 표시되어 있는 값이 될 것으로 예상합니다.

|  |
| --- |
| li t3, 0x0 # t3 = **0** # 0x00000E13  REPEAT: addi t3, t3, 6# t3 = 0 + 6 = **6** # 0x006E0E13  addi t3, t3, -1# t3 = **5** # 0xFFFE0E13  andi t3, t3, 3 # t3 = 5 & 3 = **1** # 0x003E7E13  beq zero, zero, REPEAT # Repeat the loop# 0xFE000CE3  nop # nop# 0x00000013  REPEAT: addi t3, t3, 6 # t3 = 1 + 6 = **7** # 0x006E0E13  addi t3, t3, -1# t3 = 7 – 1 = **6** # 0xFFFE0E13  andi t3, t3, 3# t3 = 6 & 3 = **2** # 0x003E7E13  beq zero, zero, REPEAT# Repeat the loop # 0xFE000CE3  ... |

**그림 15. AL\_Operations 실행 중 레지스터 t3 (x28)의 명령 흐름 및 값**

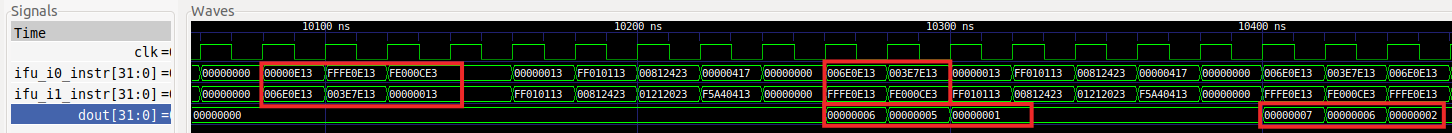
1. 루프의 첫 번째 및 두 번째 반복에 대한 세 가지 산술 논리 명령어의 실행을 분석할 수 있도록, 대략 10,100ns부분을 확대합니다(그림 16).

처음 두 명령(li t3, 0x0 = 0x00000E13 및 addi t3, t3, 6 = 0x006E0E13)이 신호 *ifu\_i0\_instr*[31:0] 및 *ifu\_i1\_instr*[31:0]에 표시된 대로 수퍼스칼라 RISC-V 프로세서의 각 방식으로 하나씩 먼저 패치(fetch)됩니다. 다음 두 명령어(addi t3, t3, -1 = 0xFFFE0E13 및 and.i t3, t3, 3 = 0x003E7E13)는 다음 주기에서 가져옵니다. 마지막 두 명령어는 다음 사이클에서 패치됩니다(beq zero, zero, REPEAT = 0xFE000CE3 및 nop = 0x00000013).

SweRV 코어의 9단계 파이프라인 프로세서 및 종속성으로 인해 명령을 가져온 후 8주기 뒤에 명령이 실행됩니다.

첫 번째 및 두 번째 명령어를 가져온 후 8 사이클이 지나면 x28 (t3) 은 첫 번째 명령어 li t3, 0x0 (0x00000E13) 때문에 0(이미 있던 값)이 됩니다. 한 사이클 후에 x28은 다음 명령어인 addi t3, t3, 6 (0x006E0E13) 때문에 0x6으로 업데이트됩니다.

다음으로 x28은 다음 명령어로 인해 5로 업데이트됩니다: addi t3, t3, -1 (0xFFFE0E13). 마지막으로 x28은 다음 명령어인 andi t3, t3, 3 (0x003E7E13) 때문에 1로 업데이트됩니다. 다음 두 명령어가 패치됩니다: beq zero, zero,REPEAT (0xFE000CE3) 및 nop (0x00000013); 분기가 수행되고 루프가 반복됩니다.



