



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 9

인터럽트 구동 I/O

1. 소개

이 LAB에서는 인터럽트의 개념을 소개하고 이를 RVfpga에서 사용하는 방법을 보여줍니다. 인터럽트는 소프트웨어 또는 하드웨어에 의해 생성될 수 있습니다. 이 LAB에서는 물리적 핀 변경 값에 의해 트리거되는 하드웨어 인터럽트에 중점을 둡니다. 특히, **프로그래밍된 I/O**와 **인터럽트 구동 I/O**의 차이점을 설명하는 것으로, 섹션 2에서 시작합니다. 그런 다음 SweRV EH1 코어 (섹션 3)의 일부인 RVfpga 시스템의 인터럽트 컨트롤러 작동에 대해 설명합니다. 섹션 4에서는 하드웨어 주변 장치용 드라이버를 포함하는 소프트웨어인 Western Digital의 PSP (Peripherals Support Package) 및 BSP (Board Support Package)를 사용하여 외부 인터럽트를 구성하는 방법에 대해 설명합니다. 마지막으로 몇 가지 예제 프로그램 (섹션 5)을 소개하고 RVfpga 시스템의 하드웨어 인터럽트를 사용하고 확장하기 위한 몇 가지 연습 (섹션 6)을 제안합니다.

2. 프로그래밍된 I/O VS. 인터럽트 기반 I/O

주변 장치와 상호 작용하는 방법에는 프로그래밍된 I/O, 인터럽트 구동 I/O 및 DMA (직접 메모리 액세스)가 있습니다. 2-8 실습에서는 **프로그래밍된 I/O**를 사용하여 주변 장치와 상호 작용했습니다. 프로그래밍된 I/O에서 사용자 프로그램은 I/O 인터페이스를 지속적으로 폴링하고 상태에 따라 그에 따라 반응합니다. 예를 들어 실습 6의 기초 실습에서는 LED의 한쪽에서 다른쪽으로 반복적으로 이동하는 4개의 점등된 LED 블록의 속도와 방향을 제어하기 위해 스위치 0과 1을 지속적으로 폴링 (읽기)하여 프로그래밍된 I/O를 사용했습니다. 프로그래밍된 I/O는 구현이 매우 간단하고 하드웨어 지원이 거의 필요하지 않지만 I/O 인터페이스의 지속적인 폴링은 프로세서가 쓸모없는 작업을 수행하는 데 바쁘게 합니다.

인터럽트 구동 I/O는 이러한 단점을 극복하고 주변 장치에서 이벤트가 발생할 때만 프로그램이 반응하도록 합니다. 이 방식에서 주변 장치는 타이머 오버플로, UART 인터페이스에서 문자 수신, 버튼 전환 등과 같은 일부 이벤트가 발생할 때 프로세서에 신호 (**인터럽트**라고 함)를 보내는 역할을 합니다. 이벤트가 없을 때 (즉, 인터럽트가 없음) 프로세서는 유용한 작업을 계속합니다. 프로세서가 인터럽트를 수신하면 실행 중이던 프로그램을 중단하고 인터럽트 처리기라고도 하는 ISR (인터럽트 서비스 루틴)을 호출합니다. ISR은 본질적으로 인터럽트를 처리하는 void 인수가 있는 함수입니다. 즉, 버튼의 새 값을 읽고 타이머 오버플로와 관련된 작업을 수행합니다. 프로세서는 일반적으로 단일 및 다중 벡터 모드를 지원합니다. 단일 벡터 모드 (그림 1)에서는 모든 인터럽트가 동일한 ISR을 호출합니다. 따라서 인터럽트가 발생하면 프로세서는 메인 프로그램을 중단하고 공통 ISR로 점프하여 먼저 인터럽트 소스를 결정한 다음 식별된 인터럽트 원인에 해당하는 특정 ISR 코드를 실행합니다. 다중 벡터 모드 (그림 2)에서 각 인터럽트는 다른 ISR을 호출합니다. 따라서 인터럽트가 발생하면 인터럽트의 원인을 먼저 파악한 다음, 확인된 원인에 해당하는 ISR로 프로그램이 점프합니다.

