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 RVfpgaEL2. 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 the RVfpgaEL2 System's Interrupt Controller, which is part of the VeeR EL2 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 the RVfpgaEL2 System'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 1-8, 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 the drawback of the processor being continuously busy polling the I/O 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.
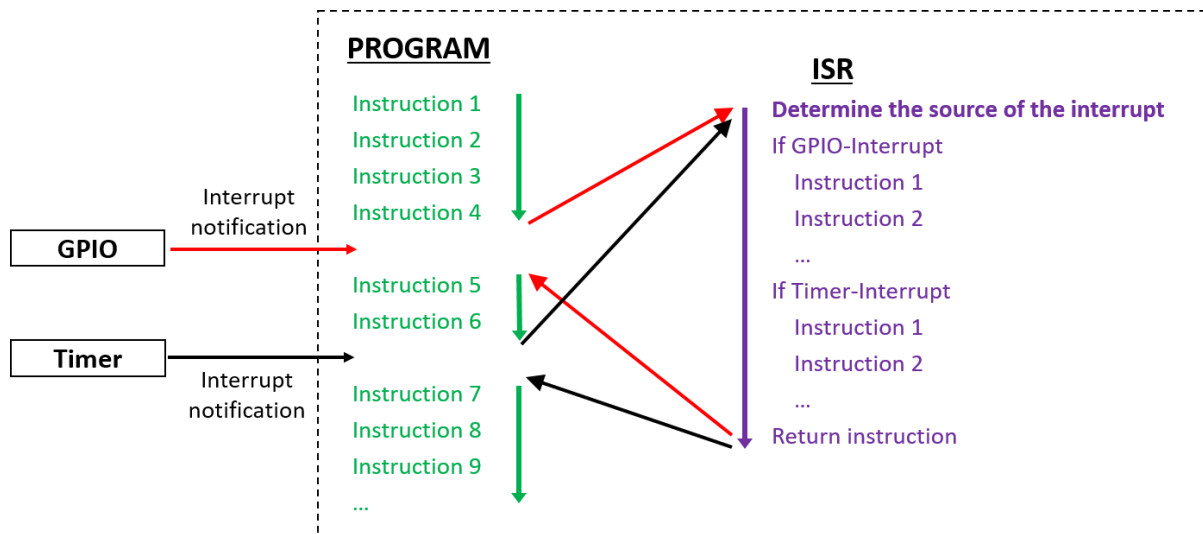
PROGRAM

Instruction 1
Instruction 2
Instruction 3
Instruction 4

Instruction 5
Instruction 6

Instruction 7
Instruction 8
Instruction 9
...

GPIO

Timer

Interrupt notification

Interrupt notification

ISR

Determine the source of the interrupt

If GPIO-Interrupt
    Instruction 1
    Instruction 2
    ...

If Timer-Interrupt
    Instruction 1
    Instruction 2
    ...

Return instruction

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

PROGRAM

Instruction 1
Instruction 2
Instruction 3
Instruction 4

Instruction 5
Instruction 6

Instruction 7
Instruction 8
Instruction 9
...

