THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 9
Interrupt-driven I/O

1. INTRODUCTION

In this lab, we introduce the concept of interrupts and show how to use them on RVfpga. Interrupts may be generated by software or hardware. In this lab we focus on hardware interrupts which are triggered by the value of a physical pin changing. Specifically, we begin in Section 2 by describing the differences between **Programmed I/O** and **Interrupt-driven I/O**. Then, we explain the operation of RVfpga's Interrupt Controller, which is part of the SweRV EH1 core (Section 3). In Section 4 we describe how to configure external interrupts using Western Digital's Peripherals Support Package (PSP) and Board Support Package (BSP), which are software that include drivers for hardware peripherals. Finally, we introduce some example programs (Section 5) and propose some exercises (Section 6) for using and extending RVfpga's hardware interrupts.

2. PROGRAMMED I/O VS. INTERRUPT-DRIVEN I/O

Several methods exist for interacting with peripherals: Programmed I/O, Interrupt-driven I/O, and Direct Memory Access (DMA). In labs 2-8, we used **Programmed I/O** to interact with peripherals. In Programmed I/O, the user program continually polls the I/O interface and, depending on its state, reacts accordingly. For example, the *Fundamental Exercise* from Lab 6 used programmed I/O by continuously polling (reading) switches 0 and 1 to control the speed and direction of a block of four lit LEDs that repeatedly moved from one side of the LEDs to the other. Programmed I/O is very simple to implement and requires very little hardware support, but the continuous polling of the I/O interface keeps the processor busy doing useless work.

Interrupt-driven I/O overcomes this drawback and enables the program to only react when an event occurs on the peripheral. In this scheme, the peripheral is responsible for sending a signal (called an **interrupt**) to the processor when some event occurs – for example, a timer overflowing, a character being received on a UART interface, a button toggling, etc. When no event occurs (i.e., there is no interrupt), the processor continues doing useful work. When the processor receives an interrupt, it stalls the program that it was running and invokes an *interrupt service routine* (ISR), also called an *interrupt handler*. An ISR is essentially a function with `void` arguments that handles the interrupt – i.e. it reads the new value of the button, it does some action related to the timer overflow, etc. Processors usually support single- and multi-vector modes. In single-vector mode (Figure 1), all interrupts invoke the same ISR. Thus, when an interrupt occurs, the processor stalls the main program and jumps to the common ISR, which first determines the interrupt source and then executes the specific ISR code that corresponds to the identified interrupt cause. In multi-vector mode (Figure 2), each interrupt invokes a different ISR. Thus, when an interrupt is generated, the cause of the interrupt is determined first, and then the program jumps to the ISR that corresponds to the identified cause.