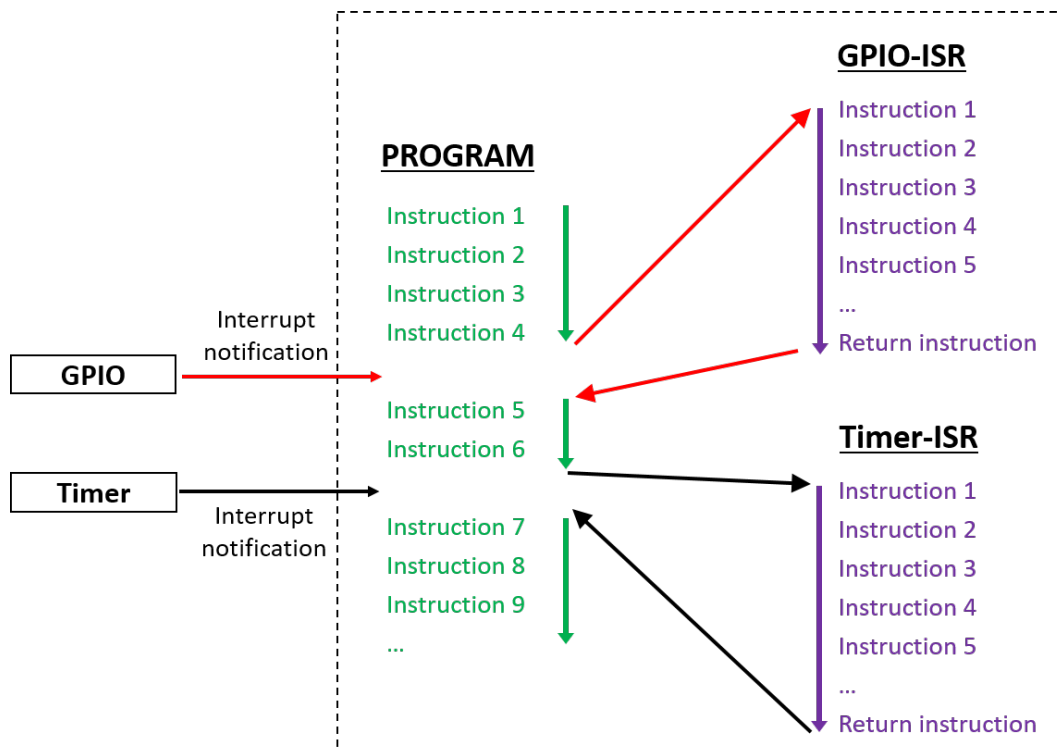


그림 1. 단일 벡터 모드에서 2 개의 인터럽트가 있는 예

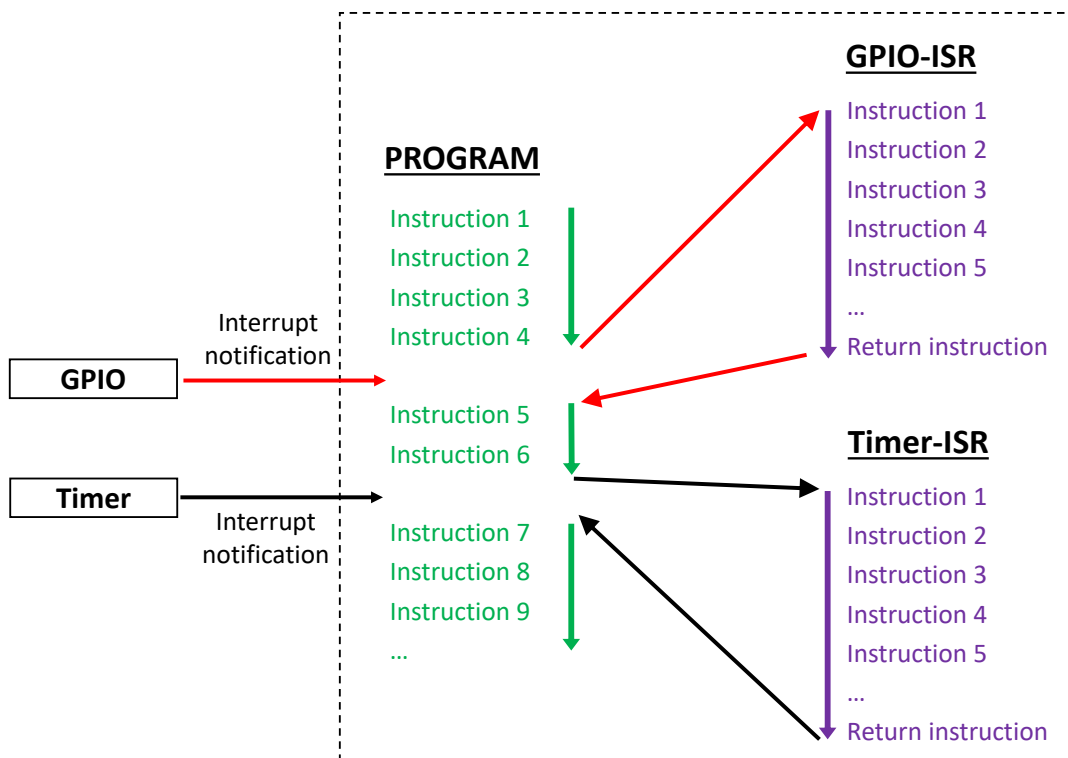


그림 2. 다중 벡터 모드에서 2 개의 인터럽트가 있는 예

프로세서는 일반적으로 인터럽트의 우선 순위를 지정할 수 있습니다. 더 높은 우선 순위의 인터럽트가 먼저 처리될 뿐만 아니라 더 높은 우선 순위의 인터럽트는 처리되는 프로세스에 있던 낮은 우선 순위의 인터럽트를 선점합니다. 예를 들어, 버튼 인터럽트가 우선 순위 5로 설정되고 타이머 인터럽트가 우선 순위 7로 설정되고 임계 값이 4로 설정되어 있다고 가정합니다 (두 우선 순위 모두 임계 값보다 높음). 프로그램이 정상적인 흐름을 실행하고 버튼을 누르면 인터럽트가 발생하고 프로세서는 ISR을 호출하여 버튼에서 데이터를 읽고 처리합니다. 버튼 ISR이 활성화된 동안 타이머가 오버플로 되면 프로세서가 타이머 오버플로를 즉시 처리할 수 있도록 ISR 자체가 중단됩니다. 완료되면 메인 프로그램(program¹)으로 돌아 가기 전에 인터럽트를 완료하기 위해 돌아갑니다.

3. SWERV EH1 에서 제공하는 프로그래밍 가능한 인터럽트 컨트롤러

SweRV EH1 코어는 다음 참조에서 설명하고 아래에 요약된 대로 인터럽트를 지원합니다.

- **[PRM v1.7]** 개정판 1.7 (2020 년 6 월 25 일), 6 장, "RISC-V SweRV EH1 프로그래머 참조 설명서", https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf
- **[ISM v1.11]** 버전 1.11- 초안 (2018 년 12 월 1 일), 7 장 "RISC-V 명령어 세트 매뉴얼 – 볼륨 II: Privileged Architecture ", <https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20181201-2650e2a>

SweRV EH1 코어 ([PRM v1.7] 참조)의 외부 인터럽트는 주로 RISC-V PLIC (플랫폼 레벨 인터럽트 컨트롤러) 사양 ([ISM v1.11] 참조)을 따라 모델링 됩니다. 그러나 인터럽트 컨트롤러는 플랫폼이 아닌 코어와 관련됩니다. 따라서 보다 일반적인 용어인 PIC (Programmable Interrupt Controller)는 SweRV EH1 코어에서 사용할 수 있는 컨트롤러를 지칭하는 데 사용됩니다. PIC 는 다음과 같은 주요 기능을 제공합니다.

- 최대 255 개의 외부 인터럽트 소스 지원 (1 (가장 높은 우선 순위)에서 255 (가장 낮은 우선 순위)) 각 소스에는 자체 활성화가 있습니다.
- 소스 번호 지정 외에도 15 개의 추가 우선 순위 수준을 제공합니다. 1-15 (1 이 가장 낮은 우선 순위) 또는 0-14 (14 가 가장 낮은 우선 순위)의 두 가지 우선 순위 체계를 사용할 수 있습니다. 각 소스에 우선 순위를 지정할 수 있습니다.
- 우선 순위가 낮은 인터럽트를 비활성화하기 위해 프로그래밍 가능한 우선 순위 임계 값을 지원합니다.
- 벡터화된 외부 인터럽트, 인터럽트 체인 및 중첩 인터럽트 지원.

그림 3 은 RVfpga 시스템의 인터럽트 시스템의 단순화된 버전을 보여줍니다. 인터럽트를 생성하는 모든 기능 단위를 외부 인터럽트 소스라고 합니다. **외부 인터럽트 소스**는 *_irq* (인터럽트 요청의 약어)로 끝나는 신호와 함께 비동기 신호를 **PIC** 에 전송하여 인터럽트 요청을 나타냅니다. 이 실습에서는 타이머와 GPIO 의 인터럽트를 사용하는 방법을 보여줍니다. 이러한 장치는 각각 *ptc_irq* 및 *gpio_irq* 신호를 사용하여 인터럽트를 생성합니다.

각 외부 인터럽트 소스는 인터럽트 요청을 코어의 클럭 도메인에 동기화하고 요청 신호를 PIC 를 위하여 공통 인터럽트 요청 형식 (즉, active low/high or level trigger)으로 변환하는 하드웨어 구조인 전용 게이트웨이 (PIC 내부에 위치)에 연결됩니다. PIC 는 한 번에 인터럽트 소스 당 하나의 인터럽트 요청만 처리할 수 있습니다. 보류 및 활성화된 모든 인터럽트 요청을 평가하고 가장 낮은 소스 ID 로 가장 높은 우선 순위의 인터럽트를 선택합니다. 그런 다음이 우선 순위를 프로그래밍 가능한 우선 순위 임계 값과 비교하고 중첩된 인터럽트를 지원하기 위해 현재 실행중인 인터럽트 처리기의 우선 순위를 비교합니다. 선택한 요청의 우선 순위가 두 임계 값보다 높으면 PIC 는 그림 1 (단일 벡터 모드) 및 그림 2 와 같이 기본 프로그램의 실행을 중단하고 해당 ISR 로 점프하는 인터럽트 알림을 코어로 보냅니다. (다중 벡터 모드).

¹ D. Harris and S. Harris. "Digital Design and Computer Architecture". Second Edition – 2012. Morgan Kaufmann Publishers (San Francisco, CA, United States). ISBN:978-0-12-394424-5.

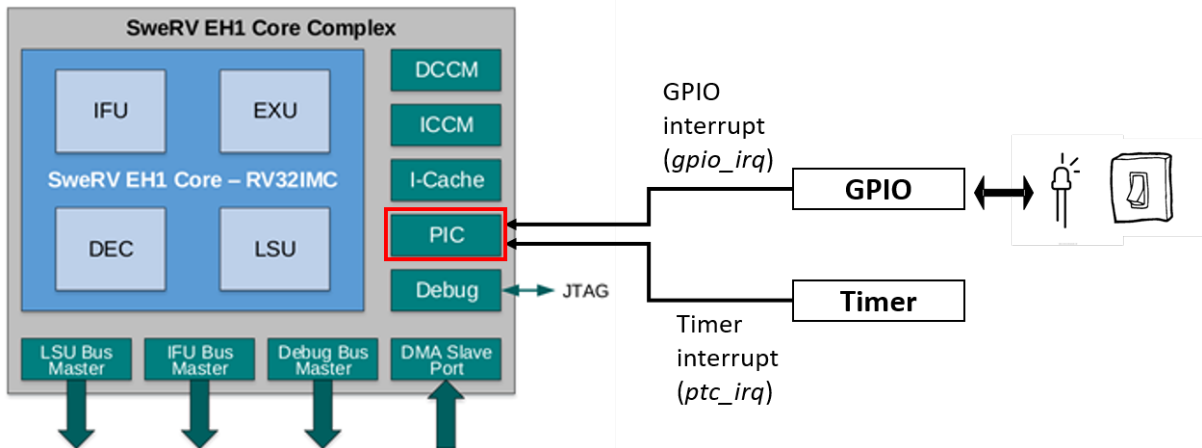


그림 3. RVfpga 시스템의 인터럽트 시스템

PIC의 주요 기능은 다음과 같은 기본 단계로 요약됩니다.

- 1) Enabling/Disabling(활성화/비활성화): PIC는 외부 인터럽트 활성화/비활성화를 허용합니다.
- 2) Configuration(구성): PIC는 극성 (액티브 하이/액티브 로우) 또는 유형 (에지 트리거/레벨 트리거)이 다른 외부 인터럽트를 수신하도록 구성할 수 있습니다. PIC는 또한 ISR을 다른 메모리 주소에 할당하는 것을 허용합니다.
- 3) Filtering and priority assignments (필터링 및 우선 순위 할당): PIC는 인터럽트에 우선 순위 수준을 할당할 수 있습니다. 주 프로그램이 실행 중일 때 PIC는 가장 높은 우선 순위 수준으로 활성화되고 트리거된 인터럽트를 선택합니다.
- 4) Notification (알림): PIC가 우선 순위가 가장 높은 인터럽트를 선택하면 선택한 인터럽트를 서비스하는 루틴으로 점프하기 위해 주 프로그램의 실행을 중지하도록 코어에 알립니다.
- 5) Pre-emption(선점): 중첩된 인터럽트가 활성화된 경우 우선 순위가 더 높은 다른 인터럽트가 서비스하는 인터럽트를 선점할 수 있습니다.

4. SweRV EH1에서 외부 인터럽트 구성

다른 주변기기와 마찬가지로 PIC는 로드/저장 명령을 통해 사용자가 액세스할 수 있는 메모리 매핑 레지스터를 사용하여 구성됩니다. 레지스터 레벨에서 인터럽트 시스템을 사용하는 것은 가능하지만 매우 복잡합니다.

