



IMAGINATION大學計劃

RVfpga實驗9

中斷驅動I/O

1. 簡介

在本實驗中，我們介紹中斷的概念並展示如何在RVfpga上使用這些中斷。中斷可能由軟體或硬體產生。在本實驗中，我們重點介紹由物理引腳值變化而觸發的硬體中斷。具體來說，我們從第2節開始，描述程式化I/O和中斷驅動I/O之間的區別。然後，我們說明RVfpga系統的中斷控制器的操作，該控制器是SweRV EH1核新的一部分（第3部分）。在第4節中，我們介紹如何使用Western Digital的週邊設備支援套件（Peripherals Support Package，PSP）和開發板支援套件（Board Support Package，BSP）來配置外部中斷，PSP和BSP是包含硬體週邊設備驅動程式的軟體。最後，我們列出了幾個範例程式（第5部分），並提供了一些使用和擴展RVfpga系統硬體中斷的練習（第6部分）。

2. 程式化I/O與中斷驅動I/O

可通過以下幾種方法與週邊設備互動：程式化I/O、中斷驅動I/O和直接記憶體存取（Direct Memory Access，DMA）。在實驗2-8中，我們使用程式化I/O與週邊設備互動。使用程式化I/O時，使用者程式會連續輪詢I/O介面，並根據其狀態相應地做出回應。例如，實驗6中的基本練習使用程式化I/O連續輪詢（讀取）開關0和1，以控制由四個點亮LED組成的區塊速度和方向，這些點亮的LED重複地從一側移到另一側。程式化I/O很容易實作，而且只需要很少的硬體支援，但是對I/O介面的連續輪詢會使處理器忙於處理無用的工作。

中斷驅動I/O可避免此問題，使程式僅在週邊設備上發生事件時才做出回應。在此方法中，週邊設備負責在某些事件發生時向處理器傳送訊號（稱為中斷），例如計時器溢位、在UART介面上接收到字元、按鈕切換等事件。當沒有事件發生（即沒有中斷）時，處理器繼續執行有用的工作。當處理器接收到中斷時，它將停止正在執行的程式，並呼叫中斷服務常式（Interrupt Service Routine，ISR），也稱作中斷處理常式。ISR本質上是一個包含void引數的函數，可用於處理中斷 - 即讀取按鈕的新值、執行一些與計時器溢位相關的操作等。處理器通常支援單向量和多向量模式。在單向量模式下（圖1），所有中斷都呼叫相同的ISR。因此，當發生中斷時，處理器將暫停主程式並跳轉到通用ISR，後者首先確定中斷來源，然後執行與所確定的中斷原因相對應的特定ISR程式碼。在多向量模式下（圖2），每個中斷都會呼叫不同的ISR。因此，當產生中斷時，會首先確定中斷原因，程式隨後跳轉到與所確定的原因相對應的ISR。