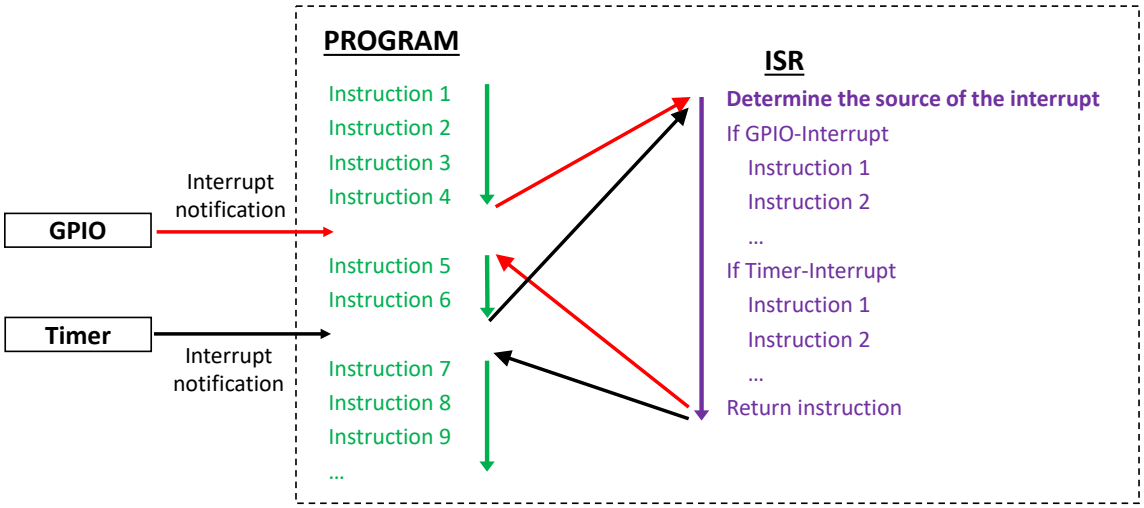

Figure 1. Example with 2 interrupts in single-vector mode

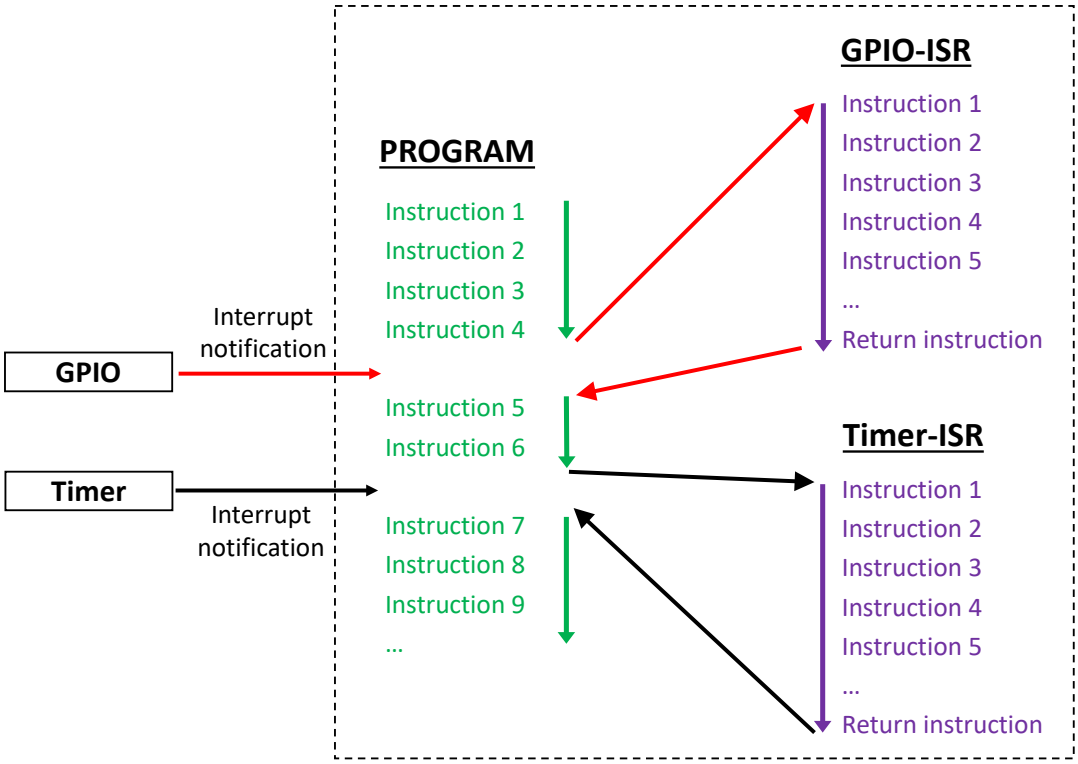

Figure 2. Example with 2 interrupts in multi-vector mode

Processors usually allow interrupts to be prioritized. Not only will higher priority interrupts be handled first, but a higher priority interrupt will pre-empt a lower-priority interrupt that was in the process of being handled. For example, suppose a button interrupt is set to priority 5, a timer interrupt is set to priority 7 and the threshold is set to 4 (so both priorities are above the threshold). If the program is executing its normal flow and the button is pressed, an interrupt will occur and the processor calls the ISR, which reads the data from the button and handles it. If a timer overflows while the button ISR is active, the ISR will itself be interrupted so that

the processor can immediately handle the timer overflow. When it is done, it will return to finish the button interrupt before returning to the main program[1].

3. THE PROGRAMMABLE INTERRUPT CONTROLLER PROVIDED BY SWERV EH1

The SweRV EH1 core supports interrupts as described in the following references and as summarized below:

- **[PRM v1.7]** Revision 1.7 (June 25, 2020), Chapter 6, "RISC-V SweRV EH1 Programmer's Reference Manual", available at https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

- **[ISM v1.11]** Version 1.11-draft (December 1, 2018), Chapter 7, "The RISC-V Instruction Set Manual – Volume II: Privileged Architecture", available at https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20181201-2650e2a

External interrupts in the SweRV EH1 core (see [PRM v1.7]) are modelled largely after the RISC-V PLIC (Platform-Level Interrupt Controller) specification (see [ISM v1.11]). However, the interrupt controller is associated with the core, not the platform. Therefore, the more general term PIC (Programmable Interrupt Controller) is used for referring to the controller available in the SweRV EH1 core. The PIC provides the following main features:

- Supports up to 255 external interrupt sources (from 1 (highest priority) to 255 (lowest priority)); each source has its own enable.
- Beyond source numbering, provides 15 additional priority levels; two priority schemes are available: 1-15 (where 1 is lowest priority), or 0-14 (where 14 is lowest priority). Each source can be assigned a priority.
- Provides support for programmable priority threshold to disable lower-priority interrupts.
- Support for vectored external interrupts, interrupt chaining, and nested interrupts.

Figure 3 illustrates a simplified version of the RVfpga's interrupt system. All functional units that generate interrupts are called **external interrupt sources**. External interrupt sources indicate an interrupt request by sending an asynchronous signal to the **PIC** with signals ending in *_irq* (an abbreviation for interrupt request). In this lab, we show how to use interrupts from the timer and the GPIO; these units generate interrupts using signals *ptc_irq* and *gpio_irq*, respectively.

Each external interrupt source connects to a dedicated gateway (located inside the PIC), a hardware structure responsible for synchronizing the interrupt request to the core's clock domain and for converting the request signal to a common interrupt request format (i.e., either active-high/low or level-triggered) for the PIC. The PIC can only handle one interrupt request per interrupt source at a time. It evaluates all pending and enabled interrupt requests and picks the highest-priority interrupt with the lowest source ID. It then compares this priority with a programmable priority threshold and, to support nested interrupts, the priority of the interrupt handler if one is currently running. If the picked request's priority is higher than both thresholds, the PIC sends an interrupt notification to the core, which stalls the

[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.

execution of the main program and jumps to the corresponding ISR, as illustrated in Figure 1 (single-vector mode) and Figure 2 (multi-vector mode).

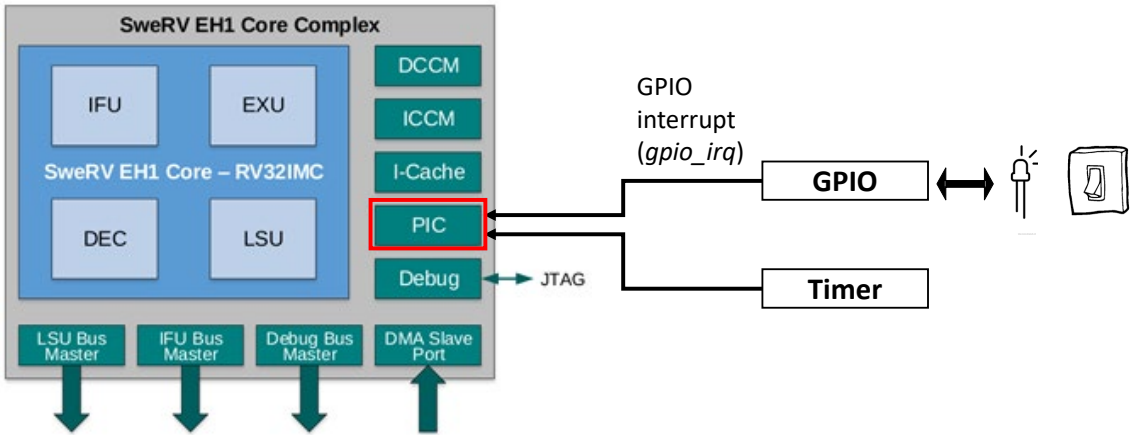

Figure 3. RVfpga interrupt system

The main functionalities of the PIC are summarized in the following basic steps:

1) Enabling/Disabling: the PIC allows enabling/disabling external interrupts
2) Configuration: the PIC can be configured to listen to external interrupts with different polarities (active-high/active-low) or type (edge-triggered/level-triggered). The PIC also permits allocating ISRs to different memory addresses.
3) Filtering and priority assignments: the PIC allows assigning priority levels to interrupts. When the main program is running, the PIC selects the enabled, triggered interrupt with the highest priority level.
4) Notification: once the PIC selects the interrupt with the highest priority, it notifies the core to stop the execution of the main program in order to jump to the routine that services the chosen interrupt.
5) Pre-emption: if nested interrupts are enabled, it is possible to pre-empt the interrupt being serviced by another one with a higher priority.

4. CONFIGURING EXTERNAL INTERRUPTS IN SweRV EH1

Similarly to any other peripheral, the PIC is configured using memory-mapped registers which are accessible to the user via load/store instructions. Using the interrupt system at a register-level would be possible but very complex; fortunately, WD's Processor Support Package (PSP) and Board Support Package (BSP) (https://github.com/westerndigitalcorporation/riscv-fw-infrastructure) include several functions that provide a much simpler approach to implement programs using interrupts. Table 1 describes the main functions and macros that are required to configure the external interrupts. For the sake of completeness, the Appendix at the end of this document provides a description of the different registers available and the steps for register-level configuration and use of the PIC.

Table 1. Basic functions and macros used to configure external interrupts

Header	Description
void **pspInterruptsSetVectorTableAddress**(void* pVectTable);	Prepares vector-table address
void **pspExternalInterruptSetVectorTableAddress**(void* pExtIntVectTable);	Prepares external interrupts vector-table address
void **bspInitializeGenerationRegister**(u32_t uiExtInterruptPolarity)	Put the Generation-Register in its initial state
void **bspClearExtInterrupt**(u32_t uiExtInterruptNumber)	Clear the trigger that generates external interrupt
void **pspExtInterruptSetPriorityOrder**(u32_t uiPriorityOrder);	Sets Priority Order (Standard or Reserved)
void **pspExtInterruptsSetThreshold**(u32_t uiThreshold);	Sets the priority threshold of the external interrupts in the PIC
void **pspExtInterruptsSetNestingPriorityThreshold**(u32_t uiNestingPriorityThreshold);	Sets the nesting priority threshold of the external interrupts in the PIC
void **pspExtInterruptSetPolarity**(u32_t uiIntNum, u32_t uiPolarity);	Sets the polarity (active-high or active-low) of a specified interrupt line
void **pspExtInterruptSetType**(u32_t uiIntNum, u32_t uiIntType);	Sets the type (Level-triggered or Edge-triggered) of a specified interrupt line
void **pspExtInterruptClearPendingInt**(u32_t uiIntNum);	Clears the indication of pending interrupt for the specified interrupt line
void **pspExtInterruptSetPriority**(u32_t uiIntNum, u32_t uiPriority);	Sets the priority of a specified interrupt line
void **pspExternalInterruptEnableNumber**(u32_t uiIntNum);	Enables a specified interrupt line in the PIC
void **pspInterruptsEnable**(void);	Enable interrupts (in all privilege levels) regardless their previous state
void **pspInterruptsDisable**(u32_t *pOutPrevIntState);	Disables interrupts and return the current interrupt state in each one of the privileged levels

Example interrupt service routines (ISRs) are given later in the lab. They follow the steps described below to configure RVfpga interrupts, based on the functions from Table 1. Note that, in addition to configuring the PIC, the peripherals generating the external interrupt must be configured as well (this will be described later for each of the peripherals used in the examples and exercises).

DEFAULT INITIALIZATION OF THE INTERRUPT SYSTEM:

1. In multi-vector mode, set the base address of the external vectored interrupt address table. Use functions `pspInterruptsSetVectorTableAddress` and `pspExternalInterruptSetVectorTableAddress`.
2. Put the Generation Register in its initial state. Use function `bspInitializeGenerationRegister`.
3. Make sure the external-interrupt triggers are cleared. Use function `bspClearExtInterrupt`.
4. Set default values for the priority order (function `pspExtInterruptSetPriorityOrder`), threshold (function `pspExtInterruptsSetThreshold`) and nesting priority threshold (function `pspExtInterruptsSetNestingPriorityThreshold`).

INITIALIZATION OF EACH INTERRUPT SOURCE:

1. For each interrupt source, set the polarity (active-high/active-low) and type (level-triggered/edge-triggered) using functions `pspExtInterruptSetPolarity` and `pspExtInterruptSetType`
2. Clear any pending interrupt using function `pspExtInterruptClearPendingInt`.
3. Set the priority level for each external interrupt source by using function `pspExtInterruptSetPriority`.
4. Enable interrupts for the appropriate external interrupt source by using function `pspExternalInterruptEnableNumber`.
5. In multi-vector mode, for each external interrupt source, write the address of the corresponding handler in the external vectored interrupt address table.

ADVANCED TASK: In order to gain a deeper understanding about these basic functions, view the PSP code located at *.platformio/packages/framework-wd-riscv-sdk/psp* and the BSP code located at *.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp*. Of special interest are the files listed below, some of them contained within the *api_inc* subfolder.

