

**THE IMAGINATION UNIVERSITY PROGRAMME**

**RVfpga Lab 3**

**RISC-V 어셈블리 언어**

# 소개

프로그래머에게는 C, Java 및 Python과 같은 고급 언어로 프로그래밍하는 것이 효율적입니다. 이러한 상위 언어는 간단한 명령어 그룹인 어셈블리 언어로 번역됩니다. 특정 타이밍을 보장하거나 계산 시간을 줄이기 위해 때때로 성능 또는 타이밍에 중요한 코드 섹션이 어셈블리로 작성됩니다. 이 LABS에서는 PlatformIO를 사용하여 RVfpga 시스템에서 실행할 수 있는 RISC-V 어셈블리 언어 프로그램을 만드는 방법을 보여줍니다. 먼저 RISC-V 어셈블리에 대한 간략한 개요를 제공한 다음 RVfpgaNexys 에서 (Verilator와 Whisper 를 사용하여 시뮬레이션 환경에서 이러한 프로그램을 수행할 수 있습니다) 어셈블리 프로그램을 만들고 실행하는 방법을 보여줍니다. 그런 다음 RISC-V 어셈블리 프로그램 작성을 연습할 수 있는 실습을 제공합니다.

# RISC-V 어셈블리 언어 개요

RISC-V 어셈블리 언어에는 상위 수준 코드를 구현하는 데 사용되는 간단한 지침이 포함되어 있습니다. 예를 들어 몇 가지 일반적인 RISC-V 명령어에는 더하기, 빼기, 곱하기를 하기 위한 add, sub, mul 명령어가 포함됩니다.

RISC-V 명령어의 기본 유형은 계산 (산술, 논리 및 시프트) 명령어, 메모리 작업 및 분기/점프입니다. 가장 일반적인 RISC-V 명령어는 표 1에 나와 있습니다. 명령어는 레지스터 또는 메모리에 있거나 상수로(즉시) 인코딩된 피 연산자를 사용합니다. RISC-V에는 32 비트 레지스터가 포함됩니다. 표 2는 32 개의 RISC-V 레지스터의 이름을 나열합니다. 이름 (예: zero, s0, t5 등) 또는 레지스터 번호 (예: x0, x8, x30)로 지정할 수 있습니다. 프로그래머는 일반적으로 레지스터의 일반적인 목적에 대한 정보를 보유하는 레지스터 이름을 사용합니다. 예를 들어 저장된 레지스터 s0-s11은 일반적으로 프로그램 변수에 사용되는 반면 임시 레지스터 t0-t6 은 임시 계산에 사용됩니다. zero 레지스터 (x0)는 프로그램에서 일반적으로 필요한 값이므로 항상 0 값을 포함합니다. 다른 레지스터도 표 2와 같이 특정 용도로 사용되지만이 LABS에서는 zero 레지스터와 임시 및 저장된 레지스터만 사용하면 됩니다.

표 1. 일반적인 RISC-V 어셈블리 지침

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | **RISC-V Assembly** | | **Description** | | **Operation** |
| **Computational** | add s0, s1, s2 | | Add | | s0 = s1 + s2 | |
| sub s0, s1, s2 | | Subtract | | s0 = s1 - s2 | |
| addi t3, t1, -10 | | Add immediate | | t3 = t1 – 10 | |
| mul t0, t2, t3 | | 32-bit multiply | | t0 = t2 \* t3 | |
| div s9, t5, t6 | | Division | | t9 = t5 / t6 | |
| rem s4, s1, s2 | | Remainder | | s4 = s1 % s2 | |
| and t0, t1, t2 | | Bit-wise AND | | t0 = t1 & t2 | |
| or t0, t1, t5 | | Bit-wise OR | | t0 = t1 | t5 | |
| xor s3, s4, s5 | | Bit-wise XOR | | s3 = s4 ^ s5 | |
| andi t1, t2, 0xFFB | | Bit-wise AND immediate | | t1 = t2 & 0xFFFFFFFB | |
| ori t0, t1, 0x2C | | Bit-wise OR immediate | | t0 = t1 | 0x2C | |
| xori s3, s4, 0xABC | | Bit-wise XOR immediate | | s3 = s4 ^ 0xFFFFFABC | |
| sll t0, t1, t2 | | Shift left logical | | t0 = t1 << t2 | |
| srl t0, t1, t5 | | Shift right logical | | t0 = t1 >> t5 | |
| sra s3, s4, s5 | | Shift right arithmetic | | s3 = s4 >>> s5 | |
| slli t1, t2, 30 | | Shift left logical immediate | | t1 = t2 << 30 | |
| srli t0, t1, 5 | | Shift right logical immediate | | t0 = t1 >> 5 | |
| srai s3, s4, 31 | | Shift right arithmetic immediate | | s3 = s4 >>> 31 | |
| **Memory** | lw s7, 0x2C(t1) | | Load word | | s7 = memory[t1+0x2C] | |
| lh s5, 0x5A(s3) | | Load half-word | | s5 = SignExt(memory[s3+0x5A]15:0) | |
| lb s1, -3(t4) | | Load byte | | s1 = SignExt(memory[t4-3]7:0) | |
| sw t2, 0x7C(t1) | | Store word | | memory[t1+0x7C] = t2 | |
| sh t3, 22(s3) | | Store half-word | | memory[s3+22]15:0 = t315:0 | |
| sb t4, 5(s4) | | Store byte | | memory[s4+5]7:0 = t47:0 | |
| **Branch** | beq s1, s2, L1 | | Branch if equal | | if (s1==s2), PC = L1 | |
| bne t3, t4, Loop | | Branch if not equal | | if (s1!=s2), PC = Loop | |
| blt t4, t5, L3 | | Branch if less than | | if (t4 < t5), PC = L3 | |
| bge s8, s9, Done | | Branch if greater than or equal | | if (s8>=s9), PC = Done | |
| **Pseudoinstructions** | li s1, 0xABCDEF12 | | Load immediate | | s1 = 0xABCDEF12 | |
| la s1, A | | Load address | | s1 = Memory address where variable A is stored | |
| nop | | Nop | | no operation | |
| mv s3, s7 | | Move | | s3 = s7 | |
| not t1, t2 | | Not (Invert) | | t1 = ~t2 | |
| neg s1, s3 | | Negate | | s1 = -s3 | |
| j Label | | Jump | | PC = Label | |
| jal L7 | | Jump and link | | PC = L7; ra = PC + 4 | |
| jr s1 | | Jump register | | PC = s1 | |

실제 RISC-V 명령어 외에도 RISC-V에는 프로그래머가 일반적으로 사용하는 Pseudoinstructions (표 1 하단 참조)이 포함되어 있습니다. Pseudoinstructions은 하나 이상의 실제 RISC-V 명령어를 사용하여 구현됩니다. 예를 들어, 이동 Pseudoinstructions (mv s1, s2)는 s2의 내용을 복사하여 s1에 넣습니다. 이러한 방식으로 실제 RISC-V 명령어를 같이 사용하여 구현됩니다: addi s1, s2, 0.