다행스럽게도 WD의 PSP (프로세서 지원 패키지) 및 BSP (보드 지원 패키지)

(<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>)에는 인터럽트를 사용하여 프로그램을 구현하는 훨씬 간단한 접근 방식을 제공하는 여러 기능이 포함되어 있습니다. 표 1은 외부 인터럽트를 구성하는 데 필요한 주요 기능과 매크로를 설명합니다. 완전성을 위해 이 문서의 끝에 있는 부록에서는 사용 가능한 다양한 레지스터에 대한 설명과 레지스터 수준 구성 및 PIC 사용 단계를 제공합니다.

표 1. 외부 인터럽트를 구성하는 데 사용되는 기본 기능 및 매크로

Header	Description
void <code>psplInterruptsSetVectorTableAddress</code> (벡터 테이블 주소 준비

<code>void* pVectTable);</code>	
<code>void pspExternalInterruptSetVectorTableAddress(void* pExtIntVectTable);</code>	외부 인터럽트 벡터 테이블 주소 준비
<code>void bspInitializeGenerationRegister(u32_t uiExtInterruptPolarity)</code>	Generation-Register 를 초기 상태로 설정
<code>void bspClearExtInterrupt(u32_t uiExtInterruptNumber)</code>	외부 인터럽트를 생성하는 트리거 지우기
<code>void pspExtInterruptSetPriorityOrder(u32_t uiPriorityOrder);</code>	우선 순위 설정 (표준 또는 예약)
<code>void pspExtInterruptsSetThreshold(u32_t uiThreshold);</code>	PIC 에서 외부 인터럽트의 우선 순위 임계 값을 설정합니다.
<code>void pspExtInterruptsSetNestingPriorityThreshold(u32_t uiNestingPriorityThreshold);</code>	PIC 에서 외부 인터럽트의 중첩 우선 순위 임계 값을 설정합니다.
<code>void pspExtInterruptSetPolarity(u32_t uiIntNum, u32_t uiPolarity);</code>	지정된 인터럽트 라인의 극성 (액티브 하이 또는 액티브 로우)을 설정합니다.
<code>void pspExtInterruptSetType(u32_t uiIntNum, u32_t uiIntType);</code>	지정된 인터럽트 라인의 유형 (레벨 트리거 또는 에지 트리거)을 설정합니다.
<code>void pspExtInterruptClearPendingInt(u32_t uiIntNum);</code>	지정된 인터럽트 라인에 대해 보류중인 인터럽트 표시를 지웁니다.
<code>void pspExtInterruptSetPriority(u32_t uiIntNum, u32_t uiPriority);</code>	지정된 인터럽트 라인의 우선 순위를 설정합니다
<code>void pspExternalInterruptEnableNumber(u32_t uiIntNum);</code>	PIC 에서 지정된 인터럽트 라인을 활성화합니다.
<code>void pspInterruptsEnable(void);</code>	이전 상태에 관계없이 모든 권한 수준에서 인터럽트 활성화
<code>void pspInterruptsDisable(u32_t *pOutPrevIntState);</code>	인터럽트를 비활성화하고 각 권한 수준에서 현재 인터럽트 상태를 반환합니다.

ISR (인터럽트 서비스 루틴)의 예는 실습 후반부에 제공됩니다. 아래에 설명된 단계에 따라 표 1 의 기능을 기반으로 RVfpga 시스템 인터럽트를 구성합니다. PIC 를 구성하는 것 외에도 외부 인터럽트를 생성하는 주변 장치도 구성해야 합니다. (이는 예제 및 연습에서 사용된 각 주변 장치에 대해 나중에 설명됩니다.)

인터럽트 시스템의 기본 초기화:

- 다중 벡터 모드에서 외부 벡터 인터럽트 주소 테이블의 기본 주소를 설정합니다.
`pspInterruptsSetVectorTableAddress` 및 `pspExternalInterruptSetVectorTableAddress` 함수를 사용하십시오.
- 생성 레지스터를 초기 상태로 둡니다. `bspInitializeGenerationRegister` 함수를 사용하십시오.
- 외부 인터럽트 트리거가 해제되었는지 확인합니다. `bspClearExtInterrupt` 함수를 사용하십시오.
- 우선 순위 (기능 `pspExtInterruptSetPriorityOrder`), 임계 값 (기능 `pspExtInterruptsSetThreshold`) 및 중첩 우선 순위 임계 값 (기능 `pspExtInterruptsSetNestingPriorityThreshold`)에 대한 기본값을 설정합니다.

각 인터럽트 소스의 초기화 :

1. 각 인터럽트 소스에 대해 `pspExtInterruptSetPolarity` 및 `pspExtInterruptSetType` 함수를 사용하여 극성 (액티브 하이/액티브 로우) 및 유형 (레벨 트리거/에지 트리거)을 설정합니다.
2. `pspExtInterruptClearPendingInt` 함수를 사용하여 보류중인 인터럽트를 지웁니다.
3. `pspExtInterruptSetPriority` 함수를 사용하여 각 외부 인터럽트 소스에 대한 우선 순위 수준을 설정합니다.
4. `pspExternalInterruptEnableNumber` 함수를 사용하여 적절한 외부 인터럽트 소스에 대한 인터럽트를 활성화합니다.
5. 다중 벡터 모드에서 각 외부 인터럽트 소스에 대해 해당 핸들러의 주소를 외부 벡터 인터럽트 주소 테이블에 기록합니다.

상급 작업: 이러한 기본 기능에 대해 더 깊이 이해하려면 `.platformio/packages/framework-wd-riscv-sdk/psp` 에 있는 PSP 코드와 `.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp` 에 있는 BSP 코드를 확인하세요. 특히 흥미로운 것은 아래 나열된 파일이며, 그중 일부는 `api_inc` 하위 폴더에 포함되어 있습니다.

- `bsp_external_interrupts.h`: RVfpga 에서 `external_interrupts` 생성
- `psp_interrupts_eh1.h`: EH1 코어의 ISR 에 대한 정보 및 등록 API 를 제공합니다.
- `psp_ext_interrupts_eh1.h`: SweRV EH1 에 대한 psp 외부 인터럽트 인터페이스를 정의합니다.
- `psp_macros_eh1.h`: SweRV EH1 에 대한 psp 매크로를 정의합니다.
- `psp_csrs_eh1.h`: SweRV EH1 CSR 의 정의

또한, 이러한 함수 중 하나 이상을 레지스터 수준까지 분석하는 것이 좋습니다. 이를 위해 SweRV EH1 Core 의 PIC 가 레지스터 수준에서 외부 인터럽트를 구성하고 관리하는 방법을 설명하는 부록에 제공된 정보를 사용할 수 있습니다.

고급 작업: 또한, Western Digital (<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>)에서 제공하고, `[RVfpgaPath]/RVfpga/Labs/Lab9/WD_demo_external_int_Original` 에 있는 PlatformIO 프로젝트에서 외부 인터럽트 데모를 분석하고 실행하는 것이 좋습니다. 모든 것이 올바르게 작동하면 직렬 콘솔에 다음 메시지가 표시됩니다:

```
Hello from SweRV core running on NexysA7
Core list:
    EH1 = 11
    EL2 = 16
Running demo on core 11...
-----
SweRVolf version 255.255255 (SHA 000000ef) (dirty 128)
-----
External Interrupts tests passed successfully
```

5. 예제

이 섹션에서는 프로그래밍된 I/O 프로그램을 인터럽트 구동 I/O 프로그램으로 변환하는 예를 제공합니다. 프로그래밍 I/O (첫 번째 및 두 번째 예제)에 내재된 서로 다른 문제를 보여주는 세 가지 예제를 보여주고 인터럽트 구동 I/O 체계를 사용하여 이러한 문제를 쉽게 해결할 수 있는 방법을 보여줍니다 (세 번째 예제).

A. LED-Switch C-Language 프로그램

LED-Switch_C-Lang 프로그램 (그림 4 참조)은 맨 오른쪽 스위치에서 0→1 전환이 발생할 때마다 맨 오른쪽 LED 상태를 반전합니다. 이 프로그램은 아래에서 제공됩니다.

[RVfpgaPath] /RVfpga/Labs/Lab9/LED-Switch_C-Lang.c

주변기기 초기화 후 프로그램은 현재 스위치 상태를 이전 스위치 상태와 비교하는 무한 루프에 들어가며, 0→1 전환이 감지되면 LED 상태를 반전시킵니다. (1→0 전환이 발생하면 아무 일도 발생하지 않습니다.)