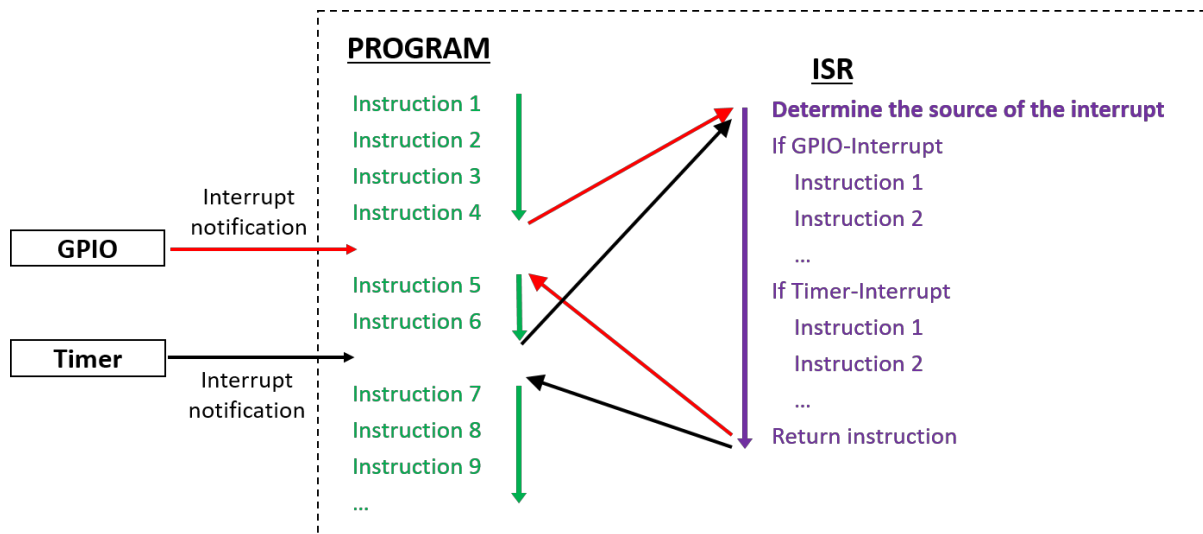


圖1. 單向量模式下2個中斷的範例

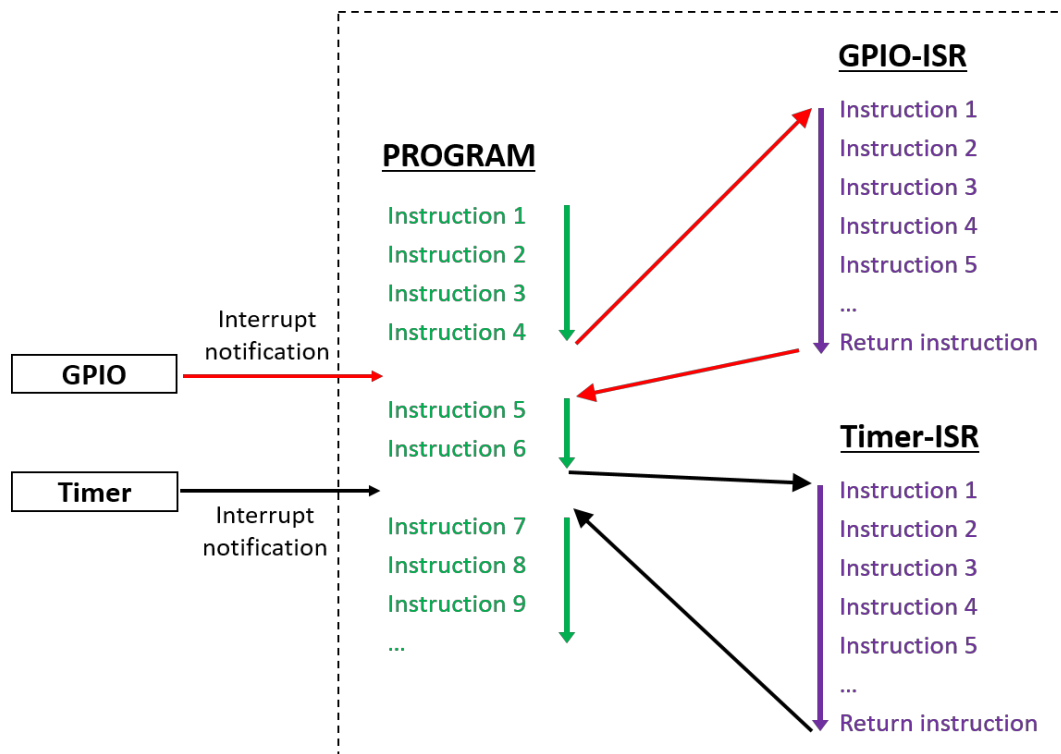


圖2. 多向量模式下2個中斷的範例

處理器通常允許對中斷進行優先處理。不僅將優先處理較高優先順序的中斷，而且較高優先順序的中斷將搶占正在處理的較低優先順序的中斷。例如，假設將按鈕中斷優先順序設定為5，將計時器中斷優先順序設定為7，並將閾值設定為4（因此兩個優先順序都高於閾值）。如果程式正在執行其正常流程，此時按下了按鈕，則將發生中斷，並且處理器呼叫ISR，ISR將從

按鈕讀取資料並加以處理。如果在按鈕ISR啟動時計時器溢位，則ISR本身將被中斷，以便處理器可以立即處理計時器溢位。處理完畢後，它將返回以完成按鈕中斷，然後返回主程式¹。

3. SWERV EH1提供的可程式化中斷控制器

SweRV EH1核心支援中斷，如以下參考資料中所述，總結如下：

- **[PRM v1.7]** 1.7版（2020年6月25日），《RISC-V SweRV EH1程式設計師參考手冊》第6章，下載位址：https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf
- **[ISM v1.11]** 1.11版初稿（2018年12月1日），《RISC-V指令集手冊 - 第II卷：特權架構》第7章，下載位址：<https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20181201-2650e2a>

SweRV EH1核心（請參閱[PRM v1.7]）的外部中斷在很大程度上是按照RISC-V PLIC（平台級中斷控制器）規格（請參閱[ISM v1.11]）建模的。但是，中斷控制器與核心而不是平台相關聯。因此，更通用的術語PIC（可程式化中斷控制器）用於指示SweRV EH1核心中可用的控制器。PIC提供以下主要特性：

- 支援多達255個外部中斷來源（從1（最高優先順序）到255（最低優先順序））；每個中斷來源都有其自己的啟用位元。
- 除來源編號外，還額外提供15個優先順序；有兩種優先方案可用：1-15（其中1是最低優先順序）或0-14（其中14是最低優先順序）。可以為每個來源分配一個優先順序。
- 提供對可程式化優先順序閾值的支援，以停用較低優先順序的中斷。
- 支援向量外部中斷、中斷鏈結和巢狀中斷。

圖3說明了RVfpga系統的中斷系統的簡化版本。所有產生中斷的功能單元都稱為**外部中斷來源**。外部中斷來源通過向**PIC**傳送異步訊號來指示中斷請求，且訊號以`_irq`（中斷請求的縮寫）結尾。在本實驗中，我們示範如何使用計時器和GPIO的中斷；這些單元分別使用訊號`ptc_irq`和`gpio_irq`產生中斷。

每個外部中斷來源都連接到專用閘道（位於PIC內部），該專用閘道是一種硬體結構，負責將中斷請求與核心的鐘域同步，並將請求訊號轉換為PIC的通用中斷請求格式（即，高電平/低電平有效或電平觸發）。PIC一次只能為每個中斷來源處理一個中斷請求。它會評估所有待處理的和啟用的中斷請求，並選擇具有最低來源ID的優先順序最高的中斷。然後，將此優先順序與可程式化的優先順序閾值進行比較，並且為了支援巢狀中斷，還會與中斷處理常式（如果目前正在執行）的優先順序進行比較。如果所選請求的優先順序高於這兩個閾值，則PIC將向核心傳送中斷通知，核心會停止主程式的執行並跳轉到相應的ISR，如圖1（單向量模式）和圖2（多向量模式）所示。

¹ D. Harris和S. Harris，《數位設計和電腦結構》，第二版 - 2012，Morgan Kaufmann出版社（美國加州舊金山），ISBN:978-0-12-394424-5。