**표 2. RISC-V 레지스터**

|  |  |  |
| --- | --- | --- |
| **Name** | **Register Number** | **Use** |
| **zero** | **x0** | Constant value 0 |
| **ra** | **x1** | Return address |
| **sp** | **x2** | Stack pointer |
| **gp** | **x3** | Global pointer |
| **tp** | **x4** | Thread pointer |
| **t0-2** | **x5-7** | Temporary variables |
| **s0/fp** | **x8** | Saved register / Frame pointer |
| **s1** | **x9** | Saved register |
| **a0-1** | **x10-11** | Function arguments / Return values |
| **a2-7** | **x12-17** | Function arguments |
| **s2-11** | **x18-27** | Saved registers |
| **t3-6** | **x28-31** | Temporary variables |

# 마침표로 시작하는 명령은 어셈블러 지시문 입니다. 이것은 번역될 코드라기 보다는 어셈블러에 대한 명령입니다. 어셈블러에게 코드와 데이터를 배치할 위치를 알려주고 프로그램에서 사용할 텍스트와 데이터 상수를 지정하는 등의 작업을 수행합니다. 표 3은 RISC-V의 기본 어셈블러 지시문을 보여줍니다. (*The RISC-V Reader: An Open Architecture Atlas, Patterson & Waterman*, © 2017).

**Table 3. RISC-V main directives**

|  |  |
| --- | --- |
| **Directive** | **Description** |
| **.text** | 후속 항목은 텍스트 섹션 (기계 코드)에 저장됩니다. |
| **.data** | 후속 항목은 데이터 섹션 (글로벌 변수)에 저장됩니다. |
| **.bss** | 후속 항목은 bss 섹션에 저장됩니다 (글로벌 변수는 0으로 초기화 됨). |
| **.section .foo** | 후속 항목은 .foo라는 섹션에 저장됩니다. |
| **.align n** | 2n 바이트 경계에서 다음 데이텀을 정렬합니다. 예를 들어, .align 2는 단어 경계에서 다음 값을 정렬합니다. |
| **.balign n** | n 바이트 경계에서 다음 데이텀을 정렬합니다. 예를 들어 .balign 4 는 단어 경계에서 다음 값을 정렬합니다. |
| **.globl sym** | 레이블 sym 이 global 이며 다른 파일에서 참조될 수 있음을 선언합니다. |
| **.string “str”** | 문자열 str을 메모리에 저장하고 null로 종료합니다. |
| **.word w1,…,wn** | n 개의 32 비트 수량을 연속적인 메모리 워드에 저장합니다. |
| **.byte b1,…,bn** | n 8 비트 수량을 연속적인 메모리 바이트에 저장합니다. |
| **.space** | 초기값 없이 변수를 저장하려면 메모리 공간을 예약하십시오. 일반적으로 입력 변수로 사용되지 않는 출력 변수를 선언하는 데 사용됩니다. 예약하려는 공간은 항상 바이트 수로 표현되어야 합니다. 예를 들어, 지시문 RES: .space 4는 초기화되지 않은 4 바이트 (즉,한 단어)를 예약합니다. |
| **.equ name,constant** | 상수값을 가지는 심볼 이름을 정의합니다. 예를 들어, .equ N, 12는 값이 12인 심볼 N을 정의합니다. |
| **.end** | 어셈블러는 작업을 완료합니다.  지시문 .end, 이 지시문 뒤에 있는 모든 텍스트는 무시됩니다. |

# 아래 예 (표 4-표 5 참조)는 RISC-V 어셈블리에서 몇 가지 높은 수준의 구조를 코딩하는 방법을 보여줍니다. Branch 명령어 (beq, bne, blt및 bge )는 조건부로 레이블로 이동합니다. 반면에 점프 명령 (j)은 무조건 레이블로 점프합니다. 한 줄 주석은 RISC-V 어셈블리에서 C 및 #에서 //로 표시됩니다.

첫 번째 예 (if/else 문 구현, 표 4 참조)에서 C 코드와 RISC-V 어셈블리 코드는 반대의 경우를 확인합니다. C 코드는 보다 작음 (<)을 확인하고 어셈블리는 크거나 같음(> =)을 확인합니다.

**표 4. RISC-V 어셈블리 예 1: if / else 문**

|  |  |
| --- | --- |
| **// C Code**  int a, b, c;  if (a < b)  c = 5;  else  c = a + b; | **# RISC-V Assembly**  # s0 = a, s1 = b, s2 = c  bge s0, s1, L1 # if (a >= b) goto L1  addi s2, zero, 5 # c = 5  j L2 # jump over else block  L1: add s2, s0, s1 # c = a + b  L2: |

두 번째 예제에서 (정수 배열 조작, 표 5 참조) RISC-V 어셈블리 코드는 임시 레지스터 (t0-t3)를 사용하여 상수 100 및 데이터 배열의 기본 주소와 같은 임시 값을 보유합니다. 처음 세 개의 명령어에서 레지스터를 초기화한 후 RISC-V 어셈블리 코드는 bge (보다 크거나 같은 경우 branch) 명령어를 사용하여 i > = 100인지 확인합니다; 다시 말해 이것은 C 코드와 반대되는 경우입니다. 해당 조건이 충족되면 for loop문이 수행됩니다. Branch를 가져 오지 **않으면** i 는 100보다 작고 나머지 코드가 실행됩니다. 인덱스 i 는 정수 (32 비트 2의 보수 숫자)가 4 바이트의 메모리를 채우기 때문에 기본 주소에 더하기 전에 4 (slli t2, s0, 2 명령어 사용)를 곱합니다. RISC-V에서 메모리는 바이트 주소 지정이 가능합니다 (즉, 각 바이트에는 고유의 주소가 있음). 배열이 문자 배열이면 (예: char data[100];) 각 배열 요소는 한 바이트만 차지하고 i 는 배열 인덱스 i (array[i].)의 주소를 형성하기 위해 기본 주소에 직접 더하여 질 수 있습니다. 배열 요소를 읽고, 10씩 감소하고 (각각 lw, addi 및 sw 명령어를 통해) 기록 후, 배열 인덱스 i (즉, s0)가 증가하고 프로그램이 for loop문의 시작 부분으로 다시 점프합니다, (j L5 명령어 사용).

**Table 5. RISC-V 어셈블리 예제 2 : 정수 배열 처리**

|  |  |
| --- | --- |
| **// C Code**  int i;  int data[100];  for (i=0; i<100; i++)  array[i] = array[i]-10; | **# RISC-V Assembly**  # s0 = i, t1 = base address of data (assumed  # to be at 0x300)  addi s0, zero, 0 # i = 0  addi t0, zero, 100 # t0 = 100  li t1, 0x300 # base address of array  L5: bge s0, t0, L7 # if (i>=100) exit loop  slli t2, s0, 2 # t2 = i\*4  add t2, t1, t2 # address of data[i]  lw t3, 0(t2) # t3 = array[i]  addi t3, t3, -10 # t3 = array[i]-10  sw t3, 0(t2) # array[i] = array[i]-10  addi s0, s0, 1 # i++  j L5 # loop  L7: |

RISC-V 어셈블리 언어에 대한 자세한 내용은 아래의 링크를 참조하십시오. RISC-V Instruction Set Manual (<https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>) *Digital Design and Computer Architecture*, Harris & Harris, Elsevier, © 2021 (2021 년 여름 출판 예정) or *The RISC-V Reader: An Open Architecture Atlas*, Patterson & Waterman, © 2017.

# RVfpga 용 RISC-V 어셈블리 프로그램 작성

이제 RISC-V 어셈블리 프로그램 작성을 연습할 준비가 되었습니다. 자체 프로그램을 작성하기 전에 다음 단계에 따라 PlatformIO 프로젝트를 설정하고 RVfpgaNexys에서 (Verilator와 Whisper 를 사용하여 시뮬레이션 환경에서 이러한 프로그램을 수행할 수 있습니다) 어셈블리 프로그램을 만들고 실행하십시오.