- *bsp_external_interrupts.h*: external_interrupts creation in RVfpga
- *psp_interrupts_eh1.h*: it provides information and registration APIs for ISRs on the EH1 core
- *psp_ext_interrupts_eh1.h*: it defines the psp external interrupts interfaces for SweRV EH1
- *psp_macros_eh1.h*: it defines the psp macros for SweRV EH1
- *psp_csrs_eh1.h*: definitions of SweRV EH1 CSRs

It is also recommended to analyse at least one of these functions down to the register-level. For this purpose, you can use the information provided in the Appendix, which describes how the SweRV EH1 Core's PIC configures and manages external interrupts at a register-level.

ADVANCED TASK: We also recommend that you analyse and execute the external interrupts demo provided by Western Digital at https://github.com/westerndigitalcorporation/riscv-fw-infrastructure and available as a PlatformIO project at: *[RVfpgaPath]/RVfpga/Labs/Lab9/WD_demo_external_int_Original*. If everything works correctly, you should see the following messages in the serial console:

```
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. EXAMPLES

In this section, we provide examples of converting programmed I/O programs to interrupt-driven I/O programs. We show three examples that illustrate the different problems inherent

to Programming I/O (first and second examples) and then show how these problems can be easily solved by using an Interrupt-driven I/O scheme (third example).

A. LED-Switch_C-Lang program

The *LED-Switch_C-Lang* program (see Figure 4) inverts the right-most LED state every time a 0→1 transition occurs on the right-most switch. The program is available at:
[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_C-Lang.c

After the initialization of the peripherals, the program enters an infinite loop that compares the current switch state with the previous switch state and, in case a 0→1 transition is detected, it inverts the LED state (note that, when a 1→0 transition occurs, nothing happens).

In previous examples and exercises written in C, we defined macros for accessing the I/O registers (`READ_GPIO`, `READ_Reg`, `WRITE_GPIO`, `WRITE_Reg`, etc.). In this example, we instead use two macros defined in the PSP for the same purpose: `M_PSP_READ_REGISTER_32`, that reads a 32-bit register provided as an argument, and `M_PSP_WRITE_REGISTER_32`, that writes a 32-bit register with the value provided in the second argument. Remember that, for being able to use these macros, you must include line `framework = wd-riscv-sdk` in file *platformio.ini* (this is the default when a project is created with RVfpga as the target) and line `#include "psp_api.h"` at the beginning of the program (Figure 4, line 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 }
```

Figure 4. *LED-Switch_C-Lang* program

TASK: Analyse the *LED-Switch_C-Lang* program to understand it in detail. If needed, you can use the debugger for analysing the program step-by-step.

The program works correctly, but it is very inefficient, as the processor does nothing else than reading/writing the switches/LEDs. Obviously, we want our processor to do more things than only communicating with the I/O devices.

B. LED-Switch_7SegDispl_C-Lang program

In this second example, *LED-Switch_7SegDispl_C-Lang*, the program extends *LED-Switch_C-Lang* with a second peripheral: the 7-segment displays. The program performs two tasks:

- As in the first example, it inverts the right-most LED every time a 0→1 transition on the right-most switch occurs.
- It shows an ascending count in the 8-digit 7-segment displays, that increments around once per second. Note that, for simplicity, we create the delay of one second with a `for` loop (in Exercise 1, you will use the timer from Lab 8 for this purpose).

You can see this program in Figure 5 and you can find it at:
[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_C-Lang.c

After some initializations, the program enters an infinite loop that compares the current switch state with the previous one and, in case a 0→1 transition is detected, it inverts the LED state. Then, the value shown on the 8-digit 7-segment displays is incremented and a delay is generated. See the red box in Figure 5.

```c
#include "psp_api.h"

#define SegEn_ADDR    0x80001038
#define SegDig_ADDR   0x8000103C

#define GPIO_SWs    0x80001400
#define GPIO_LEDs   0x80001404
#define GPIO_INOUT  0x80001408

int main ( void )
{
    int i, LED_state, Sw_current_state, Sw_next_state, count=0;

    /* Configure LEDs and Switches */
    M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF);

    /* Configure 7-Seg Displays */
    M_PSP_WRITE_REGISTER_32(0x80001038, 0x0);

    /* Init states */
    LED_state = 0;
    M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
    Sw_current_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;

    while (1) {
        /* Invert LED-0 when SW-0 goes high */
        Sw_next_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
        if(Sw_current_state==0 && Sw_next_state==1){
            LED_state = !LED_state;
            M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
        }
        Sw_current_state = Sw_next_state;

        /* Increase 7-Seg Displays */
        M_PSP_WRITE_REGISTER_32(SegDig_ADDR, count);
        count++;

        /* Delay */
        for(i=0;i<1000000;i++);
    }

    return(0);
}
```

Figure 5. *LED-Switch_7SegDispl_C-Lang* program

TASK: Analyse the *LED-Switch_7SegDispl_C-Lang* program in order to understand it in detail. If needed, you can use the debugger for analysing the program step-by-step.

Note that, in this case, the program does not even work correctly in some situations. For example, a 0→1→0 switch transition that occurs within the delay loop will never be detected. Moreover, we still have the same problem as in the previous example: the processor is busy all the time just reading/writing the devices or creating a delay.

How could we improve these situations? The answer is **Interrupt-driven I/O**. In the following example and in the exercises proposed in the next section, we show how to resolve all of these problems and implement programs that are more efficient and work correctly in all situations.

C. LED-Switch_7SegDispl_Interrupts_C-Lang program

In this final example (*[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_Interrupts_C-Lang.c*), we show how to use Interrupt-driven I/O to read the state of the right-most switch. Using this strategy fixes the problem of the program missing switch transitions that occur during the delay loop. Note, however, that the problem of having the processor busy in a delay loop still persists. (You will deal with this problem in Exercise 1.)