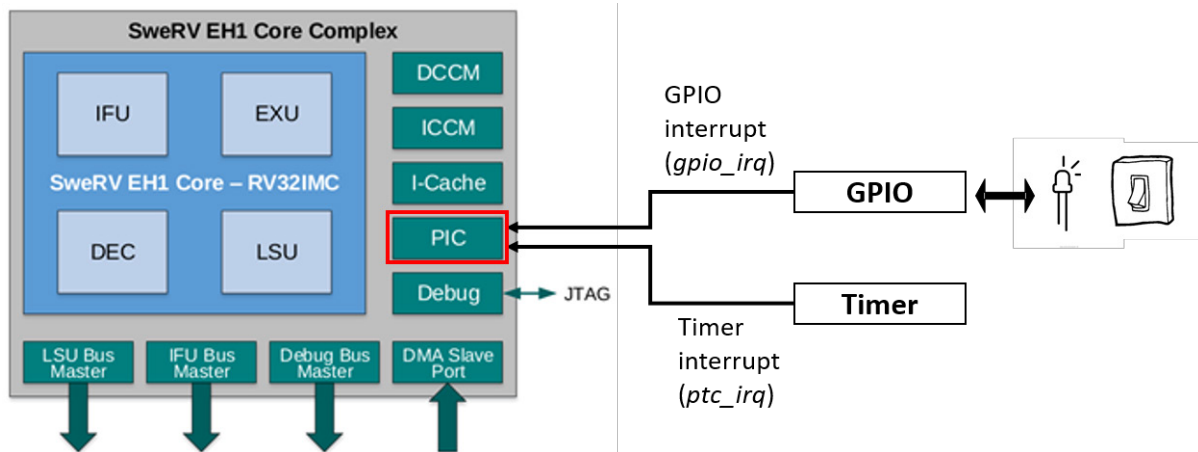


圖3. RVfpga系統的中斷系統

PIC的主要功能概括為以下幾個基本步驟：

- 1) 啟用/停用：PIC能夠啟用/停用外部中斷
- 2) 配置：可以將PIC配置為監聽具有不同極性（高電平有效/低電平有效）或類型（邊緣觸發/電平觸發）的外部中斷。PIC還允許將ISR分配給不同的記憶體位址。
- 3) 過濾和優先順序分配：PIC允許為中斷分配優先順序。當主程式執行時，PIC選擇已啟用的優先順序最高的觸發中斷。
- 4) 通知：PIC選擇了優先順序最高的中斷後，它將通知核心停止執行主程式，以便跳轉到服務選擇中斷的常式。
- 5) 搶佔：如果啟用巢狀中斷，則可以搶佔另一個具有更高優先順序的中斷服務的中斷。

4. 在SweRV EH1中配置外部中斷

與任何其他週邊設備類似，PIC使用記憶體映射暫存器進行配置，使用者可通過載入/儲存指令存取這些暫存器。可以在暫存器級別使用中斷系統，但該程序非常複雜；幸運的是，WD的處理器支援套件（Processor Support Package，PSP）和開發板支援套件（BSP）

（<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>）包括多個函數，這些函數提供了一種更簡單的方法來使用中斷實作程式。表1介紹了配置外部中斷所需的主要函數和巨集。為了完整起見，本文件末尾的附錄對可用的不同暫存器以及在暫存器級配置和使用PIC的步驟進行了說明。

表1. 用於配置外部中斷的基本函數和巨集

標題	說明
void pspInterruptsSetVectorTableAddress (void* pVectTable);	準備向量表位址
void pspExternalInterruptSetVectorTableAddress (void* pExtIntVectTable);	準備外部中斷向量表位址
void bspInitializeGenerationRegister (u32_t uiExtInterruptPolarity)	將產生暫存器置於其初始狀態
void bspClearExtInterrupt (u32_t uiExtInterruptNumber)	清除產生外部中斷的觸發器
void pspExtInterruptSetPriorityOrder (u32_t uiPriorityOrder);	設定優先順序（標準或預留）
void pspExtInterruptsSetThreshold (u32_t uiThreshold);	設定PIC中外部中斷的優先順序閾值
void pspExtInterruptsSetNestingPriorityThreshold (u32_t uiNestingPriorityThreshold);	設定PIC中外部中斷的巢狀優先順序閾值
void pspExtInterruptSetPolarity (u32_t uiIntNum, u32_t uiPolarity);	設定指定中斷線路的極性（高電平有效或低電平有效）
void pspExtInterruptSetType (u32_t uiIntNum, u32_t uiIntType);	設定指定中斷線路的類型（電平觸發或邊緣觸發）
void pspExtInterruptClearPendingInt (u32_t uiIntNum);	清除指定中斷線路的待處理中斷的指示
void pspExtInterruptSetPriority (u32_t uiIntNum, u32_t uiPriority);	設定指定中斷線路的優先順序
void pspExternalInterruptEnableNumber (u32_t uiIntNum);	啟用PIC中的指定中斷線路
void pspInterruptsEnable (void);	無論其先前狀態如何，都啟用中斷（在所有特權級別中）
void pspInterruptsDisable (u32_t *pOutPrevIntState);	在每個特權級別中停用中斷並返回目前中斷狀態

本實驗稍後將提供中斷服務常式（ISR）範例。這些程式根據表1中的函數按照以下步驟設定RVfpga系統中斷。注意，除了配置PIC外，還必須配置產生外部中斷的週邊設備（稍後將為範例和練習中使用的每個週邊設備對此進行描述）。

中斷系統的預設初始化：

1. 在多向量模式下，設定外部向量中斷位址表的基本位址。使用函數 `pspInterruptsSetVectorTableAddress` 和 `pspExternalInterruptSetVectorTableAddress`。
2. 將產生暫存器置於其初始狀態。使用函數 `bspInitializeGenerationRegister`。
3. 確保清除了外部中斷觸發器。使用函數 `bspClearExtInterrupt`。
4. 設定優先順序（函數 `pspExtInterruptSetPriorityOrder`）、閾值（函數 `pspExtInterruptsSetThreshold`）和巢狀優先順序閾值（函數 `pspExtInterruptsSetNestingPriorityThreshold`）的預設值。

每個中斷來源的初始化：

1. 對於每個中斷來源，使用函數`pspExtInterruptSetPolarity`和`pspExtInterruptSetType`設定極性（高電平有效/低電平有效）和類型（電平觸發/邊緣觸發）
2. 使用函數`pspExtInterruptClearPendingInt`清除所有待處理的中斷。
3. 使用函數`pspExtInterruptSetPriority`設定每個外部中斷來源的優先順序。
4. 使用函數`pspExternalInterruptEnableNumber`為適當的外部中斷來源啟用中斷。
5. 在多向量模式下，對於每個外部中斷來源，將相應處理常式的位址寫入外部向量中斷位址表中。

進階任務：為了更深入地瞭解這些基本函數，請檢視位於`.platformio/packages/framework-wd-riscv-sdk/psp`下的PSP程式碼和位於`.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp`下的BSP程式碼。需要特別注意的是下面列出的檔案，其中一些檔案包含在`api_inc`子資料夾中。