**그림 16. 예제의 세 가지 산술 논리 명령어 실행**

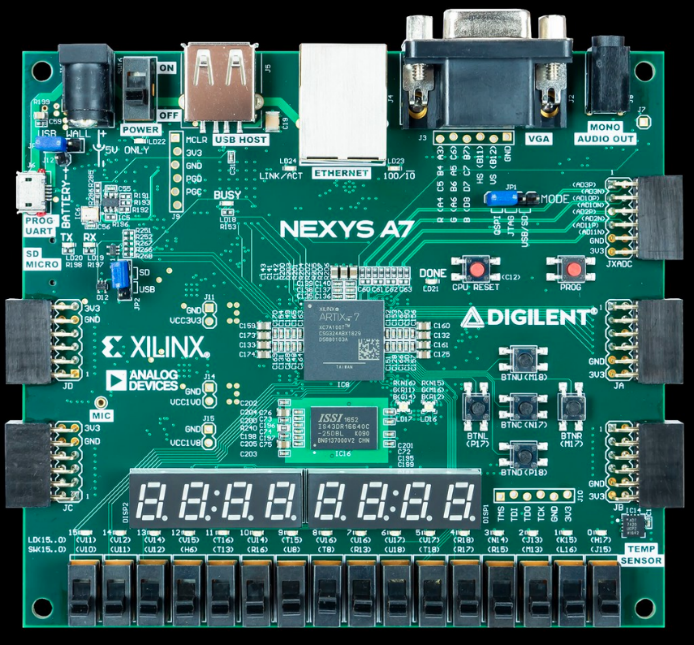
# Nexys A7 보드에서 프로그램 실행하기

이 섹션에서는 PlatformIO를 사용하는 RVfpgaNexys로 FPGA를 프로그래밍합니다. RVfpgaNexys로 FPGA를 프로그래밍하려면 다음 단계를 따르십시오.

다음 단계를 따르십시오:

**1단계.** Nexys A7 보드를 컴퓨터에 연결합니다.

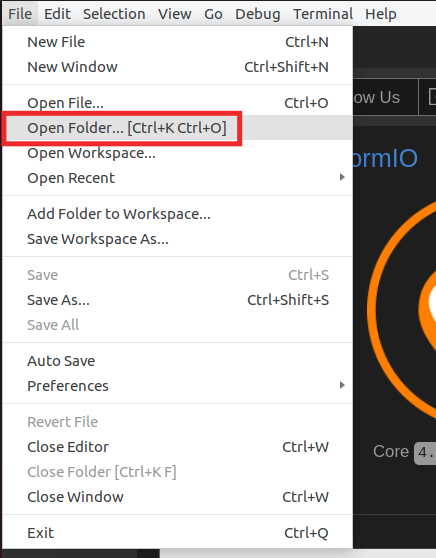
**2단계.** 왼쪽 상단의 스위치를 사용하여 Nexys A7 보드를 켭니다. (그림 17 참조)



**그림 17. Nexys A7 보드 ON/OFF 버튼**

**3단계.** VSCode 및 PlatformIO가 아직 열려 있지 않은 경우 엽니다.

**4단계**. 상단 메뉴 모음에서 파일 → 폴더 열기(그림 18 참조)를 클릭하고 *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/* 디렉토리로 이동합니다.



**그림 18. 폴더 열기**

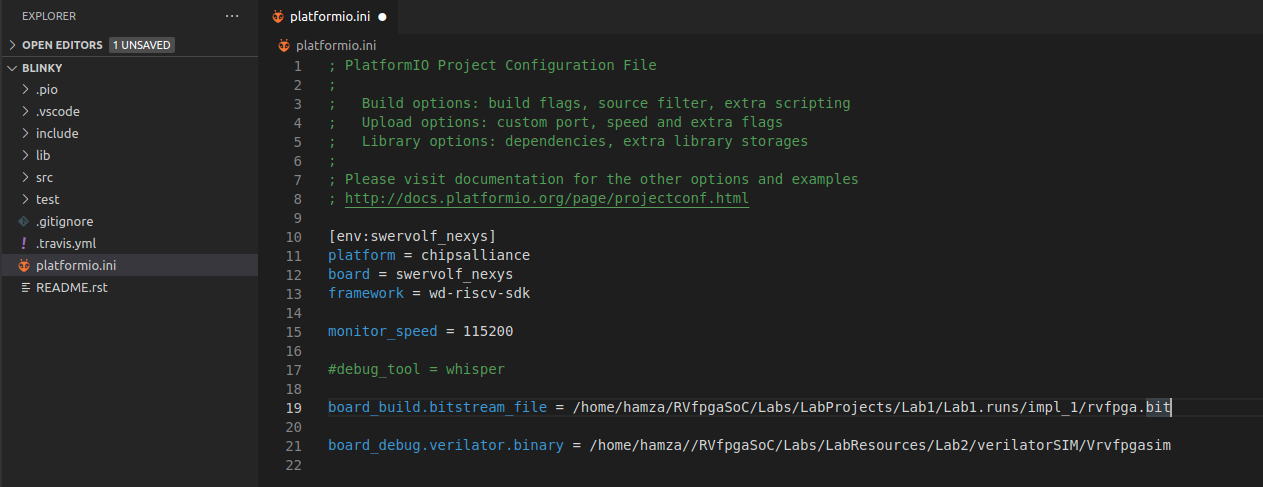
**5단계.** *Blinky* 디렉토리를 선택합니다(열지 말고 선택하고 창 상단의 확인을 클릭하기만 하면 됩니다). 이제 PlatformIO에서 예제를 엽니다.