GPIO

Timer

Interrupt notification

Interrupt notification

GPIO-ISR

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
...
Return instruction

Timer-ISR

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
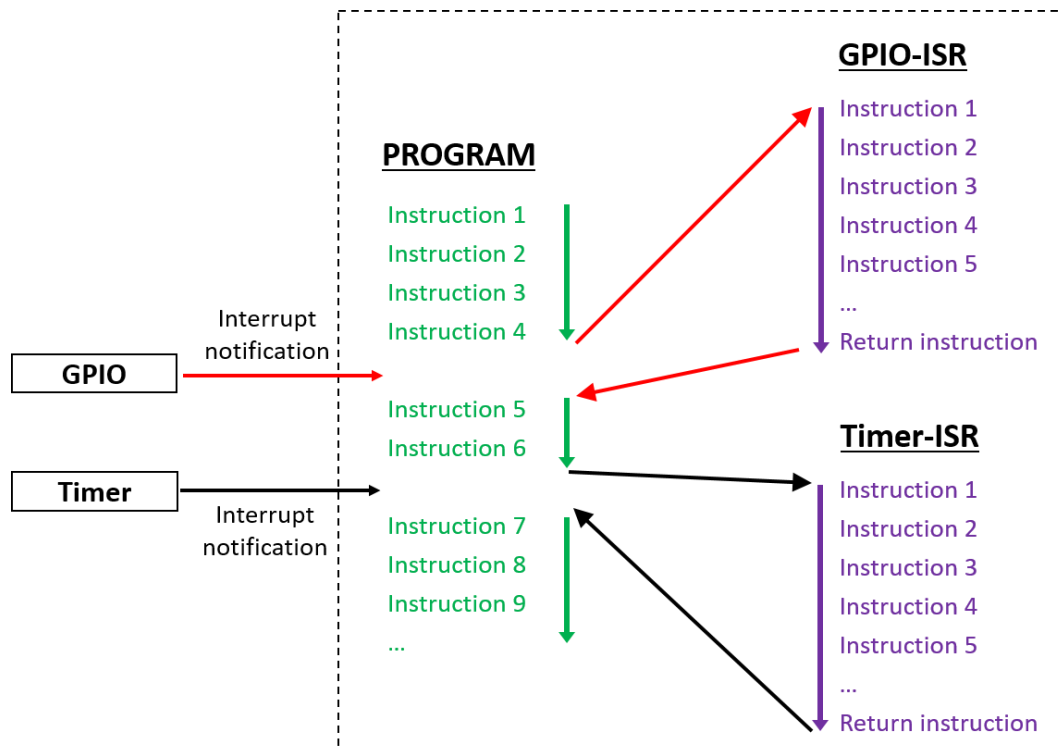...
Return instruction

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 preempt a lower-priority interrupt that was in the process of being handled. A higher priority is indicated by a higher number. For example, suppose a pushbutton 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 pushbutton 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 VeeR EL2

The VeeR EL2 core supports interrupts as described in the following references and as summarized below:

- **[PRM]** "RISC-V VeeR EL2 Programmer's Reference Manual", available at https://github.com/chipsalliance/Cores-VeeR-EL2/blob/main/docs/RISC-V_VeeR_EL2_PRM.pdf

- **[ISM]** "The RISC-V Instruction Set Manual – Volume II: Privileged Architecture", available at https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

External interrupts in the VeeR EL2 core (see [PRM]) are modelled largely after the RISC-V PLIC (Platform-Level Interrupt Controller) specification (see [ISM]). 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 VeeR EL2 core. The PIC provides the following main features:

- Up to 255 global (core-external) interrupt sources (from 1 (highest) to 255 (lowest)) with separate enable control for each source
- 15 priority levels (numbered 1 (lowest) to 15 (highest)), separately programmable for each interrupt source
- Programmable reverse priority order (14 (lowest) to 0 (highest))
- Programmable priority threshold to disable lower-priority interrupts
- Wake-up priority threshold (hardwired to highest priority level) to wake up core from power-saving (Sleep) mode if interrupts are enabled
- One interrupt target (RISC-V hart M-mode context)
- Support for vectored external interrupts
- Support for fast interrupt redirection in hardware (selectable by build argument)
- Support for interrupt chaining and nested interrupts
- Power reduction feature for disabled external interrupts.

Figure 3 illustrates a simplified version of the the RVfpgaEL2 System'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 for the PIC. The gateway must provide programmability for interrupt type (i.e., edge- vs. level-

---

[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.

triggered) as well as interrupt signal polarity (i.e., low-to-high vs. high-to-low transition for edge-triggered interrupts, active-high vs. -low for level-triggered interrupts).