- `bsp_external_interrupts.h`：在RVfpga中產生外部中斷
- `psp_interrupts_eh1.h`：為EH1核心上的ISR提供資訊和註冊API
- `psp_ext_interrupts_eh1.h`：為SweRV EH1定義psp外部中斷介面
- `psp_macros_eh1.h`：為SweRV EH1定義psp巨集
- `psp_csrs_eh1.h`：SweRV EH1 CSR的定義

此外，還建議對其中至少一個函數進行低至暫存器級別的分析。為此，可以使用附錄中提供的資訊，這些資訊描述了SweRV EH1核心的PIC如何在暫存器級配置和管理外部中斷。

進階任務：我們還建議分析並執行Western Digital提供的外部中斷示範（網址為<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>），並且可以作為PlatformIO專案在以下位置獲得：`[RVfpgaPath]/RVfpga/Labs/Lab9/WD_demo_external_int_Original`。如果一切正常，則應在序列控制台中看到以下訊息：

```
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. 範例

在本節中，我們提供了將已程式化I/O程式轉換為中斷驅動I/O程式的範例。我們顯示了三個範例，說明了程式化I/O固有的不同問題（前兩個範例），然後指出了如何使用中斷驅動I/O方案輕鬆解決這些問題（第三個範例）。

A. LED-Switch C-Lang程式

每次最右側的開關上發生0→1跳變時，*LED-Switch_C-Lang*程式（請參閱圖4）都會反轉最右側LED的狀態。該程式位於：

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

週邊設備初始化之後，程式進入一個無限迴圈，該迴圈會將目前開關狀態與先前開關狀態進行比較，如果偵測到0→1跳變，則會反轉LED狀態（請注意，當發生1→0跳變時，不會有任何變化）。

在之前的範例和練習中，我們用C語言編寫了用於存取I/O暫存器（`READ_GPIO`、`READ_Reg`、`WRITE_GPIO`、`WRITE_Reg`等）的巨集。在本範例中，我們改為使用PSP中定義的兩個巨集來實現相同的目的：`M_PSP_READ_REGISTER_32`（該巨集讀取作為引數提供的32位元暫存器）和`M_PSP_WRITE_REGISTER_32`（該巨集向32位元暫存器寫入第二個引數中提供的值）。請記住，為了能夠使用這些巨集，必須在*platformio.ini*檔中新增行`framework = wd-riscv-sdk`（以RVfpga為目標建立專案時，這是為預設操作），以及在程式的開頭新增行`#include "psp_api.h"`（圖4，第1行）。

```

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  }

```

圖4. *LED-Switch_C-Lang*程式

任務：分析*LED-Switch_C-Lang*程式以瞭解其細節。如果需要，可以使用偵錯工具來逐步分析該程式。

該程式可以正常工作，但是效率非常低，因為處理器只能讀/寫開關/LED。我們顯然希望處理器不僅僅能夠與I/O設備通訊，而且還要執行其他操作。

B. LED-Switch 7SegDispl C-Lang程式

在第二個範例中，*LED-Switch_7SegDispl_C-Lang*程式在*LED-Switch_C-Lang*的基礎上擴展了第二個週邊設備：7段顯示器。該程式執行兩項任務：

- 與第一個範例相同，每次最右側的開關上發生0→1跳變時，該程式都會反轉最右側的LED。
- 它在8位7段顯示器中顯示升序計數，該計數大約每秒遞增一次。請注意，為簡便起見，我們使用for迴圈產生一秒延遲（在練習1中，將使用實驗8的計時器來實現此目的）。

可以在圖5中檢視此程式，此程式位於：

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

執行部分初始化後，程式將進入無限迴圈，該迴圈會將目前開關狀態與先前開關狀態進行比較，如果偵測到0→1跳變，則會反轉LED狀態。然後，8位7段顯示器上顯示的值遞增，並產生延遲。請參閱圖5中的紅框部分。

```

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     return(0);
42 }

```

圖5. *LED-Switch_7SegDispl_C-Lang*程式

任務：分析*LED-Switch_7SegDispl_C-Lang*程式以瞭解其詳細資訊。如果需要，可以使用偵錯工具來逐步分析該程式。

請注意，在這種情況下，該程式在某些情況下甚至無法正常執行。例如，將永遠不會偵測到延遲迴圈內發生的0→1→0開關跳變。此外，我們仍然遇到與上一個範例相同的問題：處理器始終忙於讀/寫裝置或產生延遲。

我們如何改善這些情況？解決方案是使用**中斷驅動I/O**。在下一節提供的範例和練習中，我們將展示如何解決所有這些問題，以及如何實作效率更高且在所有情況下都能正常執行的程式。

C. LED-Switch 7SegDispl Interrupts C-Lang程式

這是最後一個範例，在本範例（`[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_Interrupts_C-Lang.c`）中，我們展示如何使用中斷驅動I/O來讀取最右側開關的狀態。使用此策略可解決程式缺少延遲迴圈期間發生的開關跳變的問題。但請注意，使處理器忙於延遲迴圈的問題仍然存在。（將在練習1中解決此問題。）

新的**main**函數（如圖7所示）執行以下任務：

- 初始化中斷系統：
 - 中斷的預設初始化：呼叫函數`DefaultInitialization`（第119行），如圖8所示。
 - 通過呼叫函數`pspExtInterruptsSetThreshold(5)`（第120行）設定特定的閾值。優先順序不超過此閾值的外部中斷將被忽略。
- 初始化外部中斷線路IRQ4：
 - 初始化線路IRQ4：呼叫函數`ExternalIntLine_Initialization`（第123行），用於中斷線路4，優先順序為6，GPIO_ISR作為中斷服務常式。我們在圖9中分析此函數。
 - 將IRQ4與GPIO中斷線路（第124行）連接。具體方法是設定字0x80001018的位元0（在本範例中標記為`Select_INT`）。該系統控制器記憶體映射暫存器包含2位元（請參閱圖6）：位元0，稱為`irq_gpio_enable`，設定為1時用於將GPIO中斷線路與IRQ4連接；位元1，稱為`irq_ptc_enable`，設定為1時用於將計時器中斷線路與IRQ3連接。目前，瞭解這種高階功能已足夠；稍後在練習2中，我們將詳細說明Verilog實作，以便可以在練習中對其進行修改。

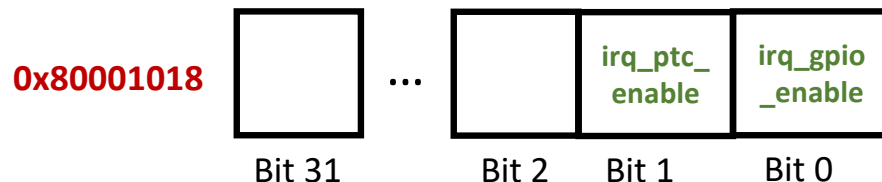


圖6. RVfpga系統的暫存器0x80001018

- 初始化週邊設備（在本範例中，為GPIO和7段顯示器）：
 - 呼叫函數`GPIO_Initialization`（第127行）。我們在圖10中分析此函數。
 - 啟用八個7段顯示器（第128行）。
- 啟用中斷：
 - 呼叫函數`pspInterruptsEnable`（第131行）和巨集`M_PSP_SET_CSR`（第132行）。常數`D_PSP_MIE_NUM`和`D_PSP_MIE_MEIE_MASK`由WD的PSP定義。
- 最後，寫入7段顯示器，並在永久重複的迴圈內建立延遲（第134-141行）。