The new **`main`** function, shown in Figure 7, performs the following tasks:

- Initialize the interrupt system:
  - Default initialization of the interrupts: invoke function `DefaultInitialization` (line 119), which we show in Figure 8.
  - Set a specific threshold, by invoking function `pspExtInterruptsSetThreshold(5)` (line 120). External interrupts whose priority is not above this threshold will be ignored.
- Initialize external interrupt line IRQ4:
  - Initialize line IRQ4: invoke function `ExternalIntLine_Initialization` (line 123) for interrupt line 4, with a priority of 6 and `GPIO_ISR` as the Interrupt Service Routine. We analyse this function in Figure 9.
  - Connect IRQ4 with GPIO interrupt line (line 124). This is done by setting bit 0 of word 0x80001018 (tagged as `Select_INT` in the example). This System Controller memory-mapped register contains 2 bits (see Figure 6): bit 0, called *irq_gpio_enable*, used to connect the GPIO interrupt line with IRQ4 when it is set to 1; and bit 1, called *irq_ptc_enable*, used to connect the timer interrupt line with IRQ3 when it is set to 1. For now, it is enough that you know this high-level functionality; later, in Exercise 2, we explain the Verilog implementation in detail, so that you can modify it as part of that exercise.

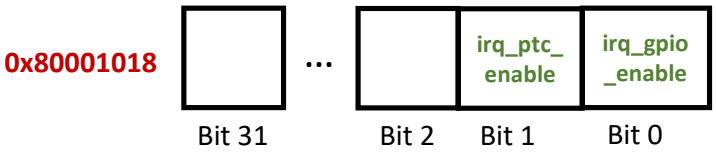

Figure 6. Register 0x80001018 of RVfpga.

- Initialize the peripherals (in this example, the GPIO and the 7-segment displays):
  - Invoke function `GPIO_Initialization` at line 127. We analyse that function in Figure 10.
  - Enable the eight 7-segment displays (line 128).
- Enable the interrupts:
  - Invoke function `pspInterruptsEnable` (line 131) and macro `M_PSP_SET_CSR` (line 132). Constants D_PSP_MIE_NUM and D_PSP_MIE_MEIE_MASK are defined by WD's PSP.
- Finally, the 7-segment displays are written, and a delay is established within a loop that repeats forever (lines 134-141).

```
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  }
```

Figure 7. *main* function.

The **`DefaultInitialization`** function, shown in Figure 8, performs the steps explained in Section 4 below item “DEFAULT INITIALIZATION OF THE INTERRUPT SYSTEM”:

- It configures the vector-table (lines 53 and 56). Note that, in this example, array `G_Ext_Interrupt_Handlers` stores the vector-table.
- It initializes the register used for triggering the IRQs (line 59).
- It clears all external interrupts (in our case IRQ3 and IRQ4) at lines 61-65. Constants D_BSP_FIRST_IRQ_NUM and D_BSP_LAST_IRQ_NUM are defined by WD’s BSP to 3 and 4, respectively.
- It establishes the default threshold and priorities (lines 68, 71 and 74). Again, the constants used by these functions are defined by WD’s PSP.

```
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  }
```

Figure 8. *DefaultInitialization* function

The **`ExternalIntLine_Initialization`** function, shown in Figure 9, performs the steps explained in Section 4 below item “INITIALIZATION OF EACH INTERRUPT SOURCE”:

- It configures the type and polarity of the IRQ4 interrupt (the constants used by these

functions are defined by WD's PSP) and it clears any potential pending interrupts at the corresponding gateway (lines 81, 84 and 87).
- It sets the priority for IRQ4 (line 90).
- It enables IRQ4 interrupts in the PIC at line 93.
- It registers the GPIO Interrupt Service Routine (GPIO_ISR) in the vector-table (at line 96), which is stored in array `G_Ext_Interrupt_Handlers`.

```
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  }
```

Figure 9. *ExternalIntLine_Initialization* function

The **`GPIO_Initialization`** function, shown in Figure 10, performs the following tasks:
- Configure the GPIO pins as input/output and initialize the LEDs to 0 (lines 103 and 104).
- Configure the GPIO interrupts. (To further understand the functionality of each GPIO register, use the GPIO Core Specification, available at: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/docs/gpio_spec.pdf*.)
    - *RGPIO_INTE*: it determines which general-purpose pins generate an interrupt (line 107).
    - *RGPIO_PTRIG*: it determines the edge that generates an interrupt (line 108).
    - *RGPIO_INTS*: it clears the interrupts of all pins (line 109).
    - *RGPIO_CTRL*: the least-significant bit of this register enables interrupt generation (line 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  }
```

Figure 10. *GPIO_Initialization* function.

Finally, the ISR (i.e., the **`GPIO_ISR`** function shown in Figure 11) is invoked when an interrupt is triggered at the GPIO. This ISR (Interrupt Service Routine) performs the following tasks:
- The current state of the LEDs is read (line 35).

- The LEDs are inverted and masked (lines 36-37).
- The LEDs are written with the new value (line 38).
- The GPIO interrupt is cleared (line 41).
- The IRQ4 external interrupt is cleared (line 44).