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 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).
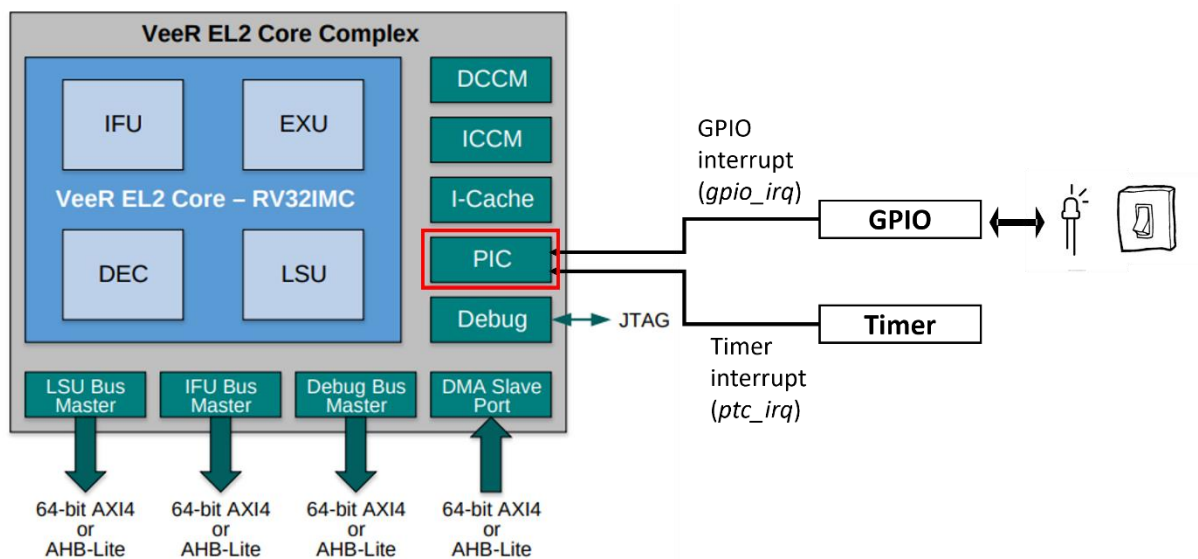
Figure 3. RVfpgaEL2's interrupt system

The main functionalities of the PIC are summarized in the following basic steps:
1) <u>Enabling/Disabling</u>: the PIC allows enabling/disabling external interrupts
2) <u>Configuration</u>: 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) <u>Filtering and priority assignments</u>: 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) <u>Notification</u>: 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) <u>Preemption</u>: if nested interrupts are enabled, it is possible to preempt the interrupt being serviced by another one with a higher priority.

4. Configuring External Interrupts in VeeR EL2

Similar 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, the Processor Support Package (PSP) and the Board Support Package (BSP) (https://github.com/chipsalliance/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.

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

Function	Description
void **pspMachineInterruptsSetVecTableAddress** (void* pVectTable)	Prepares vector-table address
void **pspMachineExtInterruptSetVectorTableAddress** (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 **pspMachineExtInterruptSetPriorityOrder** (u32_t uiPriorityOrder)	Sets Priority Order (Standard or Reserved)
void **pspMachineExtInterruptsSetThreshold** (u32_t uiThreshold)	Sets the priority threshold of the external interrupts in the PIC
void **pspMachineExtInterruptsSetNestingPriorityThreshold** (u32_t uiNestingPriorityThreshold)	Sets the nesting priority threshold of the external interrupts in the PIC
void **pspMachineExtInterruptSetPolarity** (u32_t uiIntNum, u32_t uiPolarity)	Sets the polarity (active-high or active-low) of a specified interrupt line
void **pspMachineExtInterruptSetType** (u32_t uiIntNum, u32_t uiIntType)	Sets the type (Level-triggered or Edge-triggered) of a specified interrupt line
void **pspMachineExtInterruptClearPendingInt** (u32_t uiIntNum)	Clears the indication of pending interrupt for the specified interrupt line
void **pspMachineExtInterruptSetPriority** (u32_t uiIntNum, u32_t uiPriority)	Sets the priority of a specified interrupt line
void **pspMachineExtInterruptEnableNumber** (u32_t uiIntNum)	Enables a specified interrupt line in the PIC
void **pspMachineInterruptsEnable** (void)	Enable interrupts (in all privilege levels) regardless their previous state
void **pspMachineInterruptsDisable** (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 the RVfpgaEL2 System 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 `pspMachineInterruptsSetVecTableAddress` and `pspMachineExtInterruptSetVectorTableAddress`.
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 `pspMachineExtInterruptSetPriorityOrder`), threshold (function `pspMachineExtInterruptsSetThreshold`) and nesting priority threshold (function `pspMachineExtInterruptsSetNestingPriorityThreshold`).

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 `pspMachineExtInterruptSetPolarity` and `pspMachineExtInterruptSetType`
2. Clear any pending interrupt using function `pspMachineExtInterruptClearPendingInt`.
3. Set the priority level for each external interrupt source by using function `pspMachineExtInterruptSetPriority`.
4. Enable interrupts for the appropriate external interrupt source by using function `pspMachineExtInterruptEnableNumber`.
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 and the BSP code. These libraries are included in the examples from this lab.