C로 작성된 이전 예제 및 연습에서는 I/O 레지스터 (READ_GPIO, READ_Reg, WRITE_GPIO, WRITE_Reg 등)에 액세스하기 위한 매크로를 정의했습니다. 이 예제에서는 대신 동일한 목적으로 PSP에 정의된 두 개의 매크로를 사용합니다: 인수로 제공된 32 비트 레지스터를 읽는 M_PSP_READ_REGISTER_32와 두 번째 인수로 제공된 값으로 32 비트 레지스터를 쓰는 M_PSP_WRITE_REGISTER_32. 이러한 매크로를 사용하려면 platformio.ini 파일에 framework = wd-riscv-sdk 줄 (대상으로 RVfpga를 사용하여 프로젝트를 만들 때 기본값임)과 #include "psp_api.h"를 프로그램 시작 부분에 표시합니다 (그림 4, 라인 1).

```

1  #include "psp_api.h"
2
3  #define GPIO_SWs    0x80001400
4  #define GPIO_LEDs    0x80001404
5  #define GPIO_INOUT  0x80001408
6
7  int main ( void )
8  {
9      int LED_state, Sw_current_state, Sw_previous_state;
10
11      /* Configure LEDs and Switches */
12      M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF);
13
14      /* Init states */
15      LED_state = 0;
16      M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
17      Sw_previous_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
18
19      while (1) {
20          /* Invert LED-0 when SW-0 goes high */
21          Sw_current_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
22          if(Sw_current_state==1 && Sw_previous_state==0){
23              LED_state = !LED_state;
24              M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
25          }
26          Sw_previous_state = Sw_current_state;
27      }
28
29      return(0);
30  }

```

그림 4. LED-Switch C-Language 프로그램

작업: LED-Switch C-Language 프로그램을 분석하여 자세히 이해하십시오. 필요한 경우 디버거를 사용하여 프로그램을 단계별로 분석할 수 있습니다.

프로그램은 올바르게 작동하지만 프로세서가 스위치/LED를 읽고 쓰는 것 외에는 아무것도 하지 않기 때문에 매우 비효율적입니다. 분명히 우리는 프로세서가 I/O 장치와 통신하는 것보다 더 많은 일을하기를 원합니다.

B. LED-Switch_7SegDispl_C-Lang 프로그램

이 두 번째 예인 LED-Switch_7SegDispl_C-Lang에서 프로그램은 두 번째 주변 장치인 7 세그먼트 디스플레이로 LED-Switch_C-Lang을 확장합니다. 이 프로그램은 두 가지 작업을 수행합니다.

- 첫 번째 예에서와 같이 맨 오른쪽 스위치에서 0→1 전환이 발생할 때마다 맨 오른쪽 LED를 반전합니다.
- 8 자리 7 세그먼트 디스플레이에 1 초에 한 번씩 증가하는 오름차순 카운트를 표시합니다. 단순성을 위해 for 루프를 사용하여 1 초 지연을 생성합니다 (연습 1에서는이 목적을 위해 Lab 8의 타이머를 사용합니다).

이 프로그램은 그림 5에서 볼 수 있으며 다음에서 찾을 수 있습니다.

[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_C-Lang.c

일부 초기화 후 프로그램은 현재 스위치 상태를 이전 스위치 상태와 비교하는 무한 루프에 들어가며 0→1 전환이 감지되면 LED 상태를 반전시킵니다. 그러면 8 자리 7 세그먼트 디스플레이에 표시된 값이 증가하고 지연이 발생합니다. 그림 5의 빨간색 상자를 참조하십시오.

```

1  #include "psp_api.h"
2
3  #define SegEn_ADDR    0x80001038
4  #define SegDig_ADDR   0x8000103C
5
6  #define GPIO_SWs      0x80001400
7  #define GPIO_LEDs     0x80001404
8  #define GPIO_INOUT    0x80001408
9
10 int main ( void )
11 {
12     int i, LED_state, Sw_current_state, Sw_next_state, count=0;
13
14     /* Configure LEDs and Switches */
15     M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF);
16
17     /* Configure 7-Seg Displays */
18     M_PSP_WRITE_REGISTER_32(0x80001038, 0x0);
19
20     /* Init states */
21     LED_state = 0;
22     M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
23     Sw_current_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
24
25     while (1) {
26         /* Invert LED-0 when SW-0 goes high */
27         Sw_next_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
28         if(Sw_current_state==0 && Sw_next_state==1){
29             LED_state = !LED_state;
30             M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
31         }
32         Sw_current_state = Sw_next_state;
33
34         /* Increase 7-Seg Displays */
35         M_PSP_WRITE_REGISTER_32(SegDig_ADDR, count);
36         count++;
37
38         /* Delay */
39         for(i=0;i<1000000;i++);
40     }
41
42     return(0);
43 }

```

그림 5. LED-Switch_7SegDispl_C-Lang 프로그램

작업: 자세히 이해하기 위해 *LED-Switch_7SegDispl_C-Lang* 프로그램을 분석하십시오. 필요한 경우 디버거를 사용하여 프로그램을 단계별로 분석할 수 있습니다.

이 경우 프로그램은 일부 상황에서 제대로 작동하지 않습니다. 예를 들어, 지연 루프 내에서 발생하는 0→1→0 스위치 전환은 감지되지 않습니다. 또한 이전 예제와 동일한 문제가 있습니다. 프로세서는 장치를 읽고/쓰기만 하거나 또는 지연을 생성하는 동안 항상 바깥입니다.

이러한 상황을 어떻게 개선할 수 있을까요? 정답은 **인터럽트 구동 I/O**입니다. 다음 예와 다음 섹션에서 제안하는 연습에서는 이러한 모든 문제를 해결하고 모든 상황에서 더 효율적이고 올바르게 작동하는 프로그램을 구현하는 방법을 보여줍니다.

C. LED-Switch_7SegDispl_Interrupts_C-Lang 프로그램

이 마지막 예제 (*[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_Interrupts_C-Lang.c*)에서는 인터럽트 구동 I/O를 사용하여 맨 오른쪽 스위치의 상태를 읽는 방법을 보여줍니다. 이 방법을 사용하면 지연 루프 중에 발생하는 스위치 전환이 누락된 프로그램 문제를 해결합니다. 그러나 지연 루프에서 프로세서를 사용하는 문제는 여전히 지속됩니다. (이 문제는 연습 1에서 다룰 것입니다.)

그림 7에 표시된 새로운 **main** 기능은 다음 작업을 수행합니다.

- 인터럽트 시스템 초기화:
 - o 인터럽트의 기본 초기화: 그림 8에 표시된 `DefaultInitialization` 함수를 호출합니다

(119 행).

- `pspExtInterruptsSetThreshold(5)` 함수를 호출하여 특정 임계 값을 설정합니다 (120 행). 우선 순위가 이 임계 값보다 높지 않은 외부 인터럽트는 무시됩니다.
- 외부 인터럽트 라인 IRQ4 초기화:
- IRQ4 라인 초기화: 인터럽트 서비스 루틴으로서 우선 순위가 6 이고 `GPIO_ISR` 인 인터럽트 라인 4 에 대해 `ExternalIntLine_Initialization` 함수 (123 라인)를 호출합니다. 그림 9 에서 이 함수를 분석합니다.
 - IRQ4 를 GPIO 인터럽트 라인 (라인 124)과 연결합니다. 이는 워드 `0x80001018` 의 비트 0 을 설정하여 수행됩니다 (예제에서 `Select_INT` 로 태그 지정됨). 이 시스템 컨트롤러 메모리 매핑 레지스터는 2 비트를 포함합니다 (그림 6 참조); `irq_gpio_enable` 이라고하는 비트 0 이 1 로 설정된 경우 GPIO 인터럽트 라인을 IRQ4 와 연결하는 데 사용되고, `irq_ptc_enable` 이라고하는 비트 1 이 1 로 설정된 경우 타이머 인터럽트 라인을 IRQ3 와 연결하는 데 사용됩니다. 지금은 이 고급 기능을 알고 있으면 충분합니다. 나중에 연습 2 에서 Verilog 구현에 대해 자세히 설명하므로 해당 연습의 일부로 수정할 수 있습니다.

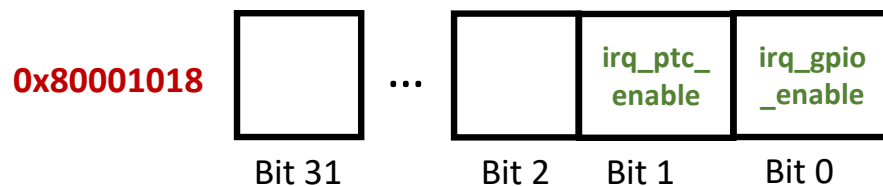


그림 6. RVfpga 시스템의 레지스터 0x80001018.