1. RVfpga 프로젝트 만들기
2. RISC-V 어셈블리 언어 프로그램 작성
3. Nexys A7 FPGA 보드에 RVfpgaNexys 다운로드
4. 어셈블리 프로그램 컴파일, 다운로드 및 실행

1 단계. RVfpga 프로젝트 만들기

RVfpga Lab 2의 1 단계를 따르십시오. 편의를 위해 여기에서 반복합니다. 시작 버튼을 클릭하고 VSCode를 입력한 다음 Virtual Studio Code를 클릭하여 VSCode를 엽니다 (그림 1 참조).

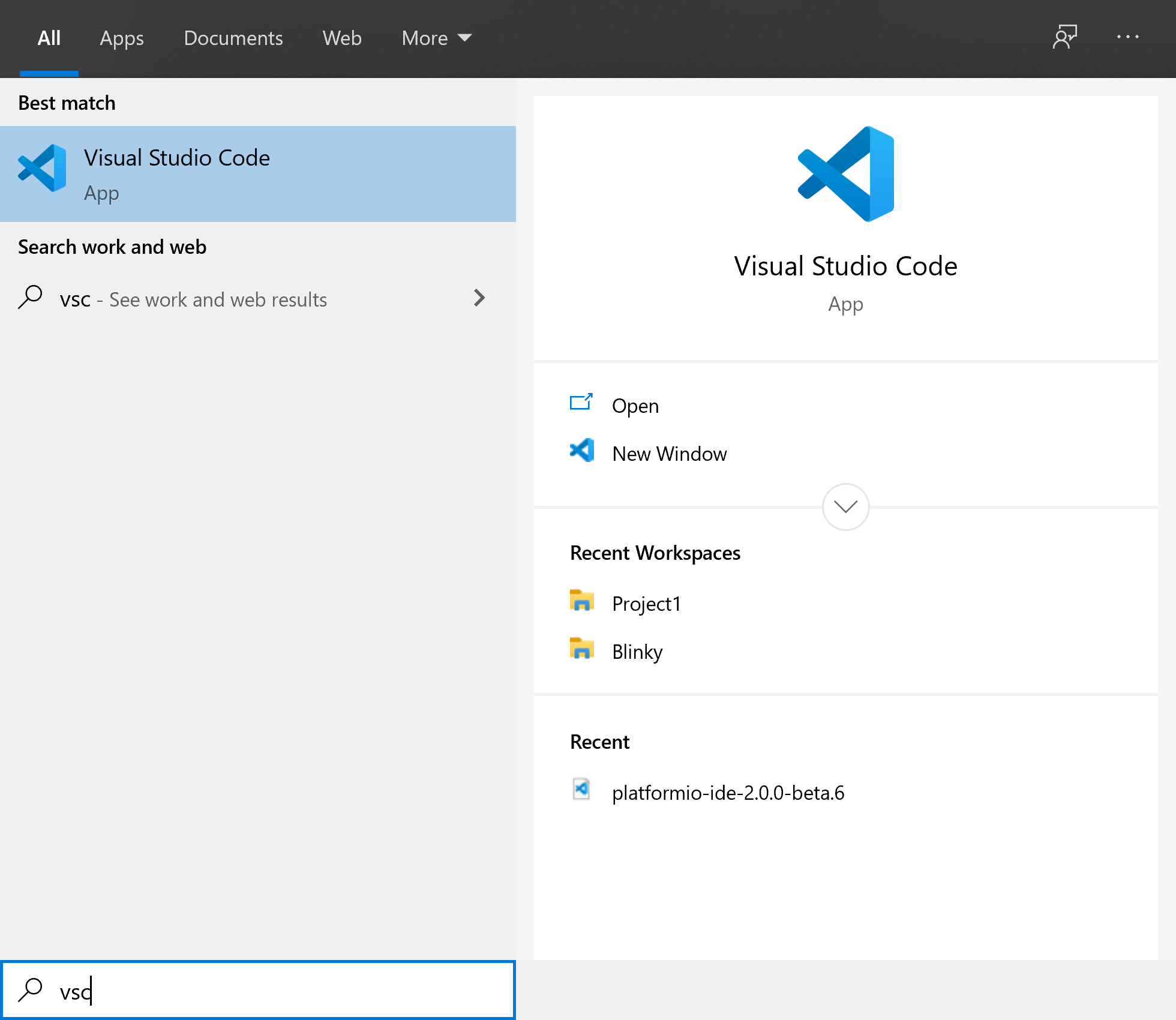


그림 1. VSCode 열기

VSCode를 시작할 때 PlatformIO가 자동으로 열리지 않는 경우 왼쪽 메뉴 리본에서 PlatformIO 아이콘을 클릭한 다음 PIO Home → Open을 클릭합니다 (그림 2 참조).

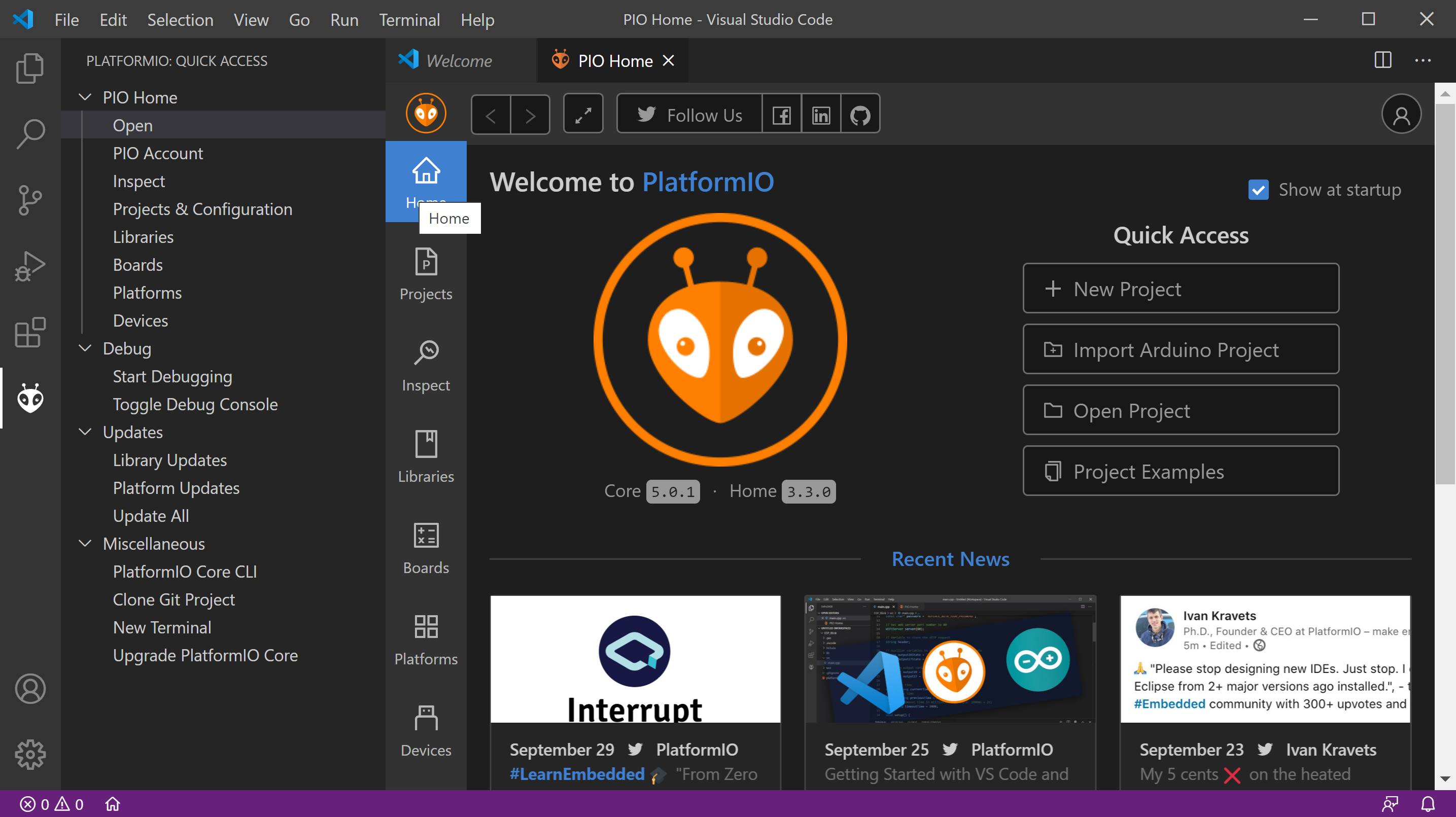


그림 2. PlatformIO를 열고 새 프로젝트 만들기

이제 PIO 홈 시작 창에서 새 프로젝트를 클릭합니다 (그림 2 참조).