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 RVfpgaEL2
- *psp_interrupts_el2.h*: it provides information and registration APIs for ISRs on the EL2 core
- *psp_ext_interrupts_eh1.h*: it defines the psp external interrupts interfaces
- *psp_macros_eh1.h*: it defines the psp macros
- *psp_csrs_el2.h*: definitions of VeeR EL2 CSRs

It is also recommended to analyse at least one of these functions down to the register-level.

ADVANCED TASK: We also recommend that you analyse and execute the external interrupts demo provided at https://github.com/chipsalliance/riscv-fw-infrastructure and available as a PlatformIO project at:
[RVfpgaEL2NexysA7DDRPath]/Labs/Lab9/WD_demo_external_int_Original. If everything works correctly, you should see the following messages in the serial console:

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:

    [RVfpgaEL2NexysA7DDRPath]/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 #include "psp_api.h" at the beginning of the program (see Figure 4).

```
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:
   [RVfpgaEL2NexysA7DDRPath]/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.

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 (*[RVfpgaEL2NexysA7DDRPath]/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:
  - o Default initialization of the interrupts: invoke function `DefaultInitialization`, which we show in Figure 8.
  - o Set a specific threshold, by invoking function

`pspMachineExtInterruptsSetThreshold(5)`. External interrupts whose priority is not above this threshold will be ignored.

- Initialize external interrupt line IRQ4:
  - o Initialize line IRQ4: invoke function `ExternalIntLine_Initialization` for interrupt line 4, with a priority of 6 and `GPIO_ISR` as the interrupt service routine (ISR). We analyse this function in Figure 9.
  - o Connect IRQ4 with GPIO interrupt line. 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.
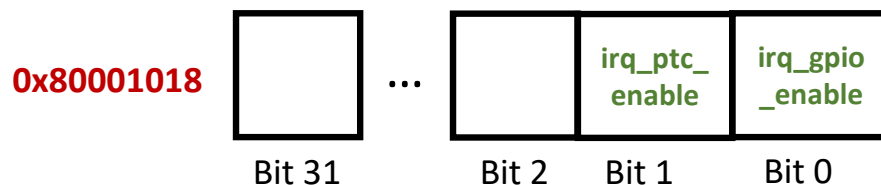
0x80001018

	...		irq_ptc_ enable	irq_gpio _enable
Bit 31		Bit 2	Bit 1	Bit 0

Figure 6. Register 0x80001018 of the RVfpgaEL2 System.

- Initialize the peripherals (in this example, the GPIO and the 7-segment displays):
  - o Invoke function `GPIO_Initialization`. We analyse that function in Figure 10.
  - o Enable the eight 7-segment displays.
- Enable the interrupts:
  - o Invoke function `pspMachineInterruptsEnable` and macro `M_PSP_SET_CSR`. Constants D_PSP_MIE_NUM and D_PSP_MIE_MEIE_MASK are defined by the PSP.
- Finally, the 7-segment displays are written, and a delay is established within a loop that repeats forever.

```c
int main(void)
{
  int count=0, i;

  /* INITIALIZE THE INTERRUPT SYSTEM */
  DefaultInitialization();                              /* Default initialization */
  pspMachineExtInterruptsSetThreshold(uiThreshold: 5);              /* Set interrupts threshold to 5 */

  /* INITIALIZE INTERRUPT LINE IRQ4 */
  ExternalIntLine_Initialization(uiSourceId: 4, priority: 6, pTestIsr: GPIO_ISR);     /* Initialize line IRQ4 with a
  M_PSP_WRITE_REGISTER_32(Select_INT, 0x1);              /* Connect the GPIO interrupt to the IRQ4 interrupt line */

  /* Check value of Select INT */
  i = M_PSP_READ_REGISTER_32(Select_INT);

  /* INITIALIZE THE PERIPHERALS */
  GPIO_Initialization();                                /* Initialize the GPIO */
  M_PSP_WRITE_REGISTER_32(SegEn_ADDR, 0x0);             /* Initialize the 7-Seg Displays */

  /* Check value of Select INT */
  M_PSP_WRITE_REGISTER_32(SegDig_ADDR, i);