**6단계.** 왼쪽 사이드바에서 *platformio.ini*를 클릭하여 *platformio.ini* 파일을 엽니다(그림 19 참조). 다음 줄을 편집하여 시스템에서 RVfpga 비트스트림의 경로를 설정합니다(그림 19 참조).

**7단계.** Vivado Block Design을 사용하여 생성된 "rvfpga.bit" 파일은 아래 경로에 있습니다:

board\_build.bitstream\_file = [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabProjects/Lab1/Lab1.runs/impl\_1/

rvfpga.bit

****

**그림 19. Platformio 초기화 파일: platformio.ini**

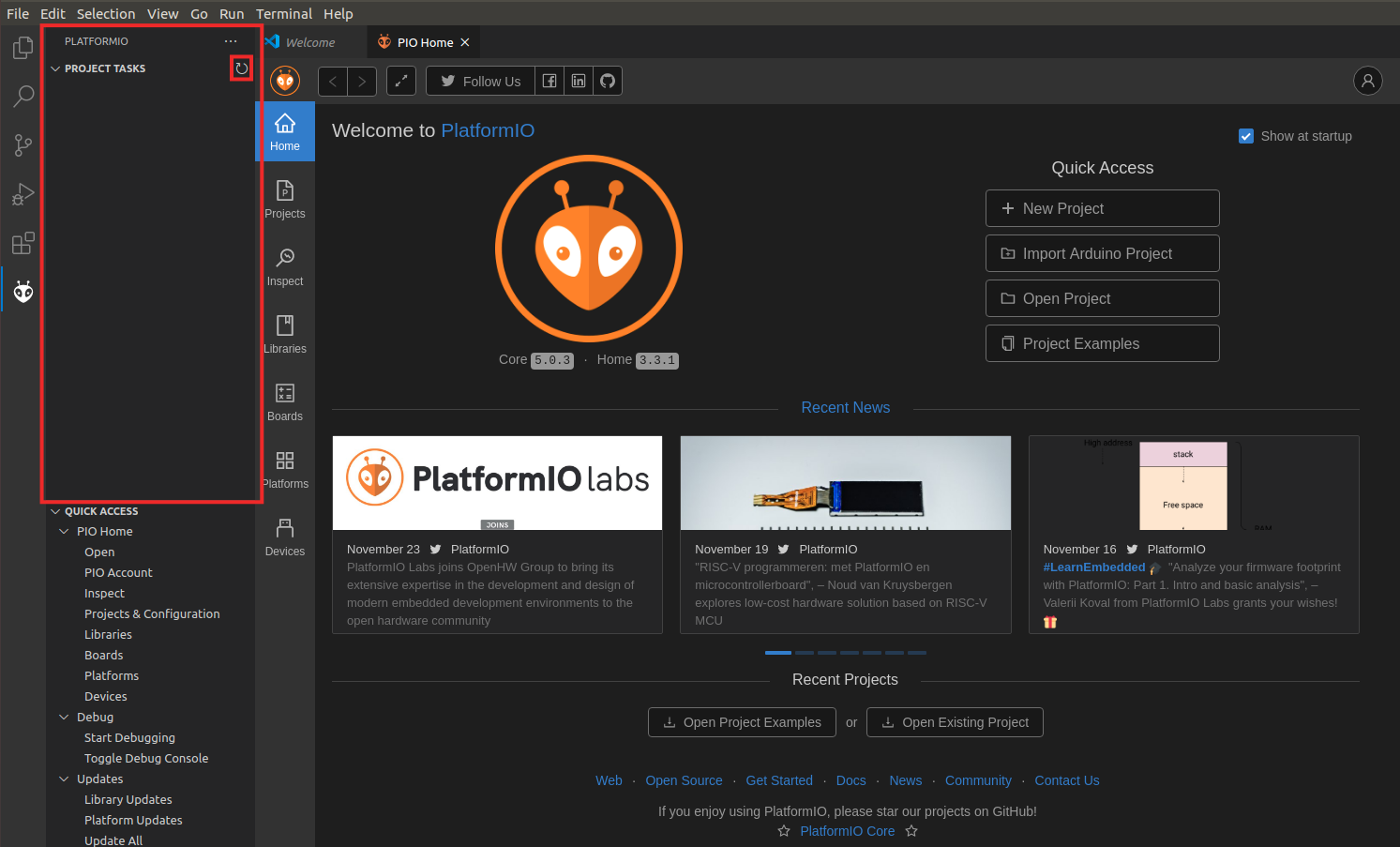
프로젝트 구성 파일(*platformio.ini*)에서 사용할 수 있는 다양한 명령이 있으며 이에 대한 정보는 다음 위치에서 찾을 수 있습니다. <https://docs.platformio.org/en/latest/projectconf/>.