그림 3과 같이 프로젝트 이름을 Project1로 지정하고 Board를 RVfpga: Digilent Nexys A7로 선택합니다 (RVfpga를 입력하기 시작하면 보드가 나타납니다). 기본 프레임 워크를 WD-framework (Western Digital framework – Freedom-E SDK gcc 및 gdb 포함)로 그대로 둡니다. 기본 위치 사용을 클릭 해제하고 다음 위치에 프로그램을 배치합니다.

[RVfpgaPath]/RVfpga/Labs/Lab3

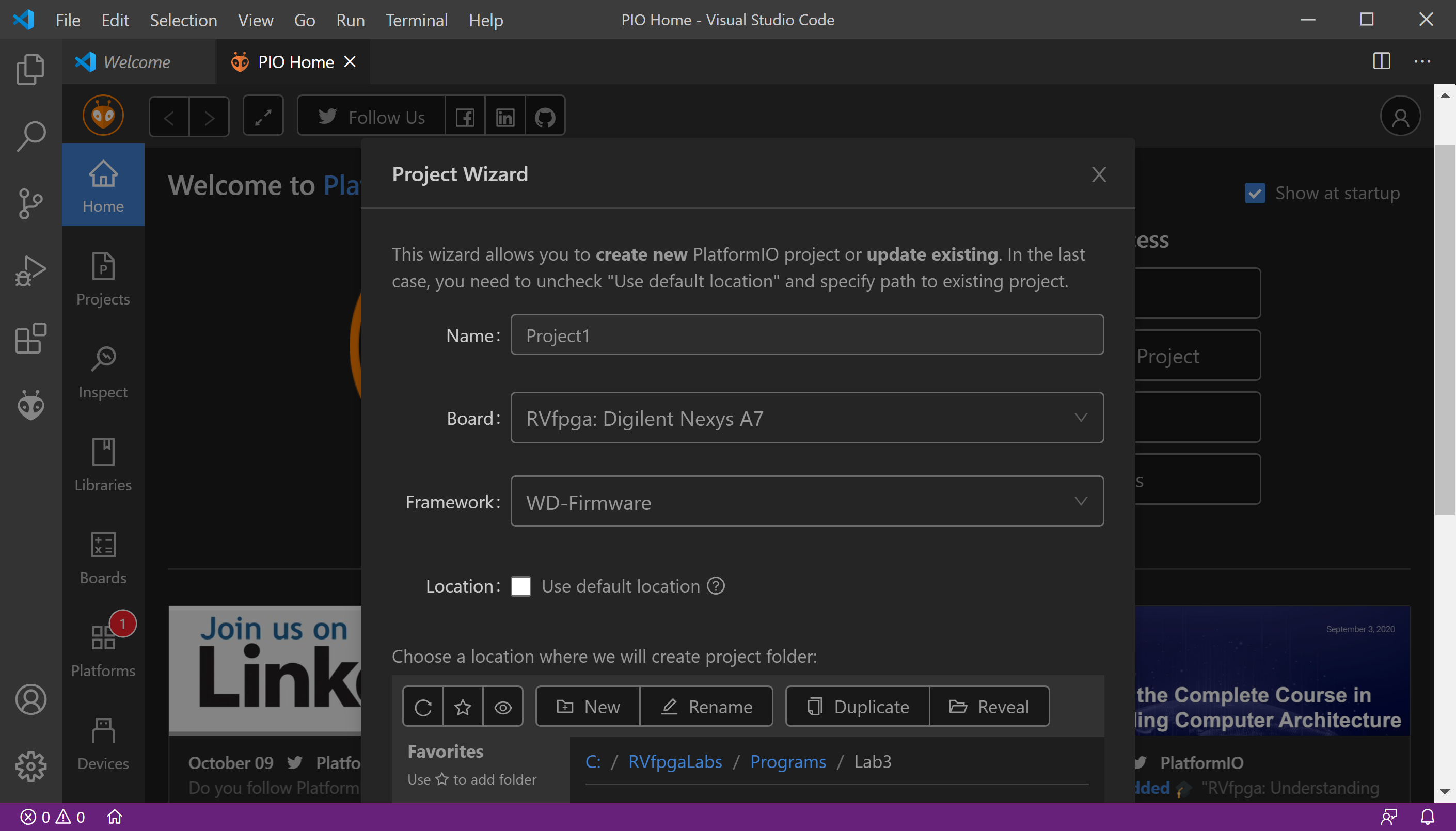


그림 3. 프로젝트 이름 지정 및 보드 및 프로젝트 폴더 선택

그런 다음 창 하단에서 마침을 클릭합니다 (그림 4 참조).