```
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  }
```

Figure 11. *GPIO_ISR* function.

TASK: Analyse the *LED-Switch_7SegDispl_Interrupts_C-Lang* program to understand it in detail. You can compare the implementation with the explanations of Section 4 and, if needed, use the debugger for analysing the program step-by-step.

6. EXERCISES

Exercise 1. Modify the *LED-Switch_7SegDispl_Interrupts_C-Lang* program to include a second interrupt source, in this case generated by the timer. Recall that a timer can act as a PWM generator, timer, or counter, so it is generally referred to as a PTC unit.

- In RVfpga, the timer interrupt is connected to IRQ3 by setting bit 1 (*irq_ptc_enable*) of word 0x80001018 (see Figure 6).
- Create a function that initializes PTC interrupts, similar to `GPIO_Initialization` in the previous example.
- Create a second ISR called `PTC_ISR`. It should be similar to `GPIO_ISR` in the *LED-Switch_7SegDispl_Interrupts_C-Lang* program, but it should instead be invoked using IRQ3. `PTC_ISR` should handle and clear the timer interrupt.

Once the program is implemented and debugged, use the PSP functions `pspExtInterruptsSetThreshold(threshold)` and `pspExtInterruptSetPriority(interrupt_source, priority)` to analyse different combinations of the priorities and the threshold. Note that you can even change the priorities at execution time; for example, you can show the 7-segment displays count up to 10 and then stop counting by modifying the priority of the appropriate external interrupt source.

Exercise 2. Modify RVfpga to include a third interrupt source coming from the second GPIO that you designed in Lab 6 for controlling the on-board pushbuttons (GPIO2). Two approaches are possible for completing this exercise:

- You can connect the GPIO2 interrupt to an unused external interrupt source. SweRV EH1 provides up to 255 different interrupt lines and so far we have only used 2 of them. The drawback of this approach is that WD's libraries

need to be modified.

- You can connect the GPIO2 interrupt to IRQ4, so that the GPIO module (that connects to the LEDs and switches) and GPIO2 (that connects to the pushbuttons) use a single-vector interrupt mode. Although multi-vector mode is preferable under some situations, the advantage of this approach is that you can reuse the BSP.

We provide some guidance for the second approach by providing some details about the low-level implementation of interrupts in RVfpga.

Figure 12 shows the RVfpga circuit that connects the various interrupt sources (GPIO interrupt, timer interrupt – and the interrupt sources originally available in the SweRVolf core, which we do not analyse nor use here) with *IRQ4* and *IRQ3*. Specifically, *IRQ4* is connected to the GPIO when *irq_gpio_enable* = 1 (Figure 6), whereas *IRQ3* is connected with the timer when *irq_ptc_enable* = 1 (Figure 6). When *irq_gpio_enable* = *irq_ptc_enable* = 0, *IRQ4* and *IRQ3* are connected with the SweRVolf original interrupt sources, which we do not use in this lab (if you are interested in using these interrupt sources, you can view more information from https://github.com/chipsalliance/Cores-SweRVolf).

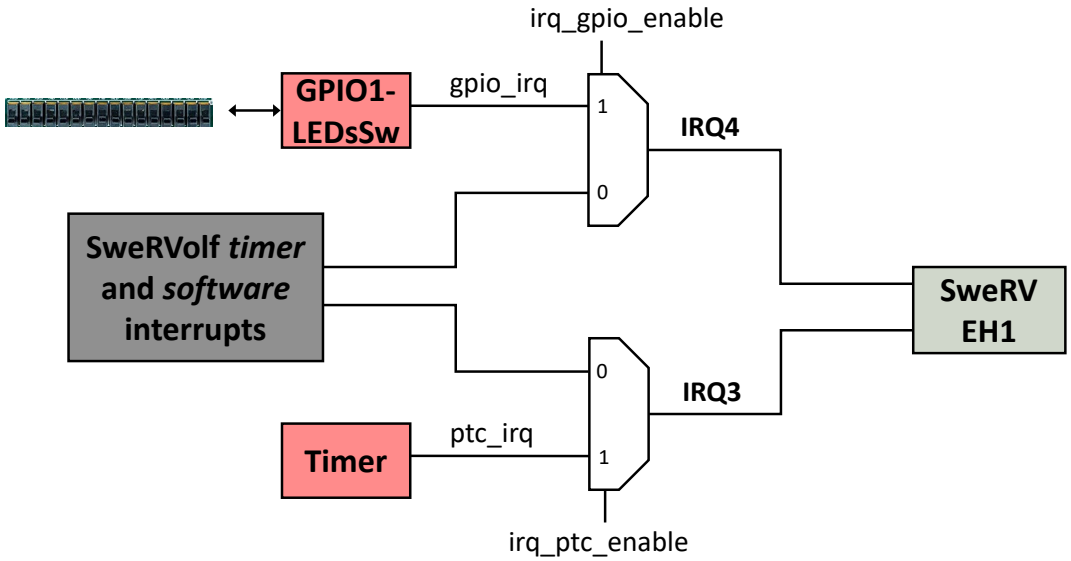

Figure 12. Logic implementation: connection of GPIO and timer interrupts with IRQ4 and IRQ3 respectively

Figure 13 shows the Verilog region of module **swervolf_core** that implements the connection between the interrupt sources and *IRQ4* and *IRQ3*. The GPIO interrupt is connected with *IRQ4* when signal *irq_gpio_enable* is 1 (top part of the red box). The timer interrupt is connected to *IRQ3* when signal *irq_ptc_enable* is 1 (bottom part of the red box). When both signals are 0 (code not highlighted in the figure), the interrupt sources implemented in SweRVolf are connected to *IRQ3* and *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
160      end
```

Figure 13. Verilog implementation: highlighted in red, connection of GPIO and timer interrupts with IRQ4 and IRQ3, respectively.

In this exercise you must extend the previous implementation (Figure 12) to include a new interrupt source connected to *IRQ4* as shown in Figure 14.