- 주변 장치를 초기화 합니다 (이 예에서는 GPIO 및 7 세그먼트 디스플레이).
 - o 127 행에서 GPIO_Initialization 함수를 호출합니다. 그림 10 에서 해당 함수를 분석합니다.
 - o 7 세그먼트 디스플레이를 활성화합니다 (128 행).
- 인터럽트 활성화 :
 - o 함수 pspInterruptsEnable (131 행) 및 매크로 M_PSP_SET_CSR (132 행)을 호출합니다. 상수 D_PSP_MIE_NUM 및 D_PSP_MIE_MEIE_MASK 는 WD 의 PSP 에 의해 정의됩니다.
- 마지막으로 7 세그먼트 디스플레이가 기록되고 영원히 반복되는 루프 내에서 지연을 설정합니다 (134-141 행).

```

114 int main(void)
115 {
116     int count=0, i;
117
118     /* INITIALIZE THE INTERRUPT SYSTEM */
119     DefaultInitialization(); /* Default initialization */
120     pspExtInterruptsSetThreshold(5); /* Set interrupts threshold to 5 */
121
122     /* INITIALIZE INTERRUPT LINE IRQ4 */
123     ExternalIntLine_Initialization(4, 6, GPIO_ISR); /* Initialize line IRQ4 with a priority of 6. Set GPIO_ISR as the Interrupt Service Routine */
124     M_PSP_WRITE_REGISTER_32(Select_INT, 0x1); /* Connect the GPIO interrupt to the IRQ4 interrupt line */
125
126     /* INITIALIZE THE PERIPHERALS */
127     GPIO_Initialization(); /* Initialize the GPIO */
128     M_PSP_WRITE_REGISTER_32(SegEn_ADDR, 0x0); /* Initialize the 7-Seg Displays */
129
130     /* ENABLE INTERRUPTS */
131     pspInterruptsEnable(); /* Enable all interrupts in mstatus CSR */
132     M_PSP_SET_CSR(D_PSP_MIE_NUM, D_PSP_MIE_MEIE_MASK); /* Enable external interrupts in mie CSR */
133
134     while (1) {
135         /* Increase 7-Seg Displays */
136         M_PSP_WRITE_REGISTER_32(SegDig_ADDR, count);
137         count++;
138
139         /* Delay */
140         for(i=0; i<500000000; i++);
141     }
142 }

```

그림 7. *main* 기능.

그림 8 에 표시된 ***DefaultInitialization*** 기능은 "중단 시스템의 기본 초기화" 항목 아래 섹션 4 에 설명된 단계를 수행합니다:

- 벡터 테이블 (53, 56 행)을 구성합니다. 이 예에서 배열 `G_Ext_Interrupt_Handlers` 는 벡터 테이블을 저장합니다.
- IRQ 를 트리거 하는데 사용되는 레지스터를 초기화합니다 (59 행).
- 라인 61-65 에서 모든 외부 인터럽트 (이 경우 IRQ3 및 IRQ4)를 지웁니다. 상수 `D_BSP_FIRST_IRQ_NUM` 및 `D_BSP_LAST_IRQ_NUM` 은 WD 의 BSP 에 의해 각각 3 과 4 로 정의됩니다.
- 기본 임계 값 및 우선 순위를 설정합니다 (68, 71 및 74 행). 다시 말하지만 이러한 함수에서 사용하는 상수는 WD 의 PSP 에 의해 정의됩니다.

```

48 void DefaultInitialization(void)
49 {
50     u32_t uiSourceId;
51
52     /* Register interrupt vector */
53     pspInterruptsSetVectorTableAddress(&M_PSP_VECT_TABLE);
54
55     /* Set external-interrupts vector-table address in MEIVT CSR */
56     pspExtInterruptSetVectorTableAddress(G_Ext_Interrupt_Handlers);
57
58     /* Put the Generation-Register in its initial state (no external interrupts are generated) */
59     bspInitializeGenerationRegister(D_PSP_EXT_INT_ACTIVE_HIGH);
60
61     for (uiSourceId = D_BSP_FIRST_IRQ_NUM; uiSourceId <= D_BSP_LAST_IRQ_NUM; uiSourceId++)
62     {
63         /* Make sure the external-interrupt triggers are cleared */
64         bspClearExtInterrupt(uiSourceId);
65     }
66
67     /* Set Standard priority order */
68     pspExtInterruptSetPriorityOrder(D_PSP_EXT_INT_STANDARD_PRIORITY);
69
70     /* Set interrupts threshold to minimal (== all interrupts should be served) */
71     pspExtInterruptsSetThreshold(M_PSP_EXT_INT_THRESHOLD_UNMASK_ALL_VALUE);
72
73     /* Set the nesting priority threshold to minimal (== all interrupts should be served) */
74     pspExtInterruptsSetNestingPriorityThreshold(M_PSP_EXT_INT_THRESHOLD_UNMASK_ALL_VALUE);
75 }

```

그림 8. *DefaultInitialization* 기능

그림 9 에 표시된 ***ExternalIntLine_Initialization*** 함수는 "각 인터럽트 소스의 초기화" 항목 아래 섹션 4 에 설명된 단계를 수행합니다.

- IRQ4 인터럽트의 유형과 극성을 구성하고 (이러한 기능에 사용되는 상수는 WD 의 PSP 에 의해 정의 됨) 해당 게이트웨이 (81, 84 및 87 행)에서 잠재적인 보류중인 인터럽트를 제거합니다.
- IRQ4 (90 행)의 우선 순위를 설정합니다.
- 93 번 라인의 PIC 에서 IRQ4 인터럽트를 활성화합니다.
- `G_Ext_Interrupt_Handlers` 배열에 저장된 벡터 테이블 (96 행)에 GPIO Interrupt Service Routine (GPIO_ISR)을 등록합니다.

```

78 void ExternalIntLine_Initialization(u32_t uiSourceId, u32_t priority, pspInterruptHandler_t pTestIsr)
79 {
80     /* Set Gateway Interrupt type (Level) */
81     pspExtInterruptSetType(uiSourceId, D_PSP_EXT_INT_LEVEL_TRIG_TYPE);
82
83     /* Set gateway Polarity (Active high) */
84     pspExtInterruptSetPolarity(uiSourceId, D_PSP_EXT_INT_ACTIVE_HIGH);
85
86     /* Clear the gateway */
87     pspExtInterruptClearPendingInt(uiSourceId);
88
89     /* Set IRQ4 priority */
90     pspExtInterruptSetPriority(uiSourceId, priority);
91
92     /* Enable IRQ4 interrupts in the PIC */
93     pspExternalInterruptEnableNumber(uiSourceId);
94
95     /* Register ISR */
96     G_Ext_Interrupt_Handlers[uiSourceId] = pTestIsr;
97 }

```

그림 9. *ExternalIntLine_Initialization* 함수

그림 10 에 표시된 ***GPIO_Initialization*** 함수는 다음 작업을 수행합니다:

- GPIO 핀을 입력/출력으로 구성하고 LED 를 0 으로 초기화합니다 (103 및 104 행).
- GPIO 인터럽트를 구성합니다. (각 GPIO 레지스터의 기능을 더 자세히 이해하려면 다음 위치에 있는 GPIO 코어 사양을 사용하십시오.)

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/docs/gpio_spec.pdf

- *RGPIO_INTE*: 인터럽트를 생성하는 범용 핀을 결정합니다 (라인 107).
- *RGPIO_PTRIG*: 인터럽트를 생성하는 에지를 결정합니다 (라인 108).
- *RGPIO_INTS*: 모든 핀의 인터럽트를 제거합니다 (라인 109).
- *RGPIO_CTRL*: 이 레지스터의 최하위 비트는 인터럽트 생성을 활성화합니다 (라인 110).

```

100 void GPIO_Initialization(void)
101 {
102     /* Configure LEDs and Switches */
103     M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF); /* GPIO_INOUT */
104     M_PSP_WRITE_REGISTER_32(GPIO_LEDS, 0x0); /* GPIO_LEDS */
105
106     /* Configure GPIO interrupts */
107     M_PSP_WRITE_REGISTER_32(RGPIO_INTE, 0x10000); /* RGPIO_INTE */
108     M_PSP_WRITE_REGISTER_32(RGPIO_PTRIG, 0x10000); /* RGPIO_PTRIG */
109     M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0); /* RGPIO_INTS */
110     M_PSP_WRITE_REGISTER_32(RGPIO_CTRL, 0x1); /* RGPIO_CTRL */
111 }

```

그림 10. *GPIO_Initialization* 기능.

마지막으로, 인터럽트가 GPIO 에서 트리거 될 때 ISR (즉, 그림 11 에 표시된 ***GPIO_ISR*** 함수)이 호출됩니다. 이 ISR (Interrupt Service Routine)은 다음 작업을 수행합니다.

- LED 의 현재 상태를 읽습니다 (35 행).
- LED 가 반전되고 마스킹 됩니다 (36-37 행).
- LED 가 새 값으로 기록됩니다 (38 행).
- GPIO 인터럽트가 해제되었습니다 (41 행).
- IRQ4 외부 인터럽트가 해제되었습니다 (44 행).

```

30 void GPIO_ISR(void)
31 {
32     unsigned int i;
33
34     /* Write the LED */
35     i = M_PSP_READ_REGISTER_32(GPIO_LEDS);          /* RGPI0_OUT */
36     i = !i;                                           /* Invert the LEDs */
37     i = i & 0x1;                                     /* Keep only the right-most LED */
38     M_PSP_WRITE_REGISTER_32(GPIO_LEDS, i);          /* RGPI0_OUT */
39
40     /* Clear GPIO interrupt */
41     M_PSP_WRITE_REGISTER_32(RGPI0_INTS, 0x0);       /* RGPI0_INTS */
42
43     /* Stop the generation of the specific external interrupt */
44     bspClearExtInterrupt(4);
45 }

```

그림 11. *GPIO_ISR* 함수.

작업: *LED-Switch_7SegDispl_Interrupts_C-Lang* 프로그램을 분석하여 자세히 이해하십시오. 섹션 4의 설명과 비교할 수 있으며 필요한 경우 디버거를 사용하여 프로그램을 단계별로 분석할 수 있습니다.