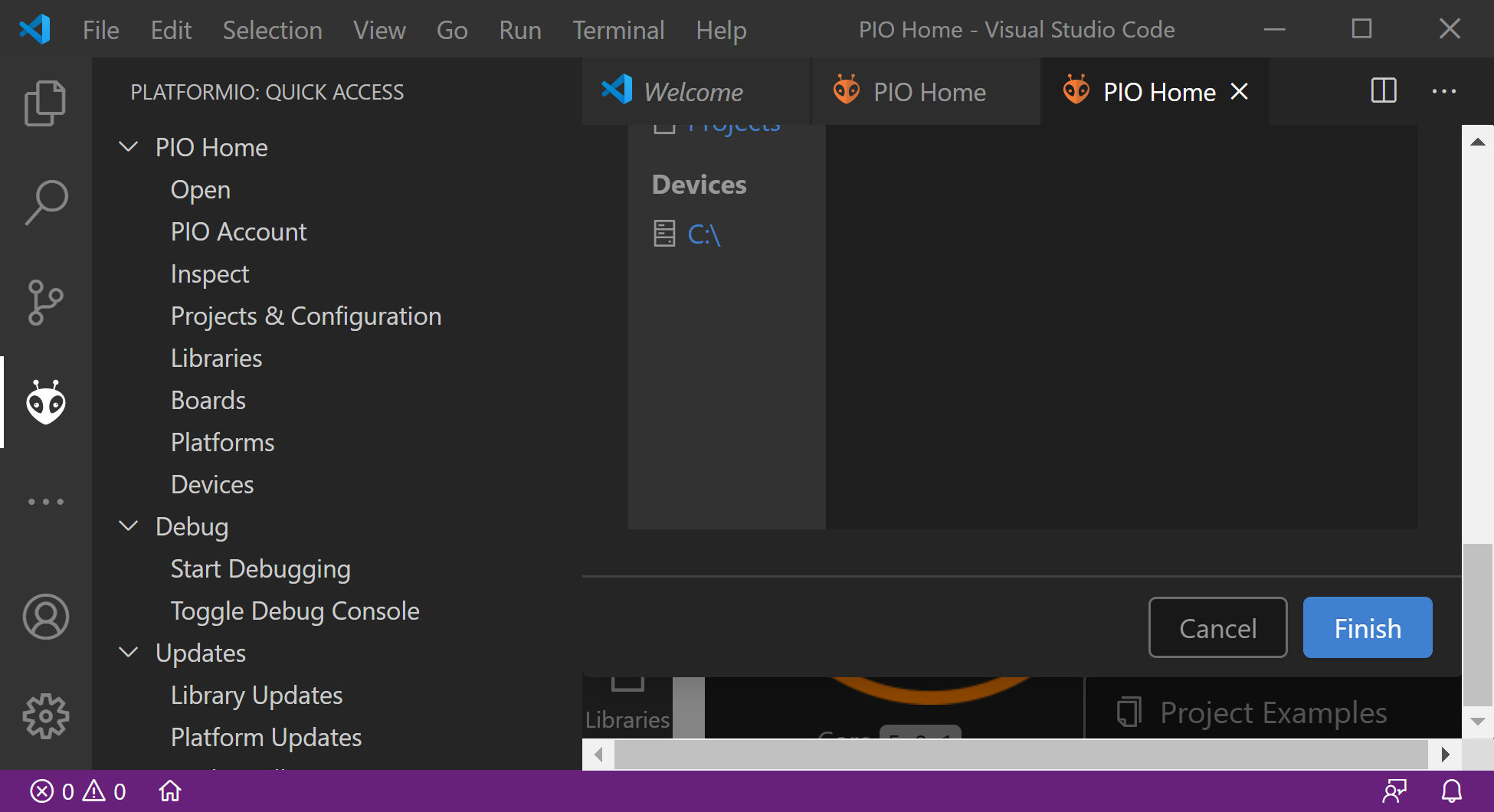


그림 4. 프로젝트 생성 완료

왼쪽 탐색기 창에서 PROJECT1 (확장해야 할 수 있음) 아래에서 platformio.ini를 두 번 클릭하여 엽니다 (그림 5 참조). PlatformIO 초기화 파일입니다.

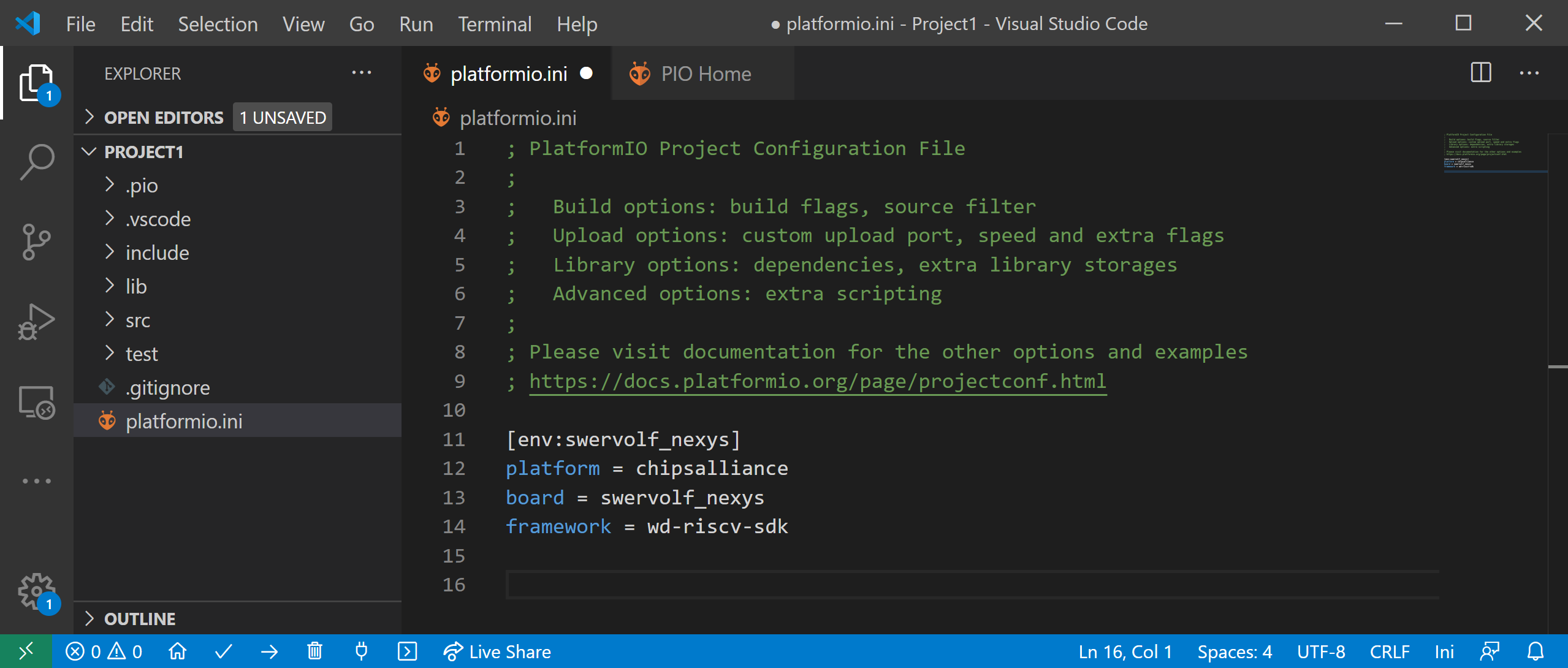


그림 5. PlatformIO 초기화 파일 : platformio.ini

그림 6과 같이 platformio.ini 파일에 다음 줄을 추가합니다.

board\_build.bitstream\_file = [RVfpgaPath]/RVfpga/Labs/Lab1/Project1/Project1.runs/impl\_1/rvfpga.bit

이 행은 PlatformIO가 FPGA에 로드할 비트 스트림 파일을 찾을 위치를 나타냅니다. 위의 경로는 LAB 1에서 만든 비트 스트림의 위치입니다. Lab 1을 완료하지 않은 경우 *[RVfpgaPath]/RVfpga/src/rvfpganexys.bit*에서 시작 안내서와 함께 배포된 RVfpganexys 비트 스트림을 사용할 수 있습니다. 작성 후 Ctrl-s를 눌러 platformio.ini 파일을 저장합니다.

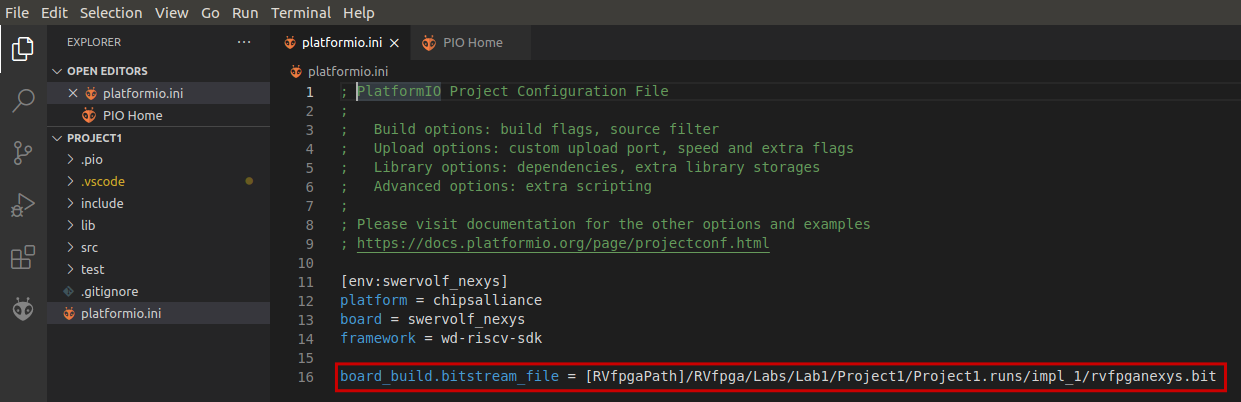


그림 6. RVfpgaNexys 비트 스트림 파일 (rvfpganexys.bit)의 위치 추가

시작 안내서에 사용된 예제에는 보다 완전한 *platformio.ini* 파일이 사용되었습니다. 추가 명령이 필요한 기능 (예: Verilator 시뮬레이터 경로, 직렬 콘솔 구성, Whisper 디버그 도구 등)을 사용하려는 경우 해당 예제에서 platformio.ini를 사용할 수 있습니다.