  /* ENABLE INTERRUPTS */
  pspMachineInterruptsEnable();                         /* Enable all interrupts in mstatus CSR */
  M_PSP_SET_CSR(D_PSP_MIE_NUM, D_PSP_MIE_MEIE_MASK);    /* Enable external interrupts in mie CSR */

  while (1) {
    /* Increase 7-Seg Displays */
    M_PSP_WRITE_REGISTER_32(SegDig_ADDR, count);
    count++;

    /* Delay */
    for(i=0;i<5000000;i++);
  }
}
```

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.
- It initializes the register used for triggering the IRQs.
- It clears all external interrupts (in our case IRQ3 and IRQ4). Constants `D_BSP_FIRST_IRQ_NUM` and `D_BSP_LAST_IRQ_NUM` are defined the BSP to 3 and 4, respectively.
- It establishes the default threshold and priorities. Again, the constants used by these functions are defined by the PSP.

```
void DefaultInitialization(void)
{
  u32_t uiSourceId;

  /* Register interrupt vector */
  pspMachineInterruptsSetVecTableAddress(pVectTable: &M_PSP_VECT_TABLE);

  /* Put the Generation-Register in its initial state (no external interrupts are generated) */
  bspInitializeGenerationRegister(uiExtInterruptPolarity: D_PSP_EXT_INT_ACTIVE_HIGH);

  for (uiSourceId = D_BSP_FIRST_IRQ_NUM; uiSourceId <= D_BSP_LAST_IRQ_NUM; uiSourceId++)
  {
    /* Make sure the external-interrupt triggers are cleared */
    bspClearExtInterrupt(uiExtInterruptNumber: uiSourceId);
  }

  /* Set Standard priority order */
  pspMachineExtInterruptSetPriorityOrder(uiPriorityOrder: D_PSP_EXT_INT_STANDARD_PRIORITY);

  /* Set interrupts threshold to minimal (== all interrupts should be served) */
  pspMachineExtInterruptsSetThreshold(uiThreshold: M_PSP_MACHINE_EXT_INT_GET_THRESHOLD_TO_UNMASK_ALL);

  /* Set the nesting priority threshold to minimal (== all interrupts should be served) */
  pspMachineExtInterruptsSetNestingPriorityThreshold(uiNestingPriorityThreshold: M_PSP_MACHINE_EXT_INT_GET_THRESHOLD_TO_UNMASK_ALL);
}
```

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 the PSP) and it clears any potential pending interrupts at the corresponding gateway.
- It sets the priority for IRQ4.
- It enables IRQ4 interrupts in the PIC.
- It registers the GPIO interrupt service routine (GPIO_ISR) in the vector-table.

```
void ExternalIntLine_Initialization(u32_t uiSourceId, u32_t priority, fptrPspInterruptHandler_t pTestIsr)
{
  /* Set Gateway Interrupt type (Level) */
  pspMachineExtInterruptSetType(uiIntNum: uiSourceId, uiIntType: D_PSP_EXT_INT_LEVEL_TRIG_TYPE);

  /* Set gateway Polarity (Active high) */
  pspMachineExtInterruptSetPolarity(uiIntNum: uiSourceId, uiPolarity: D_PSP_EXT_INT_ACTIVE_HIGH);

  /* Clear the gateway */
  pspMachineExtInterruptClearPendingInt(uiIntNum: uiSourceId);

  /* Set IRQ4 priority */
  pspMachineExtInterruptSetPriority(uiIntNum: uiSourceId, uiPriority: priority);

  /* Enable IRQ4 interrupts in the PIC */
  pspMachineExtInterruptEnableNumber(uiIntNum: uiSourceId);

  /* Register ISRs to all interrupt sources (here we use same ISR to all of them) */
  pspMachineExtInterruptRegisterISR(uiVectorNumber: uiSourceId, pIsr: pTestIsr, pParameter: 0);
}
```

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:

[RVfpgaEL2NexysA7DDRPath]/src/VeeRwolf/Peripherals/gpio/docs/gpio_spec.pdf)
- o `RGPIO_INTE`: it determines which general-purpose pins generate an interrupt.
- o `RGPIO_PTRIG`: it determines the edge that generates an interrupt.
- o `RGPIO_INTS`: it clears the interrupts of all pins.
- o `RGPIO_CTRL`: the least-significant bit of this register enables interrupt generation.