6. 실습

연습 1. *LED-Switch_7SegDispl_Interrupts_C-Lang* 프로그램을 수정하여 두 번째 인터럽트 소스 (이 경우 타이머에 의해 생성됨)를 포함합니다. 타이머는 PWM 생성기, 타이머 또는 카운터 역할을 할 수 있으므로 일반적으로 PTC 장치라고 합니다.

- RVfpga 시스템에서 타이머 인터럽트는 워드 0x80001018의 비트 1 (*irq_ptc_enable*)을 설정하여 IRQ3에 연결됩니다 (그림 6 참조).
- 이전 예제의 *GPIO_Initialization*과 유사하게 PTC 인터럽트를 초기화하는 함수를 생성합니다.
- *PTC_ISR*이라는 두 번째 ISR을 생성합니다. *LED-Switch_7SegDispl_Interrupts_C-Lang* 프로그램의 *GPIO_ISR*과 유사해야 하지만 대신 IRQ3를 사용하여 호출해야 합니다. *PTC_ISR*은 타이머 인터럽트를 처리하고 지워야 합니다.

프로그램이 구현되고 디버깅되면 PSP 함수 *pspExtInterruptsSetThreshold(threshold)* 및 *pspExtInterruptSetPriority(interrupt_source, priority)*를 사용하여 우선 순위와 임계값의 다양한 조합을 분석합니다. 실행 시간에 우선 순위를 변경할 수도 있습니다. 예를 들어, 7 세그먼트 디스플레이 카운트를 최대 10까지 표시한 다음 적절한 외부 인터럽트 소스의 우선 순위를 수정하여 카운트를 중지할 수 있습니다.

연습 2. 온 보드 푸시 버튼 (GPIO2)을 제어하기 위해 Lab 6에서 설계한 두 번째 GPIO에서 오는 세 번째 인터럽트 소스를 포함하도록 RVfpgaNexys를 수정합니다. 이 연습을 완료하는 데는 두 가지 방법이 있습니다.

- GPIO2 인터럽트를 사용하지 않는 외부 인터럽트 소스에 연결할 수 있습니다. SweRV EH1은 최대 255개의 서로 다른 인터럽트 라인을 제공하며 지금까지는 그중 2개만 사용했습니다. 이 접근 방식의 단점은 WD의 라이브러리를 수정해야 한다는 것입니다.
- GPIO2 인터럽트를 IRQ4에 연결하여 GPIO 모듈 (LED 및 스위치에 연결) 및 GPIO2 (푸시 버튼에 연결)가 단일 벡터 인터럽트 모드를 사용하도록 할 수 있습니다. 일부 상황에서는 다중 벡터 모드가 선호되지만 이 방법의 장점은 BSP를 재사용할 수 있다는 것입니다.

RVfpga 시스템에서 낮은 수준의 인터럽트 구현에 대한 세부 정보를 제공하여 두 번째 방법에 대한 지침을 제공합니다.

그림 12 는 다양한 인터럽트 소스 (GPIO 인터럽트, 타이머 인터럽트 – 그리고 여기서 분석하거나 사용하지 않는 SweRVolf 코어에서 원래 사용 가능한 인터럽트 소스)를 *IRQ4* 및 *IRQ3*와 연결하는 회로를 보여줍니다.

특히 *IRQ4*는 *irq_gpio_enable* = 1 (그림 6) 일 때 GPIO 에 연결되는 반면 *IRQ3*는 *irq_ptc_enable* = 1 일 때 타이머와 연결됩니다 (그림 6).

irq_gpio_enable = *irq_ptc_enable* = 0 인 경우, *IRQ4* 및 *IRQ3* 는 이 실습에서 사용하지 않는 SweRVolf 의 원래 인터럽트 소스와 연결되어 있습니다. (이러한 인터럽트 소스 사용에 관심이 있는 경우 <https://github.com/chipsalliance/Cores-SweRVolf>)에서 자세한 정보를 볼 수 있습니다.

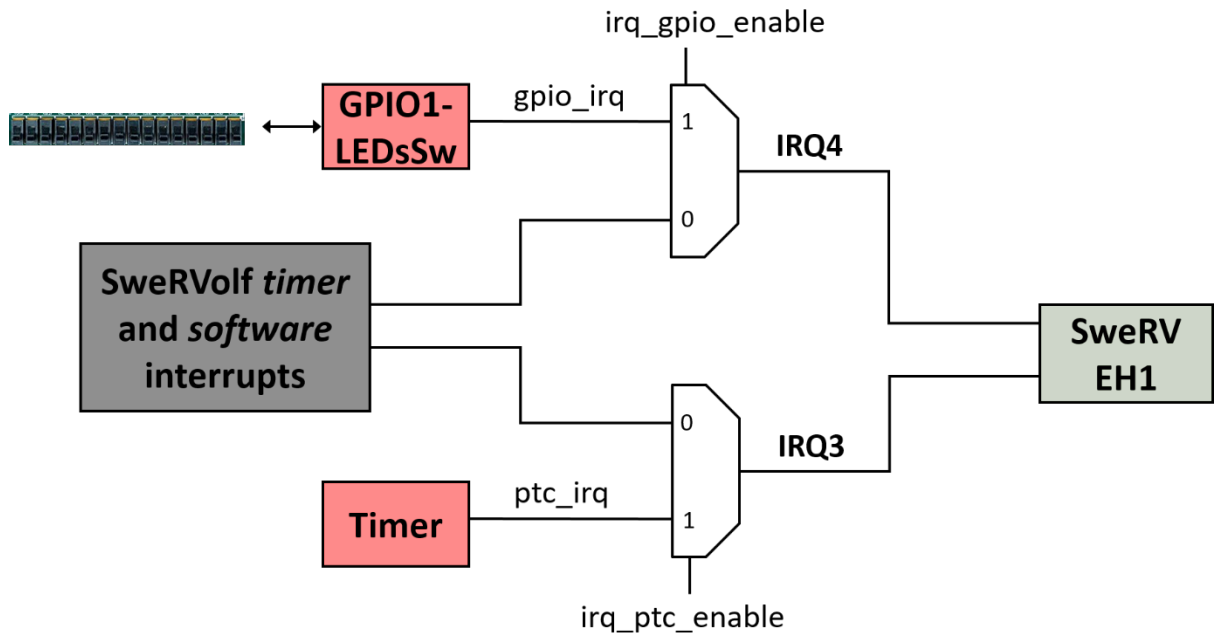


그림 12. 로직 구현: 각각 *IRQ4* 및 *IRQ3* 와 GPIO 및 타이머 인터럽트 연결

그림 13 은 인터럽트 소스와 *IRQ4* 및 *IRQ3* 간의 연결을 구현하는 모듈 **swervolf_core** 의 Verilog 영역을 보여줍니다. GPIO 인터럽트는 *irq_gpio_enable* 신호가 1 (빨간색 상자의 상단 부분) 일 때 *IRQ4*와 연결됩니다. 타이머 인터럽트는 *irq_ptc_enable* 신호가 1 (빨간색 상자 하단) 일 때 *IRQ3*에 연결됩니다. 두 신호가 모두 0 (그림에서 강조 표시되지 않은 코드)이면 SweRVolfX 에서 구현된 인터럽트 소스가 *IRQ3* 및 *IRQ4*에 연결됩니다.

```

123 always @(posedge i_clk) begin
124     o_wb_ack <= i_wb_cyc & !o_wb_ack;
125
126     nmi_int <= 1'b0;
127     nmi_int_r <= nmi_int;
128
129     // GPIO Interrupt through IRQ4. Enable by setting bit 0 of word 0x80001018
130     if (irq_gpio_enable & gpio_irq) begin
131         sw_irq4 <= 1'b1;
132     end
133
134     // Timer (PTC) Interrupt through IRQ3. Enable by setting bit 1 of word 0x80001018
135     if (irq_ptc_enable & ptc_irq) begin
136         sw_irq3 <= 1'b1;
137     end
138
139     // SweRVolf simple timer and software interrupts. Enable by resetting bits 0 and 1 of word 0x80001018
140     if (!irq_gpio_enable & !irq_ptc_enable) begin
141
142         if (sw_irq3 edge)
143             sw_irq3 <= 1'b0;
144         if (sw_irq4 edge)
145             sw_irq4 <= 1'b0;
146
147         if (irq_timer_en)
148             irq_timer_cnt <= irq_timer_cnt - 1;
149
150         if (irq_timer_cnt == 32'd1) begin
151             irq_timer_en <= 1'b0;
152             if (sw_irq3_timer)
153                 sw_irq3 <= 1'b1;
154             if (sw_irq4_timer)
155                 sw_irq4 <= 1'b1;
156             if (!(sw_irq3_timer | sw_irq4_timer))
157                 nmi_int <= 1'b1;
158         end
159     end
160 end

```

그림 13. Verilog 구현: 빨간색으로 강조 표시, 각각 IRQ4 및 IRQ3 와 GPIO 및 타이머 인터럽트 연결.

이 연습에서는 그림 14 와 같이 *IRQ4*에 연결된 새 인터럽트 소스를 포함하도록 이전 구현 (그림 12)을 확장해야 합니다.