**8단계.** 왼쪽 메뉴 리본에서 PlatformIO 아이콘을 클릭합니다(그림 20 참조).



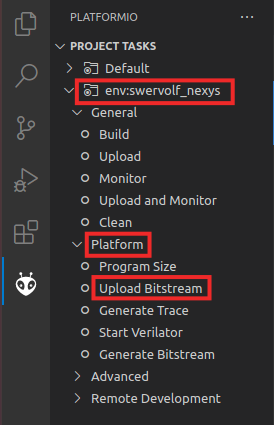
**그림 20. PlatformIO 아이콘**

프로젝트 작업 창이 비어 있는 경우(그림 21) 먼저 을 클릭하여 프로젝트 작업을 업데이트를(새로 고침) 해야 합니다. 몇 분이 소요될 수 있습니다.



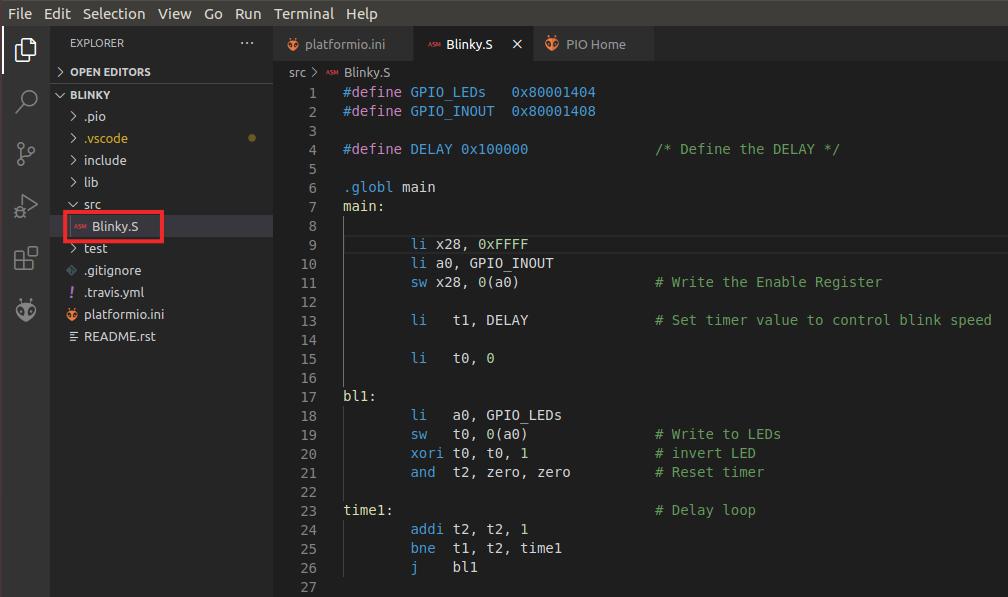
**그림 21. PROJECT TASKS 창 비어 있음 – 새로 고침**

Project Task→ env:swervolf\_nexys → Platform을 확장하고 그림 22와 같이 Upload Bitstream 를 클릭합니다. **1~2초 후에 FPGA는 블록 디자인 SoC로 프로그래밍 됩니다**(보드에서 사용 가능한 7-세그먼트 디스플레이는 8개의 0(zero)을 출력해야 합니다).



**그림 22. 비트스트림 업로드**

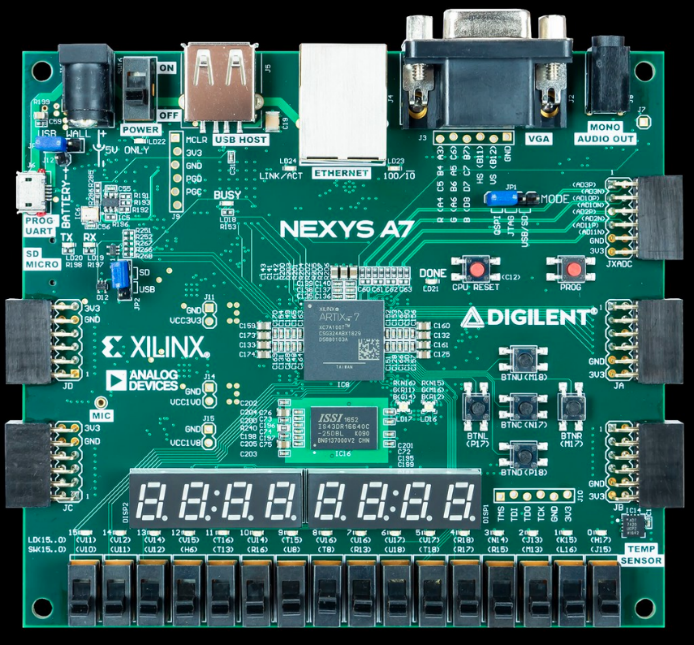
이제 비트스트림이 업로드 되었으므로 디버깅 프로세스를 시작합니다.