```
void GPIO_Initialization(void)
{
  /* Configure LEDs and Switches */
  M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF);        /* GPIO_INOUT */
  M_PSP_WRITE_REGISTER_32(GPIO_LEDs, 0x0);            /* GPIO_LEDs */

  /* Configure GPIO interrupts */
  M_PSP_WRITE_REGISTER_32(RGPIO_INTE, 0x10000);       /* RGPIO_INTE */
  M_PSP_WRITE_REGISTER_32(RGPIO_PTRIG, 0x10000);      /* RGPIO_PTRIG */
  M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0);           /* RGPIO_INTS */
  M_PSP_WRITE_REGISTER_32(RGPIO_CTRL, 0x1);           /* RGPIO_CTRL */
}
```

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.
- The LEDs are inverted and masked.
- The LEDs are written with the new value.
- The GPIO interrupt is cleared.
- The IRQ4 external interrupt is cleared.

```
void GPIO_ISR(void)
{
  unsigned int i;

  /* Write the LED */
  i = M_PSP_READ_REGISTER_32(GPIO_LEDs);              /* RGPIO_OUT */
  i = !i;                                             /* Invert the LEDs */
  i = i & 0x1;                                        /* Keep only the right-most LED */
  M_PSP_WRITE_REGISTER_32(GPIO_LEDs, i);              /* RGPIO_OUT */

  /* Clear GPIO interrupt */
  M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0);           /* RGPIO_INTS */

  /* Stop the generation of the specific external interrupt */
  bspClearExtInterrupt(uiExtInterruptNumber: 4);
}
```

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 the RVfpgaEL2 System, 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 `pspMachineExtInterruptsSetThreshold(threshold)` and `pspMachineExtInterruptSetPriority(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.

Recall that you can run the same program both in RVfpgaEL2NexysA7 and RVfpgaEL2-ViDBo.

Exercise 2. Modify RVfpgaEL2-NexysA7 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. VeeR EL2 provides up to 255 different interrupt lines and so far we have only used 2 of them. The drawback of this approach is that the 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 the RVfpgaEL2 System.

Figure 12 shows the circuit that connects the various interrupt sources (GPIO interrupt, timer interrupt – and the interrupt sources originally available in the VeeRwolf 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 VeeRwolf 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-VeeRwolf).
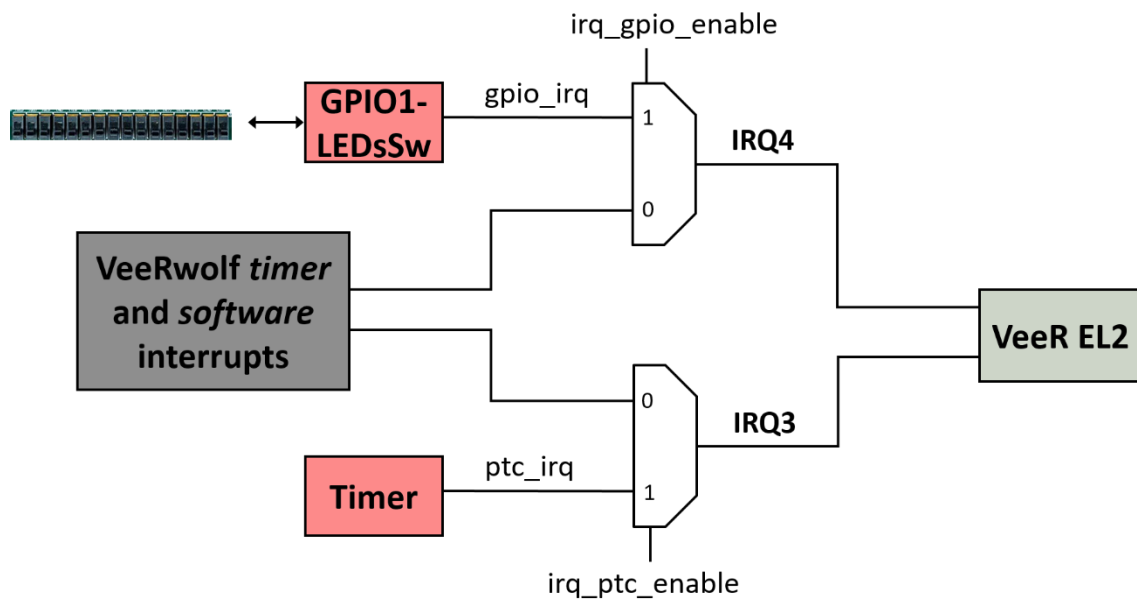
Figure 12. GPIO and timer interrupts: connections to IRQ4 and IRQ3

Figure 13 shows the Verilog region of module `veerwolf_syscon` 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 VeeRwolfX are connected to `IRQ3` and `IRQ4`.