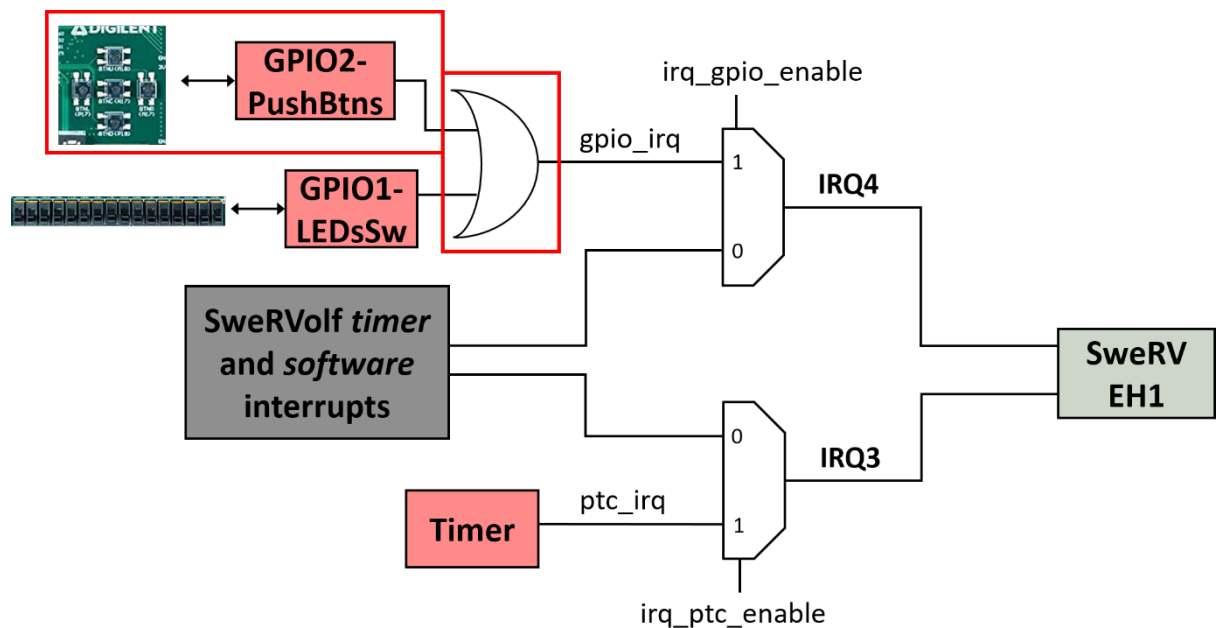


그림 14. 로직 구현: 두 번째 인터럽트 소스 (푸시 버튼을 읽는 GPIO 에서 제공)와 IRQ4 의 연결

이 예제에서는 수정할 필요가 없더라도 이해해야 할 몇 가지 다른 Verilog 영역을 강조합니다.

- 인터럽트 소스는 **swervolf_core** 모듈의 라인 599 에서 SweRV 프로세서에 삽입됩니다 (그림 15). 4 개의 인터럽트 소스를 사용할 수 있지만 이 실습에서는 *sw_irq4* 및 *sw_irq3* 소스에만 관심이 있습니다.

```
600      .extintsrc_req ({4'd0, sw_irq4, sw_irq3, spi0_irq, uart_irq}),
```

그림 15. SweRV 로 전송된 인터럽트 소스

- 활성화 신호 *irq_gpio_enable* 및 *irq_ptc_enable* (주소 0x80001018 에서 액세스 가능, 그림 6 참조)은 **swervolf_syscon** 모듈의 라인 192-196 에 있는 코어에서 작성합니다 (그림 16).

```
192      6: begin //0x18-0x1B
193          if (i_wb_sel[0])
194              irq_gpio_enable <= i_wb_dat[0];
195              irq_ptc_enable <= i_wb_dat[1];
196      end
```

그림 16. SweRV 코어에서 레지스터 0x80001018 쓰기

이러한 활성화 신호, *irq_gpio_enable* 및 *irq_ptc_enable* 은 코어의 **swervolf_syscon** 모듈에 의해 248-249 행에서 읽을 수 있습니다 (그림 17 참조).

```
248      //0x18-0x1B
249      6 : o_wb_rdt <= {30'd0, irq_ptc_enable, irq_gpio_enable};
```

그림 17. SweRV 코어로 레지스터 0x80001018 읽기

연습 3. 이전 연습에서 설계한 확장 RVfpgaNexys 버전을 사용하여 LED 에 점점 증가하는 이진 카운트를 표시하는 C 프로그램을 구현하여 1 부터 시작합니다. 인터럽트를 사용하여 타이머를 사용하여 각각의 표시 사이에 대기하는 지연을 만듭니다. 사람의 눈으로 값을 볼 수 있도록 증가된 값입니다. BTNC 를 읽어서 카운트 속도를 변경하고 스위치 [0]을 읽어서 누를 때마다 카운트를 다시 시작합니다.

연습 2 의 확장된 RVfpgaNexys 를 사용하면 이제 세 가지 인터럽트 소스를 사용할 수 있습니다.

- **GPIO (스위치에서 인터럽트)**
- **GPIO2 (이전 연습인 연습 2 에서 디자인 한 버튼에서 중단됨)**
- **PTC (타이머)**

연습 2 의 확장된 RVfpgaNexys 구현에 동일한 라인 (IRQ4)을 공유하는 두 개의 인터럽트 소스가 있는 경우 해당 인터럽트 서비스 루틴 (GPIO_ISR)은 인터럽트를 생성한 장치를 식별해야 합니다. GPIO 레지스터에서 해당 정보를 얻을 수 있습니다.

부록

이 부록에서는 SweRV EH1 Core 의 PIC (Programmable Interrupt Controller)가 레지스터 수준에서 외부 인터럽트를 관리하는 방법을 설명합니다. PIC 는 표 2 에 표시된 메모리 매핑 레지스터를 사용합니다. PIC 메모리 공간은 주소 0xF00C0000 에서 시작합니다. 이 주소를 *RV_PIC_BASE*라고합니다. 주소는 이 기본 주소를 기준으로 제공됩니다.

표 2. PIC 메모리 매핑 레지스터 주소 맵

Name	Addresses (relative to <i>RV_PIC_BASE</i>)	Description	Location at the manual
<i>meipIS</i>	$0x0004 - 0x0004 + S_{max} * 4 - 1$	외부 인터럽트 우선 순위 레지스터	Table 6-2 of [PRM v1.7]
<i>meipX</i>	$0x1000 - 0x1000 + (X_{max} + 1) * 4 - 1$	외부 인터럽트 보류 레지스터	Table 6-3 of [PRM v1.7]
<i>meieS</i>	$0x2000 - 0x2000 + S_{max} * 4 - 1$	외부 인터럽트 활성화 레지스터	Table 6-4 of [PRM v1.7]
<i>mpiccfg</i>	0x3000 – 0x3003	외부 인터럽트 PIC 구성 레지스터	Table 6-1 of [PRM v1.7]
<i>meigwctrlS</i>	$0x4004 - 0x4004 + S_{max} * 4 - 1$	외부 인터럽트 게이트웨이 구성 레지스터 (구성 가능한 게이트웨이 전용)	Table 6-11 of [PRM v1.7]
<i>meigwclrS</i>	$0x5004 - 0x5004 + S_{max} * 4 - 1$	외부 인터럽트 게이트웨이 클리어 레지스터 (구성 가능한 게이트웨이에서만 해당)	Table 6-12 of [PRM v1.7]

모든 레지스터는 32 비트 폭이며 메모리 매핑된 I/O 의 경우와 같이 로드 및 저장 명령어를 통해 액세스할 수 있습니다. 액세스 유형은 액세스하려는 특정 비트에 따라 다릅니다 ([PRM v1.7]에서 볼 수 있음).

일부 레지스터에는 S 또는 X 로 끝나는 매개 변수화된 이름이 있습니다. 이러한 레지스터의 여러 인스턴스가 존재할 수 있습니다. 매개 변수 S 는 외부 인터럽트 소스의 수를 나타내며 SweRV EH1 에서 게이트웨이 수와 동일합니다. 따라서 'S'로 끝나는 레지스터는 1 ~ 255 개의 레지스터 인스턴스를 사용할 수 있습니다. 이 실습에서는 **IRQ3** (타이머 관련) 및 **IRQ4** (GPIO 관련)의 두 가지 외부 인터럽트 소스만 사용합니다. 매개 변수 X 는 32 개의 게이트웨이 그룹을 나타냅니다.

이것은 게이트웨이가 그룹화 된다는 것을 의미하지는 않지만, 게이트웨이를 그룹화하면 외부 인터럽트 소스 그룹에 대한 작업을 수행하는 데 충분한 1 비트가 특정 32 비트 레지스터에 필요한 메모리 크기를 줄여 줍니다.

이는 외부 인터럽트 보류 레지스터의 경우이며, 1 비트는 인터럽트가 서비스되었는지 여부를 식별하기에 충분합니다. 이러한 레지스터에 대한 자세한 정보를 얻기 위해 표 1 의 맨 오른쪽 열은 비트 레벨 (특정 인터럽트) 설명이 포함된 [PRM v1.7] 내의 위치를 가리 킵니다.