```

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<50000000; i++);
141     }
142 }
143

```

圖7. *main*函數

DefaultInitialization函數（如圖8所示）執行第4節的項目「中斷系統的預設初始化」中所述的步驟：

- 配置向量表（第53和56行）。請注意，在本範例中，陣列 `G_Ext_Interrupt_Handlers` 儲存向量表。
- 初始化用於觸發IRQ的暫存器（第59行）。
- 清除第61-65行的所有外部中斷（在本範例中為IRQ3和IRQ4）。常數 `D_BSP_FIRST_IRQ_NUM` 和 `D_BSP_LAST_IRQ_NUM` 分別由WD的BSP定義為3和4。
- 建立預設閾值和優先順序（第68、71和74行）。同樣，這些函數使用的常數由WD的PSP定義。

```

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     pspExternalInterruptSetVectorTableAddress(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 }

```

圖8. *DefaultInitialization*函數

ExternalIntLine_Initialization函數（如圖9所示）執行第4節的項目「每個中斷來源的初始化」中所述的步驟：

- 配置IRQ4中斷的類型和極性（這些函數使用的常數由WD的PSP定義），並在相應的閘道處清除任何潛在的待處理中斷（第81、84和87行）。

- 設定IRQ4的優先順序（第90行）。
- 在PIC中啟用IRQ4中斷（第93行）。
- 將GPIO中斷服務常式（GPIO_ISR）記錄在向量表中（第96行），該向量表儲存在陣列G_Ext_Interrupt_Handlers中。

```

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 }

```

圖9. *ExternalIntLine_Initialization* 函數

GPIO_Initialization 函數（如圖10所示）執行以下任務：

- 將GPIO引腳配置為輸入/輸出，並將LED初始化為0（第103和104行）。
- 配置GPIO中斷。（要進一步瞭解每個GPIO暫存器的功能，請使用GPIO核心規格，可從以下位置獲得該規格：
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/docs/gpio_spec.pdf。）
 - o **RGPIO_INTE**：確定哪些通用引腳產生中斷（第107行）。
 - o **RGPIO_PTRIG**：確定產生中斷的邊緣（第108行）。
 - o **RGPIO_INTS**：清除所有引腳的中斷（第109行）。
 - o **RGPIO_CTRL**：該暫存器的最低有效位元啟用產生中斷（第110行）。

```

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 }

```

圖10. *GPIO_Initialization* 函數

最後，在GPIO上觸發中斷時呼叫ISR（即圖11所示的**GPIO_ISR**函數）。此ISR（中斷服務常式）執行以下任務：

- 讀取LED的目前狀態（第35行）。

- 反轉和屏蔽LED（第36-37行）。
- 向LED寫入新值（第38行）。
- 清除GPIO中斷（第41行）。
- 清除IRQ4外部中斷（第44行）。