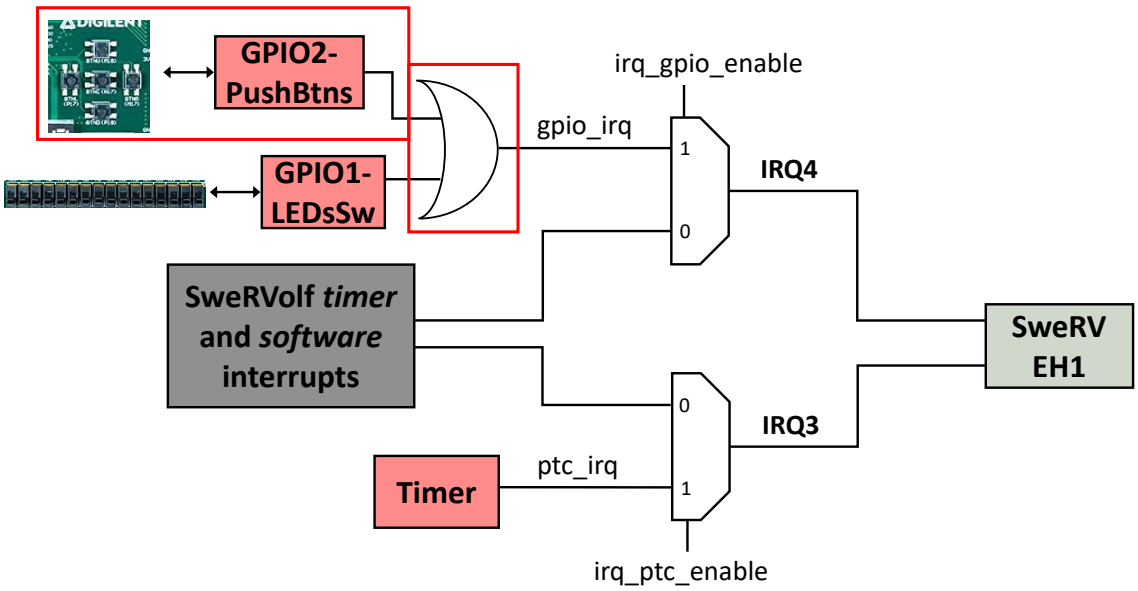

Figure 14. Logic implementation: connection of a second interrupt source (provided by the GPIO that reads the pushbuttons) with IRQ4

We highlight a few other Verilog regions that you should also understand, although you do not need to modify them in this example.

- The interrupt sources are inserted into the SweRV processor at line 599 of

the **swervolf_core** module (Figure 15). Although four interrupt sources are available, in this lab we are only interested in sources *sw_irq4*, and *sw_irq3*.