****

**그림 23. PlatformIO의 blinky.S**

**9단계**. 프로그램을 실행하고 디버그하려면 를 클릭하십시오. 그런 다음 재생 버튼을 클릭하여 디버깅을 시작합니다. PlatformIO는 주 기능(main function)의 시작 부분에 임시 중단점(breakpoint)을 설정합니다. 따라서 계속 버튼을 클릭하여 프로그램을 실행합니다.

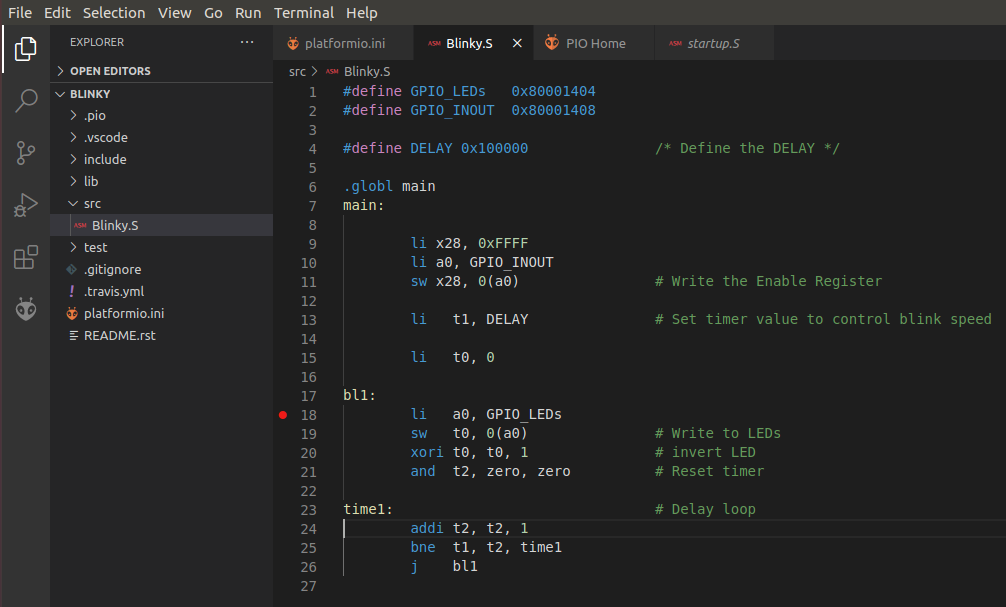
**10단계.** 보드에서 가장 오른쪽에 있는 LED가 깜박이기 시작하는 것을 볼 수 있습니다.



**그림 24. 맨 오른쪽 LED 깜박임**

**11단계.** 일시 중지 버튼 을 클릭하여 실행을 일시 중지합니다. 실행은 무한 루프 내부(아마도 time1 지연 루프 내부)에서 중지됩니다.

**12단계.** 줄 번호 18의 왼쪽을 클릭하여 중단점을 만듭니다. 빨간색 점이 나타나고 중단점이 BREAKPOINTS 탭에 추가됩니다(그림 25 참조).

****

**그림 25. blinky.S 에서 중단점 설정하기**

**13단계.** 그런 다음 계속 버튼을 클릭하여 실행을 계속합니다. 실행은 계속되고 가장 오른쪽 LED에 1(또는 0)을 쓰는 저장 워드(sw) 명령 후에 중지됩니다.

**단계 14.** 여러 번 실행을 계속하십시오. 가장 오른쪽에 있는 LED에 대한 값 구동이 매번 변경되는 것을 볼 수 있습니다.

**15단계.** 디버깅을 중지하고 를 클릭하여 탐색기 창으로 돌아갑니다. *File* → *Close Folder* 를 선택하여 프로그램을 닫습니다.

실습 1에서 만든 블록 설계 모듈을 사용하여 RVfpgaSIM 및 RVfpgaNexys에서 예제 프로그램을 성공적으로 실행했습니다.