```

30 void GPIO_ISR(void)
31 {
32     unsigned int i;
33
34     /* Write the LED */
35     i = M_PSP_READ_REGISTER_32(GPIO_LEDS);          /* RGPIO_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);          /* RGPIO_OUT */
39
40     /* Clear GPIO interrupt */
41     M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0);        /* RGPIO_INTS */
42
43     /* Stop the generation of the specific external interrupt */
44     bspClearExtInterrupt(4);
45 }

```

圖11. GPIO_ISR函數

任務：分析*LED-Switch_7SegDispl_Interrupts_C-Lang*程式以瞭解其詳細資訊。可以將實作與第4節的說明進行比較，並根據需要使用偵錯工具來逐步分析程式。

6. 練習

練習1. 修改*LED-Switch_7SegDispl_Interrupts_C-Lang*程式以包括第二個中斷來源，在本例中是由計時器產生的中斷來源。請記住，計時器可以用作PWM產生器、計時器或計數器，因此通常稱為PTC單元。

- 在RVfpga系統中，透過設定字0x80001018的位元1（*irq_ptc_enable*）可以中斷計時器到IRQ3的連接（參見圖6）。
- 建立一個初始化PTC中斷的函數，類似於前面範例中的GPIO_Initialization。
- 建立第二個ISR，稱為PTC_ISR。它應類似於*LED-Switch_7SegDispl_Interrupts_C-Lang*程式中的GPIO_ISR，但應使用IRQ3來呼叫。PTC_ISR應處理並清除計時器中斷。

實作並偵錯程式後，使用PSP函數pspExtInterruptsSetThreshold(threshold) 和 pspExtInterruptSetPriority(interrupt_source, priority) 來分析優先順序和閾值的不同組合。請注意，甚至可以在執行時變更優先順序；例如，可以在7段顯示器上顯示計數到10，然後通過修改相應外部中斷來源的優先順序停止計數。

練習2. 修改RVfpgaNexys以包含第三個中斷來源，該中斷來源來自實驗6中設計用於控制開發板上按鈕的第二個GPIO（GPIO2）。可通過兩種方法完成此練習：

- 可以將GPIO2中斷連接到未使用的外部中斷來源。SweRV EH1最多提供255條不同的中斷線路，目前為止，我們僅使用了其中的2條。這方法的缺點在於需要修改WD的函數庫。

- 可以將GPIO2中斷連接到IRQ4，以便GPIO模組（連接到LED和開關）和GPIO2（連接到按鈕）使用單向量中斷模式。盡管在某些情況下多向量模式更為可取，但這種方法的優點是可以重複使用BSP。

透過提供有關RVfpga系統中的中斷低階實作的一些詳細資訊，我們為第二種方法提供了一些指導。

圖12顯示了電路，該電路將各種中斷來源（GPIO中斷、計時器中斷和SweRVolf核心最初可用的中斷來源，我們在此不分析和使用這些中斷來源）與IRQ4和IRQ3連接起來。具體來說，當`irq_gpio_enable = 1`（圖6）時，IRQ4與GPIO連接；而`irq_ptc_enable = 1`（圖6）時，IRQ3與計時器連接。當`irq_gpio_enable = irq_ptc_enable = 0`時，IRQ4和IRQ3與SweRVolf原始中斷來源連接，在本實驗中，我們不使用這些中斷來源（如果您有興趣使用這些中斷來源，請造訪<https://github.com/chipsalliance/Cores-SweRVolf>來檢視詳細資訊）。

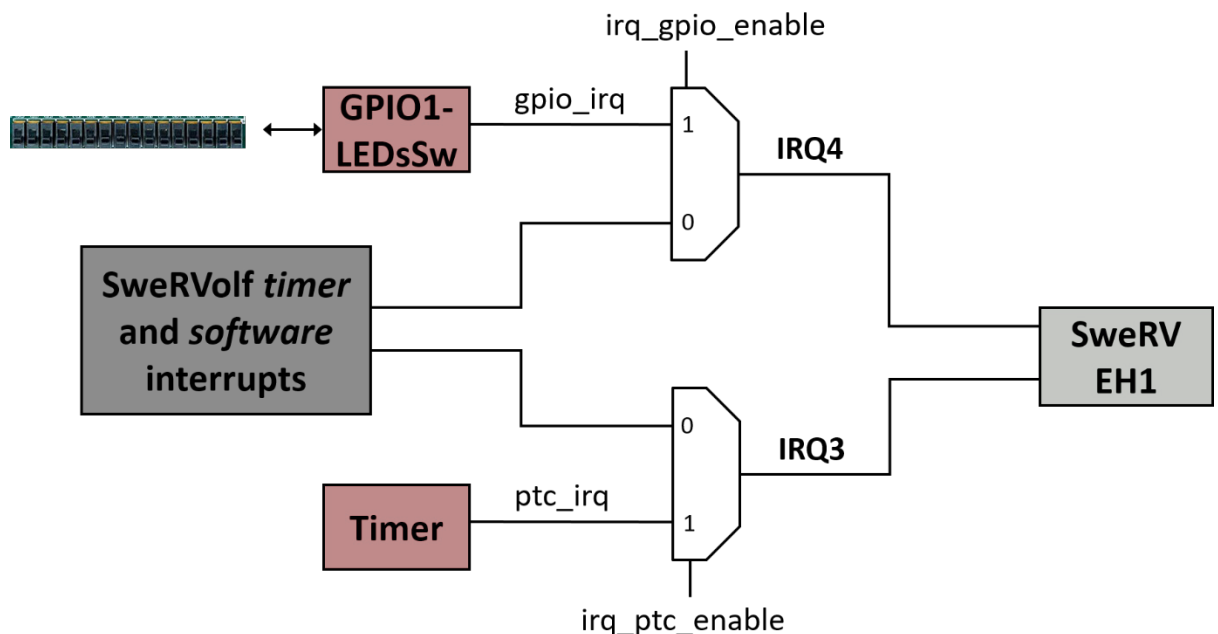


圖12. 邏輯實作：GPIO和計時器中斷分別與IRQ4和IRQ3的連接

圖13顯示了模組`swervolf_core`的Verilog區域，該模組實作中斷來源與IRQ4和IRQ3之間的連接。當訊號`irq_gpio_enable`為1時，GPIO中斷與IRQ4連接（紅框上部）。當訊號`irq_ptc_enable`為1時，計時器中斷與IRQ3連接（紅框下部）。當兩個訊號均為0（圖中未突出顯示的代碼）時，在SweRVolfX中實作的中斷來源將連接到IRQ3和IRQ4。