2 단계. RISC-V 어셈블리 언어 프로그램 작성

RISC-V 어셈블리 프로그램을 작성합니다. 파일 → 새 파일을 클릭합니다 (그림 7 참조).

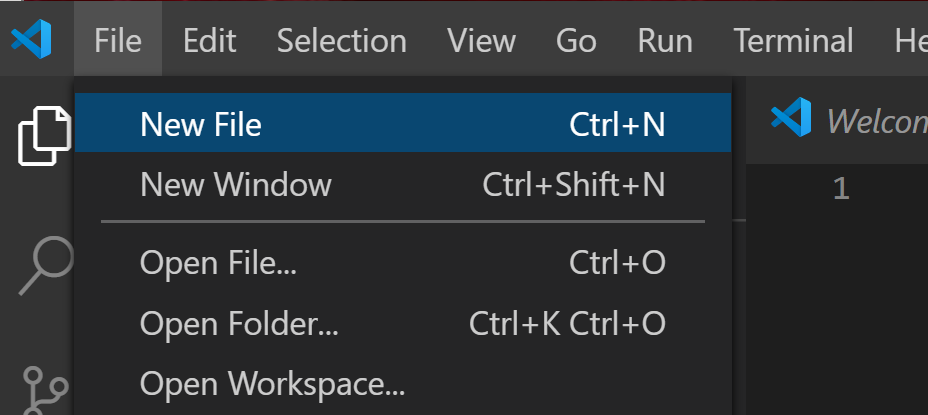


그림 7. 프로젝트에 파일 추가

빈 창이 열립니다. 그 다음 RISC-V 어셈블리 프로그램을 해당 창에 입력 (또는 복사/붙여 넣기)합니다 (그림 8 참조). 이 프로그램은 다음 파일에 제공됩니다.

[RVfpgaPath]/RVfpga/Labs/Lab3/ReadSwitches.S

// memory-mapped I/O addresses

# GPIO\_SWs = 0x80001400

# GPIO\_LEDs = 0x80001404

# GPIO\_INOUT = 0x80001408

.globl main

main:

main:

li t0, 0x80001400 # base address of GPIO memory-mapped registers

li t1, 0xFFFF # set direction of GPIOs

# upper half = switches (inputs) (=0)

# lower half = outputs (LEDs) (=1)

sw t1, 8(t0) # GPIO\_INOUT = 0xFFFF

repeat:

lw t1, 0(t0) # read switches: t1 = GPIO\_SWs

srli t1, t1, 16 # shift val to the right by 16 bits

sw t1, 4(t0) # write value to LEDs: GPIO\_LEDs = t1

j repeat # repeat loop

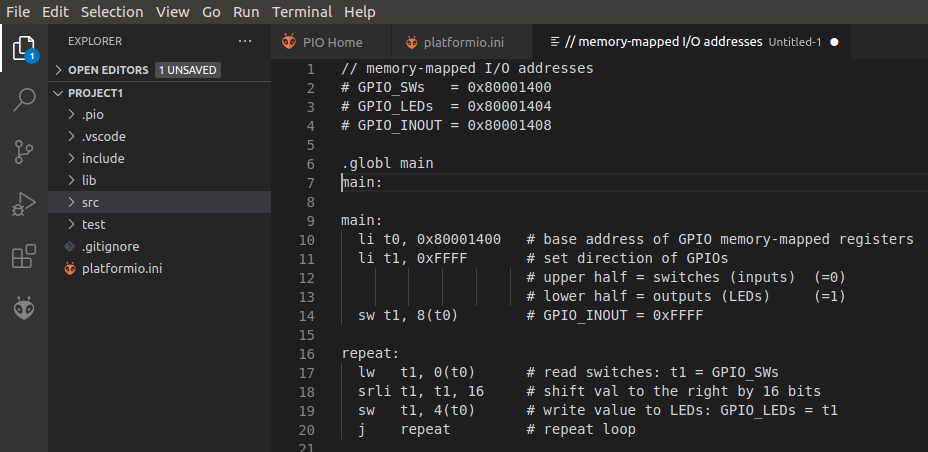


그림 8. RISC-V 어셈블리 프로그램 입력

어셈블리 코드는 코드 시작 부분에 다음 줄을 포함해야 합니다.

.globl main

main:

.globl어셈블러 지시문은 링크된 모든 파일에서 레이블을 표시합니다. boot 코드 (*~/.platformio/packages/framework-wd-riscv-sdk/board/nexys\_a7\_eh1/startup.S*)는 시스템을 구성하고 이 레이블 (*main*)로 이동합니다. 디버거는 시작할 때 임시 중단점을 설정합니다.

이RISC-V 어셈블리 프로그램은 Lab 2와 동일한 예제 프로그램이지만 이번에는 RISC-V 어셈블리로 작성되었습니다. 범용 I/O (GPIO)의 입력 및 출력 방향을 설정한 다음 스위치 값을 반복적으로 읽고 해당 값을 LED에 씁니다.

창에 프로그램을 입력한 후 Ctrl-s를 눌러 파일을 저장합니다. 이름을 ReadSwitches.S로 지정하고 Project1 디렉토리의 src 폴더에 저장합니다 (그림 9 참조).