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

Figure 15. Interrupt sources sent to SweRV

- The enable signals, *irq_gpio_enable* and *irq_ptc_enable* (accessible at address 0x80001018, see Figure 6), are written by the core at lines 192-196 of the **swervolf_syscon** module (Figure 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
```

Figure 16. Writing of register 0x80001018 from the SweRV core

These enable signals, *irq_gpio_enable* and *irq_ptc_enable*, are read at lines 248-249 by the **swervolf_syscon** module from the core (see Figure 17).

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

Figure 17. Reading of register 0x80001018 into the SweRV core

Exercise 3. Use the extended RVfpga version that you designed in the previous exercise to implement a C program that displays an increasingly incrementing binary count on the LEDs, starting at 1. Create a delay with the timer, using interrupts, for waiting between displaying each incremented value so that the values are viewable by the human eye. Read BTNC and use it to change the speed of the count, and read Switch[0] and use it to restart the count whenever it is pressed.

With your extended RVfpga from Exercise 2, you now have three possible interrupt sources:

- **GPIO** (interrupts from the switches)
- **GPIO2** (interrupts from the buttons, that you designed in the previous exercise, Exercise 2)
- **PTC** (the timer)

Given that the extended RVfpga implementation from Exercise 2 has two interrupt sources that share the same line (*IRQ4*), the corresponding Interrupt Service Routine (`GPIO_ISR`) has to identify the device that generated the interrupt. You can obtain that information from the GPIO registers.

APPENDIX

This appendix describes how the SweRV EH1 Core's Programmable Interrupt Controller (PIC) manages external interrupts at a register level. The PIC uses the memory-mapped registers shown in Table 2. It must be noted that the PIC memory space starts at address 0xF00C0000; This address is referred to as *RV_PIC_BASE*. Addresses are given relative to this base address.

Table 2. PIC Memory-mapped Register Address Map

Name	Addresses (relative to RV_PIC_BASE)	Description	Location at the manual
meipl*S*	0x0004 – 0x0004+S_{max}*4-1	External interrupt priority level register	Table 6-2 of [PRM v1.7]
meip*X*	0x1000 – 0x1000+(X_{max}+1)*4-1	External interrupt pending register	Table 6-3 of [PRM v1.7]
meie*S*	0x2000 – 0x2000+S_{max}*4-1	External interrupt enable register	Table 6-4 of [PRM v1.7]
mpiccfg	0x3000 – 0x3003	External interrupt PIC configuration register	Table 6-1 of [PRM v1.7]
meigwctrl*S*	0x4004 – 0x4004+S_{max}*4-1	External interrupt gateway configuration register (for configurable gateways only)	Table 6-11 of [PRM v1.7]
meigwclrS	0x5004 – 0x5004+S_{max}*4-1	External interrupt gateway clear register (for configurable gateways only)	Table 6-12 of [PRM v1.7]

All registers are 32 bits wide and are accessible through load and store instructions, as usual for memory-mapped I/O. The access type depends on the specific bits that we want to access (this can be viewed at [PRM v1.7]).

Some of the registers have parameterized names, which end in S or X. Several instances of these registers can exist. The parameter *S* refers to the number of external interrupt sources, which in SweRV EH1 is equivalent to the number of gateways. Thus, registers ending with '*S*' have 1 to 255 register instances available. In this lab we only use 2 external interrupt sources: **IRQ3** (associated with the timer), and **IRQ4** (associated with the GPIO). The parameter *X* refers to a group of 32 gateways. This does not mean that the gateways are grouped, but grouping them reduces the size of required memory for certain 32-bit registers where 1 bit is enough for performing an action on a group of external interrupt sources. Such is the case of the external interrupt pending register, where one bit is enough to distinguish whether or not the interrupt has been serviced. In order to get more information about these registers, the rightmost column of Table 1 points to the place within [PRM v1.7] where the bit-level (specific interrupt) description is contained.

Besides the registers shown in Table 2, the PIC contains Control and Status Registers (CSRs). The standard RISC-V ISA establishes a 12-bit encoding space (*csr[11:0]*) for up to 4,096 CSRs. By convention, the upper 4 bits of the CSR address (*csr[11:8]*) are used to encode the read and write accessibility of the CSRs according to privilege level. The top two bits (*csr[11:10]*) indicate whether the register is read/write (00, 01, or 10) or read-only (11). The next two bits (*csr[9:8]*) encode the lowest privilege level that can access the CSR. More information about the CSRs is available in [PRM v1.7] and [ISM v1.11]. Table 3 lists those CSRs that are useful for managing the external interrupts in the SwerRV EH1 core. These are accessible through dedicated load and store instructions such as *csrrw* or *csrrs* (CSR read/write and CSR read/set).

Table 3. PIC Non-standard RISC-V CSR Address Map.

Name	Number	Description	Location
meivt	0xBC8	External interrupt vector table register	Table 6-6 of [PRM v1.7]
meipt	0xBC9	External interrupt priority threshold register	Table 6-5 of [PRM v1.7]
meicpct	0xBCA	External interrupt claim ID / priority level capture trigger register	Table 6-8 of [PRM v1.7]
meicidpl	0xBCB	External interrupt claim ID's priority level register	Table 6-9 of [PRM v1.7]
meicurpl	0xBCC	External interrupt current priority level register	Table 6-10 of [PRM v1.7]
meihap	0xFC8	External interrupt handler address pointer register	Table 6-7 of [PRM v1.7]
mie	0x304	Machine interrupt enable register	Table 11-1 of [PRM v1.7]
mstatus	0x300	Machine status register	Figure 3.7 of [ISM v1.11]

The right-most column on Table 3 points to the place in [PRM v1.7] or [ISM v1.11] where bit-level information is described for the given CSR (note that the *mstatus* bits description is not provided in [PRM v1.7] but in [ISM v1.11] instead).

A. External Interrupt Configuration

In this subsection we summarize the basic steps needed to configure an external interrupt using the aforementioned registers:

1. Disable all external interrupts by clearing bit *miep* within the *mie* CSR.
2. Configure the priority order by writing the *priord* bit of the *mpiccfg* register.
3. In multi-vector mode, if not configured, set the base address of the external vectored interrupt address table by writing the base field of the *meivt* register.
4. Set the priority threshold by writing the *prithresh* field of the *meipt* register.
5. Initialize the nesting priority thresholds by writing '0' (or '15' for reversed priority order) to the *clidpri* field of the *meicidpl* and the *currpri* field of the *meicurpl* registers.
6. For each configurable gateway *S*, set the polarity (active-high/active-low) and type (level-triggered/edge-triggered) in the *meigwctrlS* register and clear the IP bit by writing to the gateway's *meigwclrS* register.
7. In multi-vector mode, for each external interrupt source S, write the address of the corresponding handler in the external vectored interrupt address table.
8. Set the priority level for each external interrupt source *S* by writing the corresponding priority field of the *meiplS* registers.
9. Enable interrupts for the appropriate external interrupt sources by setting the *inten* bit of the *meieS* registers for each interrupt source *S*.
10. Activate the *mei* bit within the *mstatus* CSR.
11. Enable all external interrupts by setting bit *miep* within the *mie* CSR.

These are the general steps for *S* gateways. However, in RVfpga we only use 2 interrupt sources (IRQ3 and IRQ4), each of which has its own gateway. Furthermore, it must be noted that the order is not fully strict, as some actions are interchangeable (e.g., step 4 can be

completed prior to step 2). Besides, because each function calls pspInterruptsDisable upon entry, step 1 is not strictly needed.

B. External Interrupt Operating Mode

In this subsection we describe how the PIC operates once an external interrupt is triggered. Once the desired event occurs on the external interrupt line (wire), the following actions take place:

1. The PIC decides which pending interrupt possesses the highest priority.
2. When the target hart (hardware thread) takes the external interrupt, it disables all interrupts (i.e., it clears the *mie* bit in the RISC-V hart's *mstatus* register) and jumps to the external interrupt handler.
3. The external interrupt handler writes to the *meicpct* register to trigger the capture of the interrupt source ID of the highest priority external interrupt that is pending (in the *meihap* register) and its corresponding priority (in the *meicidpl* register).
4. The handler then reads the *meihap* register to obtain the interrupt source ID provided in the *claimid* field. Based on the contents of the *meihap* register, the external interrupt handler jumps to the handler specific to this external interrupt source. This can be observed in Figure 18.
5. The source-specific interrupt handler (ISR) services the external interrupt, and then:
    a. For level-triggered interrupt sources, the interrupt handler clears the state in the SoC IP which initiated the interrupt request.
    b. For edge-triggered interrupt sources, the interrupt handler clears the IP bit in the source's gateway by writing to the *meigwclrS* register.

    This deasserts the source's interrupt request.
6. Meanwhile, in the background, the PIC continues evaluating pending interrupts.

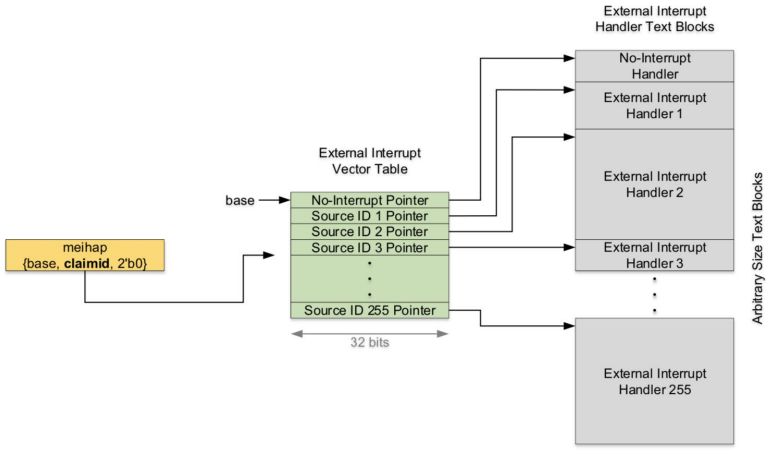

Figure 18. Vectored External Interrupts (taken from [PRM v1.7])

It must be noted that this is regular operation mode. Nested interrupts (a maximum of 15) are also supported in the SweRV EH1 core. For more information, please consult the [PRM v1.7].