```

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

```

圖13. Verilog實作：GPIO和計時器中斷分別與IRQ4和IRQ3的連接，如紅色突出顯示部分。

在本練習中，必須擴展先前的實作（圖12）以包括一個新的中斷來源，該中斷來源連接到IRQ4，如圖14所示。

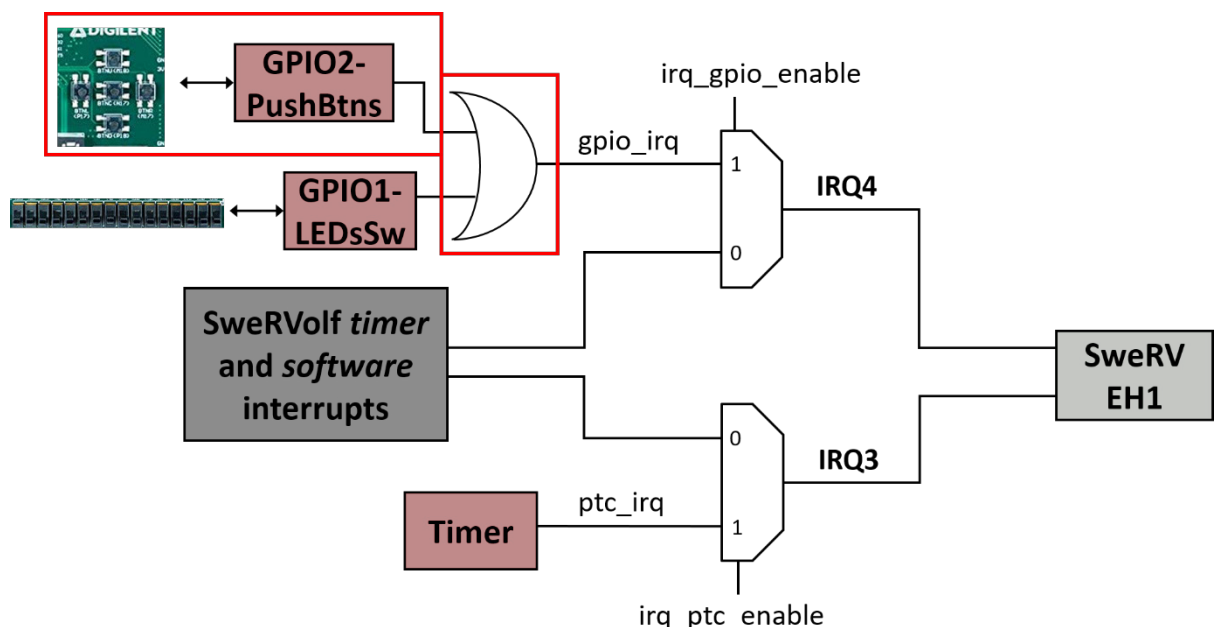


圖14. 邏輯實作：第二個中斷來源（由讀取按鈕的GPIO提供）與IRQ4連接

我們突出顯示了其他一些也需要瞭解的Verilog區域，但在本範例中無需對其進行修改。

- 在 **swervolf_core** 模組的第600行（圖15）將中斷來源插入SweRV處理器。儘管有四個中斷來源可用，但在本實驗中，我們僅關注中斷來源 **sw_irq4** 和 **sw_irq3**。

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

圖15. 中斷來源傳送到SweRV

- 在 **swervolf_syscon** 模組的第192-196行（圖16），通過核心寫入啟用訊號 **irq_gpio_enable** 和 **irq_ptc_enable**（可通過位址0x80001018存取，請參閱圖6）。