표 2 에 표시된 레지스터 외에 PIC 에는 제어 및 상태 레지스터 (CSR)가 포함되어 있습니다. 표준 RISC-V ISA 는 최대 4,096 개의 CSR 에 대해 12 비트 인코딩 공간 (*csr [11:0]*)을 설정합니다. 관례적으로 CSR 주소 (*csr [11:8]*)의 상위 4 비트는 권한 수준에 따라 CSR 의 읽기 및 쓰기 접근성을 인코딩하는 데 사용됩니다. 상위 2 개 비트 (*csr [11:10]*)는 레지스터가 읽기/쓰기 (00, 01 또는 10)인지 읽기 전용 (11)인지를 나타냅니다. 다음 두 비트 (*csr [9:8]*)는 CSR 에 액세스할 수 있는 가장 낮은 권한 수준을 인코딩합니다. CSR 에 대한 자세한 내용은 [PRM v1.7]

및 [ISM v1.11]에서 확인할 수 있습니다. 표 3 은 SwerRV EH1 코어에서 외부 인터럽트를 관리하는 데 유용한 CSR 을 나열합니다. 이는 *csrrw* 또는 *csrrs* (CSR 읽기/쓰기 및 CSR 읽기/설정)와 같은 전용 로드 및 저장 명령을 통해 액세스할 수 있습니다.

표 3. PIC 비표준 RISC-V CSR 주소 맵.

Name	Number	Description	Location
meivt	0xBC8	외부 인터럽트 벡터 테이블 레지스터	Table 6-6 of [PRM v1.7]
meipt	0xBC9	외부 인터럽트 우선 순위 임계 값 레지스터	Table 6-5 of [PRM v1.7]
meicpct	0xBCA	외부 인터럽트 클레임 ID/우선 순위 캡처 트리거 레지스터	Table 6-8 of [PRM v1.7]
meicidpl	0xBCB	외부 인터럽트 클레임 ID 의 우선 순위 레지스터	Table 6-9 of [PRM v1.7]
meicurpl	0xBCC	외부 인터럽트 전류 우선 레벨 레지스터	Table 6-10 of [PRM v1.7]
meihap	0xFC8	외부 인터럽트 핸들러 주소 포인터 레지스터	Table 6-7 of [PRM v1.7]
mie	0x304	머신 인터럽트 활성화 레지스터	Table 11-1 of [PRM v1.7]
mstatus	0x300	머신 상태 레지스터	Figure 3.7 of [ISM v1.11]

표 3 의 맨 오른쪽 열은 주어진 CSR 에 대한 비트 레벨 정보가 설명된 [PRM v1.7] 또는 [ISM v1.11]의 위치를 가리 킵니다 (*mstatus* 비트 설명은 [PRM 에 제공되지 않음). v1.7] 대신 [ISM v1.11]에서).

A. 외부 인터럽트 구성

이 하위 섹션에서는 앞서 언급한 레지스터를 사용하여 외부 인터럽트를 구성하는 데 필요한 기본 단계를 요약합니다.

1. *mie* CSR 내에서 비트 *miep* 를 지워 모든 외부 인터럽트를 비활성화 합니다.
2. *mpiccfg* 레지스터의 *0/전* 비트를 작성하여 우선 순위를 구성합니다.
3. 다중 벡터 모드에서 구성되지 않은 경우 *meivt* 레지스터의 기본 필드를 작성하여 외부 벡터 인터럽트 주소 테이블의 기본 주소를 설정합니다.
4. *meipt* 레지스터의 *prithresh* 필드를 작성하여 우선 순위 임계 값을 설정합니다.
5. *meicidpl*의 *clidpri* 필드와 *meicurpl* 레지스터의 *currpri* 필드에 '0' (또는 반전된 우선 순위의 경우 '15')을 써서 중첩 우선 순위 임계 값을 초기화 합니다.
6. 구성 가능한 각 게이트웨이 S 에 대해 *meigwctrls* 레지스터에서 극성 (액티브 하이/액티브 로우) 및 유형 (레벨 트리거/에지 트리거)을 설정하고 게이트웨이의 *meigwctrls* 레지스터에 기록하여 IP 비트를 지웁니다.
7. 다중 벡터 모드에서 각 외부 인터럽트 소스 S 에 대해 외부 벡터 인터럽트 주소 테이블에 해당 핸들러의

주소를 기록합니다.

8. *meipIS* 레지스터의 해당 우선 순위 필드를 작성하여 각 외부 인터럽트 소스 *S*에 대한 우선 순위 레벨을 설정하십시오.
9. 각 인터럽트 소스 *S*에 대한 *meieS* 레지스터의 *inten* 비트를 설정하여 적절한 외부 인터럽트 소스에 대한 인터럽트를 활성화합니다.
10. *mstatus* CSR 내에서 *mei* 비트를 활성화합니다.
11. *mie* CSR 내에서 비트 *miepl*를 설정하여 모든 외부 인터럽트를 활성화합니다.

다음은 *S* 게이트웨이의 일반적인 단계입니다. 그러나 RVfpga 시스템에서는 각각 자체 게이트웨이가 있는 2개의 인터럽트 소스 (IRQ3 및 IRQ4) 만 사용합니다. 또한 일부 작업은 상호 교환이 가능하므로 순서가 완전히 엄격하지 않다는 점에 유의해야 합니다 (예: 2 단계 이전에 4 단계를 완료할 수 있음). 게다가 각 함수는 입력시 *psplInterruptsDisable* 을 호출하기 때문에 1 단계가 반드시 필요한 것은 아닙니다.

B. 외부 인터럽트 작동 모드

이 하위 섹션에서는 외부 인터럽트가 트리거되면 PIC가 어떻게 작동하는지 설명합니다. 외부 인터럽트 라인 (와이어)에서 원하는 이벤트가 발생하면 다음 작업이 수행됩니다.

1. PIC는 어떤 보류중인 인터럽트가 가장 높은 우선 순위를 갖는지 결정합니다.
2. 대상 hart (하드웨어 스레드)가 외부 인터럽트를 받으면 모든 인터럽트를 비활성화하고 (즉, RISC-V hart의 *mstatus* 레지스터에서 *mie* 비트를 지움) 외부 인터럽트 핸들러로 점프합니다.
3. 외부 인터럽트 핸들러는 *meicpct* 레지스터에 기록하여 보류중인 (*meihap* 레지스터에서) 우선 순위가 가장 높은 외부 인터럽트의 인터럽트 소스 ID와 해당 우선 순위 (*meicidpl* 레지스터에서)의 캡처를 트리거 합니다.
4. 그런 다음 핸들러는 *클레임* ID 필드에 제공된 인터럽트 소스 ID를 얻기 위해 *meihap* 레지스터를 읽습니다. *meihap* 레지스터의 내용에 따라 외부 인터럽트 핸들러는 이 외부 인터럽트 소스에 특정한 핸들러로 점프합니다. 이것은 그림 18에서 볼 수 있습니다.
5. 소스 별 인터럽트 핸들러 (ISR)가 외부 인터럽트를 처리한 후 다음을 수행합니다.:
 - a. 레벨 트리거 인터럽트 소스의 경우 인터럽트 핸들러는 인터럽트 요청을 시작한 SoC IP의 상태를 지웁니다.
 - b. 에지 트리거 인터럽트 소스의 경우 인터럽트 핸들러는 *meigwclrs* 레지스터에 기록하여 소스 게이트웨이의 IP 비트를 지웁니다.

이것은 소스의 인터럽트 요청을 비활성화 합니다.

6. 한편 백그라운드에서 PIC는 보류중인 인터럽트를 계속 평가합니다.

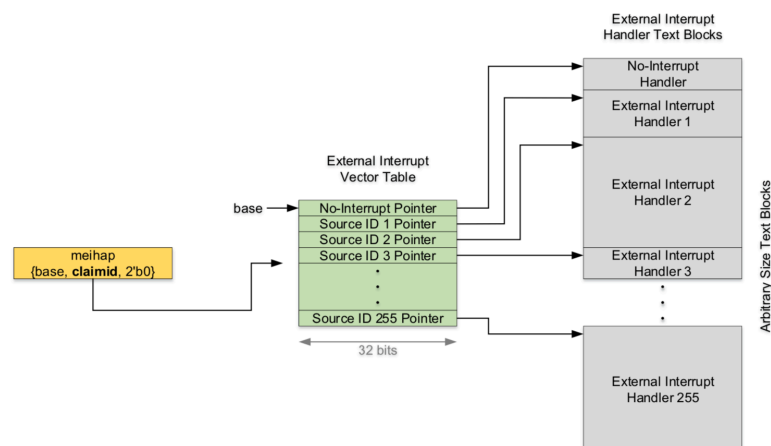


그림 18. 벡터화된 외부 인터럽트 ([PRM v1.7]에서 가져옴)

이것은 일반 작동 모드라는 점에 유의해야 합니다. 중첩 인터럽트 (최대 15 개)도 SweRV EH1 코어에서 지원됩니다. 자세한 내용은 [PRM v1.7]을 참조하세요.