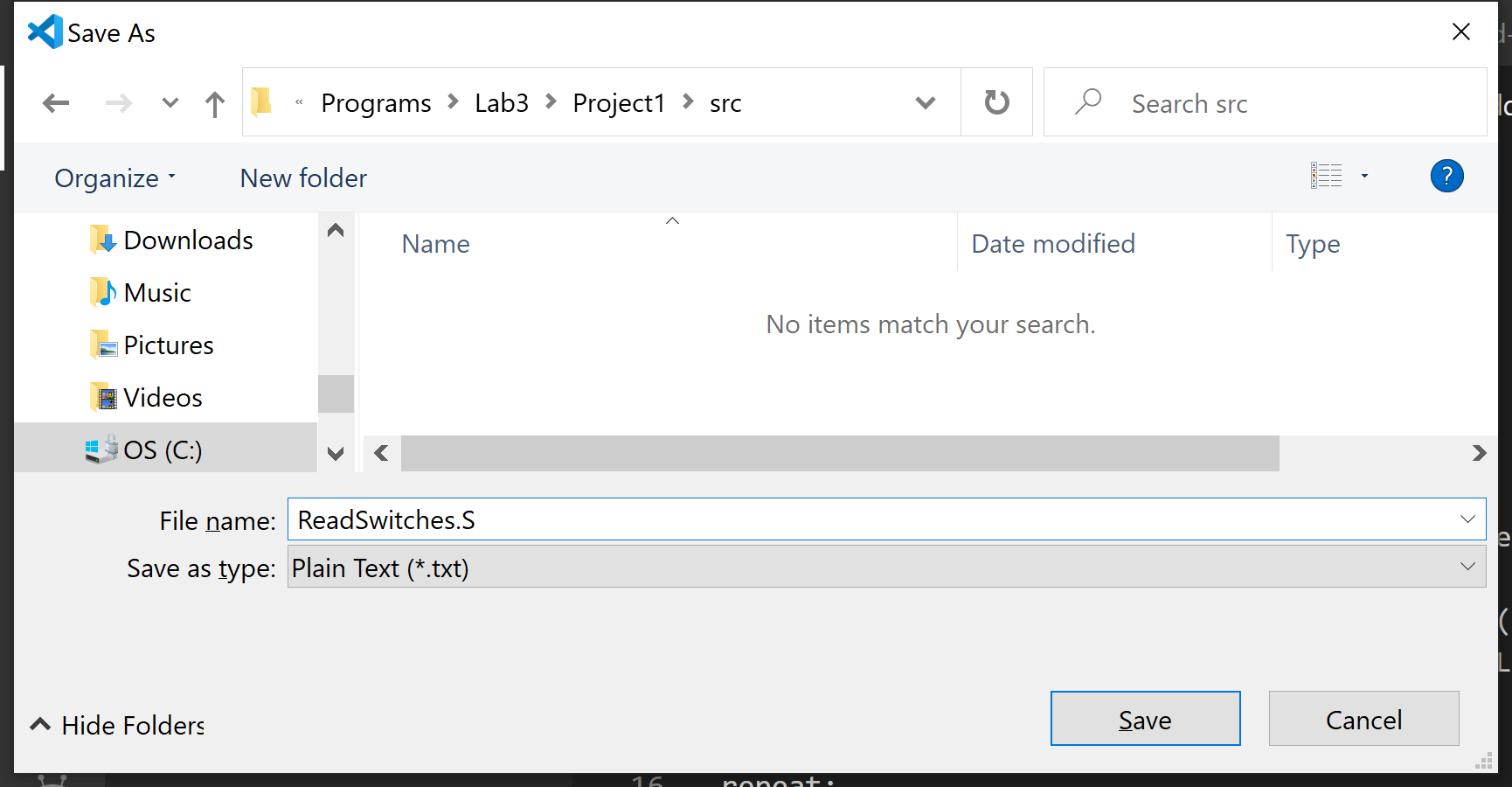


그림 9. 파일을 ReadSwitches.S로 저장

3 단계. Nexys A7 FPGA 보드에 RVfpgaNexys 다운로드

이제 Nexys A7 FPGA 보드에 RVfpgaNexys를 다운로드합니다. GSG 및 LAB 2에 설명된 대로 RVfpgaNexys 다운로드 지침을 따르십시오 - 편의를 위해 여기에서 반복합니다.

왼쪽 메뉴 리본에서 PlatformIO 아이콘을 클릭하여 Nexys A7 보드에 RVfpgaNexys를 다운로드한 다음 프로젝트 작업 → env:swervolf\_nexys → 플랫폼을 확장하고 Upload Bitstream 을 클릭합니다.

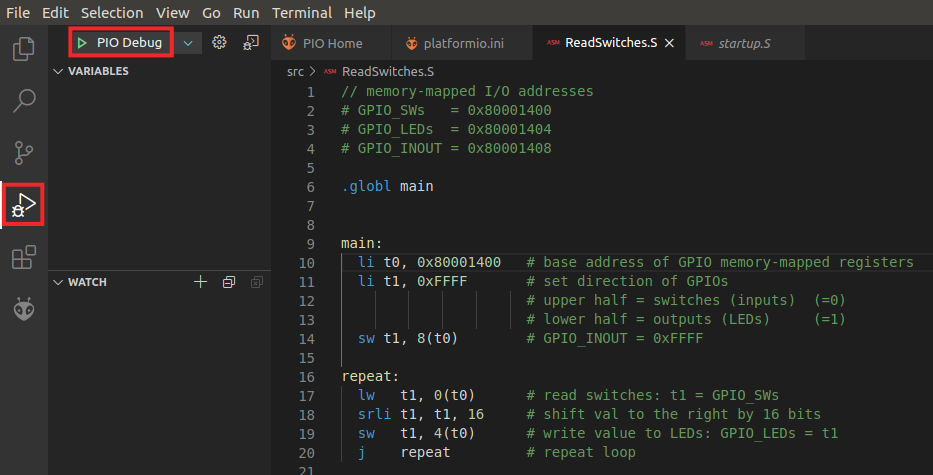
또는 PlatformIO 창의 하단 메뉴에서 PlatformIO: 새 터미널 버튼 ()을 클릭한 다음 PlatformIO 터미널에 다음을 입력 (또는 복사)하여 PlatformIO 터미널 창을 사용하여 RVfpgaNexys를 다운로드할 수 있습니다.

pio run -t program\_fpga

4 단계. RISC-V 어셈블리 프로그램 컴파일, 다운로드 및 실행

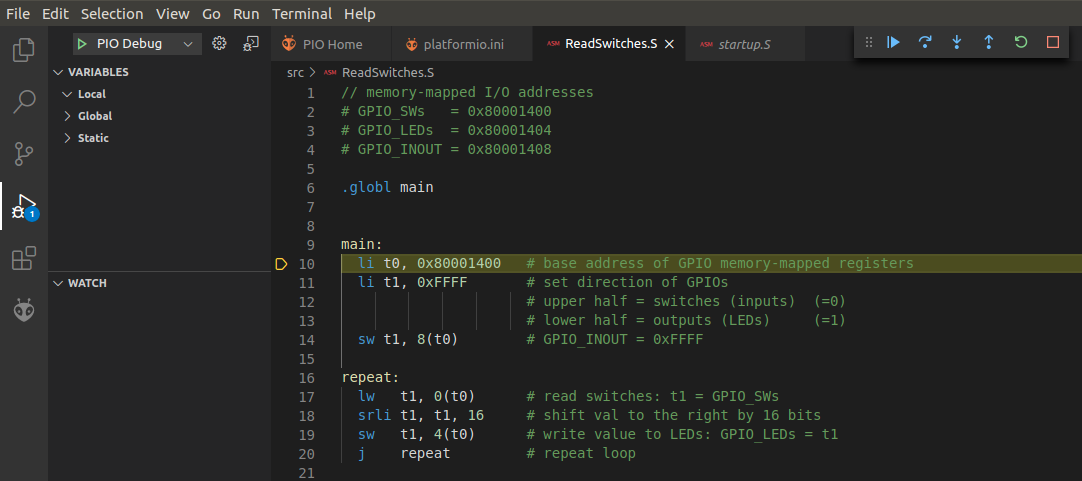
보드에서 RVfpgaNexys가 실행되고 있으므로 프로그램을 컴파일하고 RVfpgaNexys에 다운로드한 다음 실행 / 디버그합니다. VSCode가 아직 열려 있지 않으면 VScode를 엽니다. 마지막으로 작성한 프로젝트인 Project1이 자동으로 열립니다. 자동으로 열리지 않은 경우 PlatformIO 확장이 열려 있는지 확인하고 파일 → 폴더 열기를 클릭하고 이 LAB의 앞부분에서 만든 Project1을 선택합니다 (열지 않음).

왼쪽 메뉴 리본에서 실행 버튼을 클릭한 다음 디버깅 시작 버튼을 클릭합니다 (그림 10 참조).