```
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
```

圖16. 從SweRV核心寫入暫存器0x80001018

在 **swervolf_syscon** 模組的第248-249行，從核心讀取啟用訊號 **irq_gpio_enable** 和 **irq_ptc_enable**（請參閱圖17）。

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

圖17. 將暫存器0x80001018讀取到SweRV核心中

練習3. 使用在前面練習所設計的擴展RVfpgaNexys版本來實作C程式，該程式在LED上從1開始顯示逐漸遞增的二進位計數。將中斷與計時器結合使用來產生延遲，以在各個遞增值的顯示之間留出等待時間，這樣人眼便可看到這些值。讀取BTNC並使用它來變更計數速度，讀取Switch[0]並使用它重新啟動計數（按下時）。

對於練習2中的擴展RVfpgaNexys，現在有三個可能的中斷來源：

- **GPIO**（來自開關的中斷）
- **GPIO2**（來自在練習2中設計的按鈕的中斷）
- **PTC**（計時器）

假定練習2中的擴展RVfpgaNexys實作有兩個中斷來源共享同一行（**IRQ4**），則相應的中斷服務常式（**GPIO_ISR**）必須識別產生中斷的裝置。可以從GPIO暫存器中獲取相關資訊。

附錄

本附錄介紹SweRV EH1核心的可程式化中斷控制器（Programmable Interrupt Controller，PIC）如何在暫存器級管理外部中斷。PIC使用表2所示的記憶體映射暫存器。必須注意的是，PIC記憶體空間的起始位址為0xF00C0000；此位址稱為RV_PIC_BASE。位址是相對於該基本位址所提供的。

表2. PIC記憶體映射暫存器位址映射

名稱	位址（相對於RV_PIC_BASE）	說明	在手冊中的位置
meipIS	$0x0004 - 0x0004 + S_{max} * 4 - 1$	外部中斷優先順序暫存器	[PRM v1.7]的表6-2
meipX	$0x1000 - 0x1000 + (X_{max} + 1) * 4 - 1$	外部中斷待處理暫存器	[PRM v1.7]的表6-3
meieS	$0x2000 - 0x2000 + S_{max} * 4 - 1$	外部中斷啟用暫存器	[PRM v1.7]的表6-4
mpiccfg	0x3000 – 0x3003	外部中斷PIC配置暫存器	[PRM v1.7]的表6-1
meigwctrlS	$0x4004 - 0x4004 + S_{max} * 4 - 1$	外部中斷閘道配置暫存器（僅適用於可配置的閘道）	[PRM v1.7]的表6-11
meigwclrS	$0x5004 - 0x5004 + S_{max} * 4 - 1$	外部中斷閘道清除暫存器（僅適用於可配置的閘道）	[PRM v1.7]的表6-12

所有暫存器均為32位元寬，並與記憶體映射I/O一樣可通過載入和儲存指令存取。存取類型取決於我們要存取的特定位元（可以在[PRM v1.7]中進行檢視）。

一些暫存器具有以S或X結尾的參數化名稱。這些暫存器可能存在多個實例。參數S是指外部中斷來源的數量，在SweRV EH1中等於閘道的數量。因此，以「S」結尾的暫存器具有1到255個可用的暫存器實例。在本實驗中，我們僅使用2個外部中斷來源：IRQ3（與計時器相關）和IRQ4（與GPIO相關）。參數X是指一組32個閘道。這並不意味著對閘道進行了分組，但是對其進行分組可以減少某些32位元暫存器所需的記憶體大小，其中1個位元便足以對一組外部中斷來源執行操作。外部中斷待處理暫存器就是這種情況，其中1個位元便足以區分是否已處理該中斷。為了獲得有關這些暫存器的更多資訊，表1的最右欄指出了[PRM v1.7]中包含位元級別（特定中斷）說明的位置。

除了表2中所示的暫存器外，PIC還包含控制和狀態暫存器（CSR）。標準RISC-V ISA建立了12位元編碼空間（*csr[11:0]*），最多支援4,096個CSR。按照慣例，CSR位址的高4位元（*csr[11:8]*）用於根據權限等級對CSR的讀寫可存取性進行編碼。前兩位元（*csr[11:10]*）指示暫存器是讀/寫（00、01或10）還是唯讀（11）。後兩位元（*CSR[9:8]*）對可以存取CSR的最低特權級別進行編碼。有關CSR的更多資訊，請參閱[PRM v1.7]和[ISM v1.11]。表3列出了有助於管理SweRV EH1核心中的外部中斷的CSR。可通過*csrrw*或*csrrs*（CSR讀/寫和CSR讀/設定）等專用載入和儲存指令存取這些CSR。

表3. PIC非標準RISC-V CSR位址映射

名稱	編號	說明	位置
meivt	0xBC8	外部中斷向量表暫存器	[PRM v1.7]的表6-6
meipt	0xBC9	外部中斷優先順序閾值暫存器	[PRM v1.7]的表6-5
meicpct	0xBCA	外部中斷宣告ID/優先順序擷取觸發暫存器	[PRM v1.7]的表6-8
meicidpl	0xBCB	外部中斷宣告ID的優先順序暫存器	[PRM v1.7]的表6-9
meicurpl	0xBCC	外部中斷目前優先順序暫存器	[PRM v1.7]的表6-10
meihap	0xFC8	外部中斷處理常式位址指標暫存器	[PRM v1.7]的表6-7
mie	0x304	機器中斷啟用暫存器	[PRM v1.7]的表11-1
mstatus	0x300	機器狀態暫存器	[ISM v1.11]的圖3.7

表3的最右欄指出了[PRM v1.7]或[ISM v1.11]中說明給定CSR的位元級別資訊的位置（注意，[PRM v1.7]中未提供*mstatus*位元說明，而[ISM v1.11]中提供了此說明）。

A. 外部中斷配置

在本小節中，我們總結了使用上述暫存器配置外部中斷所需的基本步驟：

1. 通過將*mie* CSR中的位元*miep*清零，停用所有外部中斷。
2. 通過寫入*mpiccfg*暫存器的*priord*位來配置優先順序。
3. 在多向量模式下，如果未配置，則通過寫入*meivt*暫存器的基本位址欄位來設定外部向量中斷位址表的基本位址。
4. 通過寫入*meipt*暫存器的*prithresh*欄位來設定優先順序閾值。
5. 通過向*meicidpl*暫存器的*clidpri*欄位和*meicurpl*暫存器的*currpri*欄位寫入「0」（或對於反轉的優先順序順序寫入「15」）來初始化巢狀優先順序閾值。
6. 對於每個可配置的閘道*S*，在*meigwctrlS*暫存器中設定極性（高電平有效/低電平有效）和類型（電平觸發/邊緣觸發），並通過寫入閘道的*meigwclrS*暫存器將IP位清零。
7. 在多向量模式下，對於每個外部中斷來源*S*，將相應處理常式的位址寫入外部向量中斷位址表中。
8. 通過寫入*meiplS*暫存器的相應優先順序欄位，設定每個外部中斷來源*S*的優先順序。
9. 通過設定每個中斷來源*S*的*meieS*暫存器的*inten*位元，為適當的外部中斷來源啟用中斷。
10. 激活*mstatus* CSR中的*mei*位。
11. 設定*mie* CSR中的位元*miep*來啟用所有外部中斷。

以上是用於S閘道的常規步驟。但是，在RVfpga系統中，我們僅使用2個中斷來源（IRQ3和IRQ4），每個中斷來源都有其自己的閘道。而且，需要注意的是，不必非要遵循上述順序，因為一些操作是可以互換的（例如，可以在步驟2之前完成步驟4）。此外，由於每個函數在輸入時都會呼叫`psplInterruptsDisable`，因此不一定要執行步驟1。

B. 外部中斷工作模式

在本小節中，我們介紹觸發外部中斷後PIC將如何工作。一旦在外部中斷線路（導線）上發生所需事件，就會執行以下操作：

1. PIC決定哪個待處理中斷具有最高優先順序。
2. 當目標hart（硬體執行緒）採用外部中斷時，它會停用所有中斷（即將RISC-V hart的`mstatus`暫存器中的`mie`位元清零），並跳轉到外部中斷處理常式。
3. 外部中斷處理常式寫入`meicpct`暫存器，以觸發對待處理的最高優先順序外部中斷的中斷來源ID（在`meihap`暫存器中）及其相應優先順序（在`meicidpl`暫存器中）的擷取操作。
4. 然後，處理常式讀取`meihap`暫存器以獲取在`claimid`欄位中提供的中斷來源ID。根據`meihap`暫存器的內容，外部中斷處理常式跳轉到該外部中斷來源的特定處理常式。在圖18中可以觀察到此程序。
5. 對於特殊中斷來源的中斷處理常式（ISR）：
 - a. 對於電平觸發的中斷來源，中斷處理常式會清除SoC IP中發起中斷請求的狀態。
 - b. 對於邊緣觸發的中斷來源，中斷處理常式通過寫入`meigwclr`S暫存器將來源閘道中的IP位元清零。
 這會將來源的中斷請求置為無效。
6. 同時，PIC在背景繼續評估待處理的中斷。

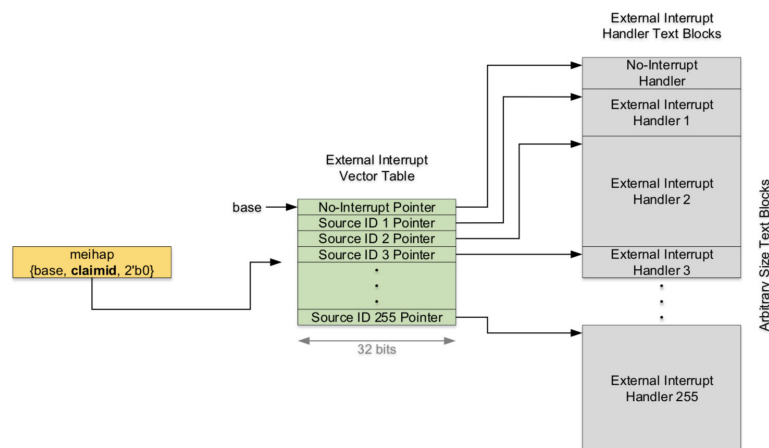


圖18. 向量化外部中斷（取自[PRM v1.7]）

必須注意的是，這是一種常規的工作模式。此外，SweRV EH1核心還支援巢狀中斷（最多15個）。有關更多資訊，請查閱[PRM v1.7]。