```
// GPIO Interrupt through IRQ4. Enable by setting
if (irq_gpio_enable & gpio_irq) begin
    sw_irq4 <= 1'b1;
end

// Timer (PTC) Interrupt through IRQ3. Enable by
if (irq_ptc_enable & ptc_irq) begin
    sw_irq3 <= 1'b1;
end

// SweRVolf simple timer and software interrupts.
if (!irq_gpio_enable & !irq_ptc_enable) begin

    if (sw_irq3_edge)
        sw_irq3 <= 1'b0;
    if (sw_irq4_edge)
        sw_irq4 <= 1'b0;

    if (irq_timer_en)
        irq_timer_cnt <= irq_timer_cnt - 1;

    if (irq_timer_cnt == 32'd1) begin
        irq_timer_en <= 1'b0;
        if (sw_irq3_timer)
            sw_irq3 <= 1'b1;
        if (sw_irq4_timer)
            sw_irq4 <= 1'b1;
        if (!(sw_irq3_timer | sw_irq4_timer))
            nmi_int <= 1'b1;
    end

end

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.
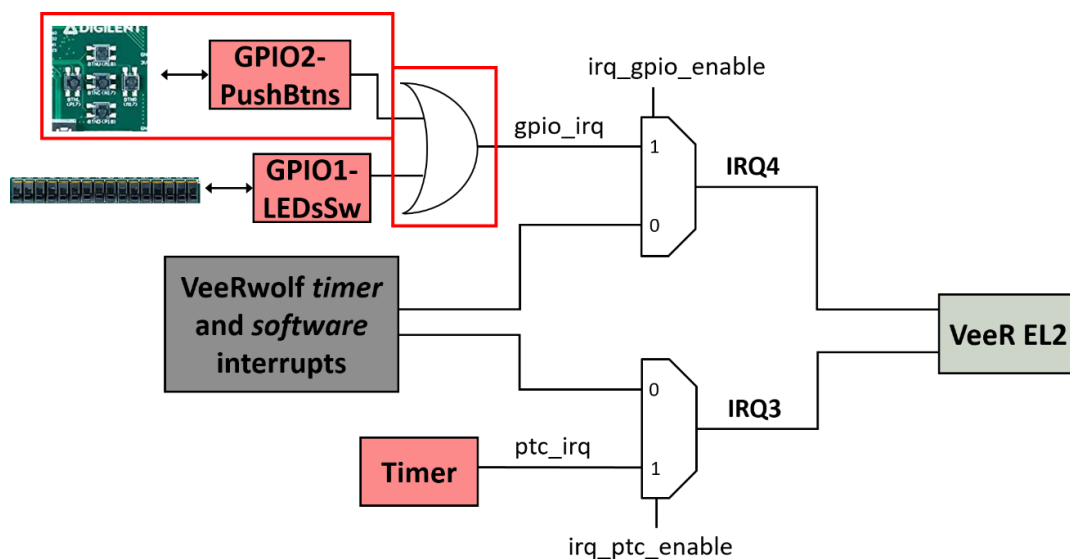

Figure 14. Logic implementation: connection of a second interrupt source (provided by the GPIO that reads the pushbuttons) to IRQ4

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 VeeR processor at line 600 of the **veerwolf_core** module (Figure 15). Although four interrupt sources are available, in this lab we are only interested in sources `sw_irq4`, and `sw_irq3`.

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

Figure 15. Interrupt sources sent to VeeR

- The enable signals, `irq_gpio_enable` and `irq_ptc_enable` (accessible at address 0x80001018, see Figure 6), are written by the core at the `veerwolf_syscon` module (Figure 16).

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

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

These enable signals, `irq_gpio_enable` and `irq_ptc_enable`, are read at the `veerwolf_syscon` module from the core (see Figure 17).

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

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

Exercise 3. Use the extended RVfpgaEL2-NexysA7 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.

With your extended RVfpgaEL2-NexysA7 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 RVfpgaEL2-NexysA7 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.

Recall that you can run the program in either RVfpgaEL2-NexysA7 or RVfpgaEL2-ViDBo.