**그림 10. RVfpgaNexys에서 프로그램 실행**

프로그램은 Nexys A7 보드의 FPGA에서 실행되는 RVfpga**Nexys**로 다운로드 됩니다. 이제 프로그램 실행 및 디버깅을 시작할 수 있습니다 (그림 11 참조).



**그림 11. RVfpgaNexys에서 실행되는 프로그램**

RVfpga 시작 안내서 및 Lab 2의 설명 대로 디버깅 도구 모음과 디버거 옵션을 사용하여 프로그램을 실행하고 관리합니다. 예를 들어 17 행에 breakpoint 를 설정한 다음 (행 번호 바로 왼쪽을 클릭하여) 스위치값이 로드될 때 레지스터 t1을 볼 수 있습니다. Stop 버튼 (또는 Shift-F5)을 눌러 디버깅 세션을 중지하면 디버깅 세션이 종료되지만 프로그램은 RVfpga**Nexys**에서 계속 실행됩니다.

# 실습

이제 LAB 2에서와 동일한 실습을 완료하여 고유한 RISC-V 어셈블리 프로그램을 작성하십시오. 이번에는 C 대신 RISC-V 어셈블리에서 연습하세요. 연습 설명에 대한 설명은 아래에서 반복됩니다.

Nexys A7 보드를 컴퓨터에 연결하고 전원을 켠 상태로 두면 다른 프로그램을 실행하는 동안 RVfpgaNexys를 보드에 다시 로드할 필요가 없습니다. 그러나 Nexys A7 보드를 끄면 PlatformIO를 사용하여 보드에 RVfpgaNexys를 다시 로드해야 합니다.

Verilator와 Whisper 를 사용하여 시뮬레이션 환경에서 이러한 프로그램을 수행할 수 있습니다.

실습 1. 스위치 값을 LED에 표시하는 RISC-V 어셈블리 프로그램을 작성하십시오. 값은 눈으로 깜박임을 볼 수 있을 만큼 느린 속도로 깜빡여야 합니다. 프로그램 이름을 FlashSwitchesToLEDs.S로 지정합니다.

실습 2. LED에있는 스위치의 역 값을 표시하는 RISC-V 어셈블리 프로그램을 작성하십시오. 예를 들어, 스위치가 (2 진수): 0101010101010101이면 LED는 다음과 같이 표시 되어야 합니다, 1010101010101010; 스위치가 1111000011110000이면 LED에 0000111100001111이 표시됩니다. 프로그램 이름을 DisplayInverse.S로 지정합니다.

실습 3. 모든 LED가 켜질 때까지 켜진 LED 수를 앞뒤로 스크롤하는 RISC-V 어셈블리 프로그램을 작성하십시오. 그런 다음 패턴이 반복 되어야 합니다. 프로그램 이름을 ScrollLEDs.S로 지정합니다.

프로그램으로 인해 다음이 발생해야 합니다.

1. 먼저 켜진 LED 하나가 오른쪽에서 왼쪽으로 스크롤되어야 합니다.

2. 가장 왼쪽의 LED에 도달하면 두 개의 LED가 왼쪽에서 오른쪽으로 스크롤한 다음 오른쪽에서 왼쪽으로 스크롤해야 합니다.

3. 이 두 LED가 맨 왼쪽 LED에 도달하면 세 개의 LED가 왼쪽에서 오른쪽으로 스크롤한 다음 오른쪽에서 왼쪽으로 스크롤해야 합니다.

4. 그런 다음 4 개의 LED가 스크롤되어야 합니다.

5. 모든 LED가 켜질 때까지 진행합니다.

6. 그런 다음 패턴이 반복되어야 합니다.

실습 4. 스위치의 LSB 4 개와 스위치의 MSB 4 개의 부호 없는 4 비트 합계를 표시하는 RISC-V 어셈블리 프로그램을 작성하십시오. LED의 LSB (가장 오른쪽) 4 개에 결과를 표시합니다. 프로그램 이름을 4bitAdd.S로 지정합니다. Unsigned 오버플로가 발생하면 LED의 다섯 번째 비트가 켜집니다 (carry out이 1 일 때).

**실습 5.** Euclidean 알고리즘에 따라 두 숫자 a와 b의 최대 공약수를 찾는 RISC-V 어셈블리 프로그램을 작성합니다. 값 a와 b는 프로그램에서 static으로 정의된 변수이여야 합니다. 프로그램 이름을 GCD.S로 지정합니다. 다음은 Euclidean 알고리즘에 대한 추가 정보입니다. <https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm> 혹은 "**유클리드 알고리즘**"을 구글에 검색 하시기 바랍니다.

**연습 6.** Fibonacci sequence 의 처음 12 개 숫자를 계산하고 그 결과를 길이가 12 인 유한 벡터 (즉, 배열) V에 저장하는 RISC-V 어셈블리 프로그램을 작성합니다. 이 무한 수열의 피보나치 수는 다음과 같이 정의됩니다.

V(0)=0, V(1)=1, V(i)=V(i-1)+V(i-2) (where i=0,1,2...)

즉, 요소 i에 해당하는 피보나치 수는 시리즈에서 이전 두 피보나치 수의 합입니다. 표 6은 i = 0에서 8까지의 피보나치 수를 보여줍니다.

표 6. 피보나치 시리즈

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***i*** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| ***V*** | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |

벡터의 차원 N은 프로그램에서 상수로 정의되어야 합니다. 프로그램 이름을 **Fibonacci.S**로 지정합니다.

**실습 7**. *N*-element 벡터(즉, array), *A*가 주어지면 *B*가 0보다 큰 짝수인 *A*의 요소만 포함하는 다른 벡터 *B*를 생성합니다. 예를 들어 *N* = 12이고 *A* =[0,1,2,7, -8,4,5,12,11, -2,6,3]라고 하면, *B*는 *B* = [2,4,12,6]입니다. 프로그램 이름을 **EvenPositiveNumbers.S**로 지정합니다.

**실습 8.** 두 개의 *N*-element 벡터(즉, array), *A*와 *B*가 주어지면 다음과 같이 정의된 다른 벡터 *C*를 만듭니다:

C(i) = |A[i] + B[N-i-1]|, i = 0,..,N-1.

새 벡터를 계산하는 프로그램을 RISC-V 어셈블리로 작성합니다. 프로그램에서 12 개 element 인 배열을 사용하십시오. 프로그램 이름을 **AddVectors.S**로 지정합니다.

**실습 9.** RISC-V 어셈블리에서 bubble sort 알고리즘을 구현합니다. 이 알고리즘은 다음 절차에 따라 벡터의 구성 요소를 오름차순으로 정렬합니다.

1. 완료 될 때까지 벡터를 반복해서 이동합니다.
2. V (i)> V (i + 1) 인 경우 인접한 구성 요소 쌍을 교환합니다.
3. 연속된 모든 구성 요소 쌍이 순서대로 정렬되면 알고리즘이 중지됩니다.

프로그램을 테스트하려면 요소가 12 개인 배열을 사용하십시오. 프로그램 이름을 **BubbleSort.S**로 지정합니다.

**실습 10.** 반복적인 곱셈을 통해 주어진 음이 아닌 숫자 *n*의 계승을(factorial) 계산하는 프로그램을 RISC-V 어셈블리로 작성하십시오. *n*의 여러 값에 대해 프로그램을 테스트해야 하지만 최종적으로 *n* = 7이어야 합니다. *n*은 프로그램 내에서 정적으로 정의된 변수이여야 합니다. 프로그램 이름을 **Factorial.S**로 지정합니다.