Lab Assignment 3. Basic Management of Interrupts, Exceptions and Traps.

1. Purpose

In this practice, the student is expected to understand the hardware mechanisms that support interruptions, exceptions and traps, so that they are able to define and use basic routines for their management. The use of these routines will allow the student to understand how other higher level services of embedded software systems can be defined (device drivers or system calls), and how to deal with the exceptional situations that can occur during their execution.

2. Introduction

The event handling mechanisms (interrupts, exceptions and traps) provided by the processors facilitate access to system resources in a protected manner, ensuring their integrity against misuse.

On the one hand, by means of TRAP instructions (also called software interruptions) the user can request services that were configured during the system initialization stage. These services allow managing abstractions such as file systems, processes or access to communication ports.

The interrupt mechanism, on the other hand, allows external devices to request the processor's attention, so that a routine is executed in response to its request. Thanks to this mechanism, it is possible to avoid the polling of the job status assigned to the devices.

Finally, the exceptions allow you to define what to do when the processor is in an unstable state, such as when it executes an instruction whose code is not valid, or a division by zero or an overflow occurs in the arithmetic operating instructions.

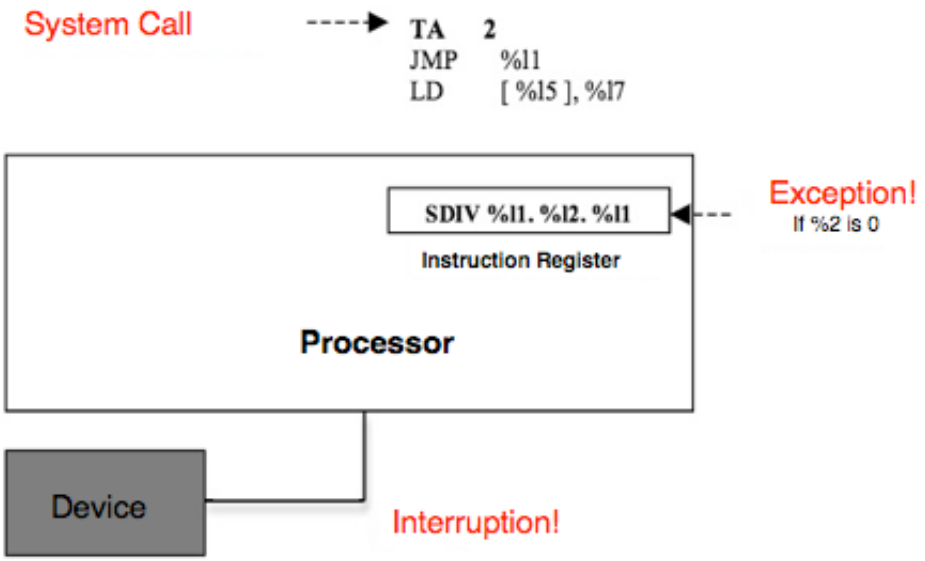

Figure 1. Event handling mechanisms provided by the processor.

3. Interrupt Vector Table

The interrupt vector table is a memory area where the processor locates the code to be executed when an interrupt, an exception or a *TRAP* type instruction is executed. The structure of this table depends on each processor, and may be very different in size depending on whether it is a general-purpose processor or a small 8-bit CPU integrated into a microcontroller. Figure 2 shows a generic interrupt vector table structure where each event is addressed by the processor by executing the handler whose address is stored in the vector itself.

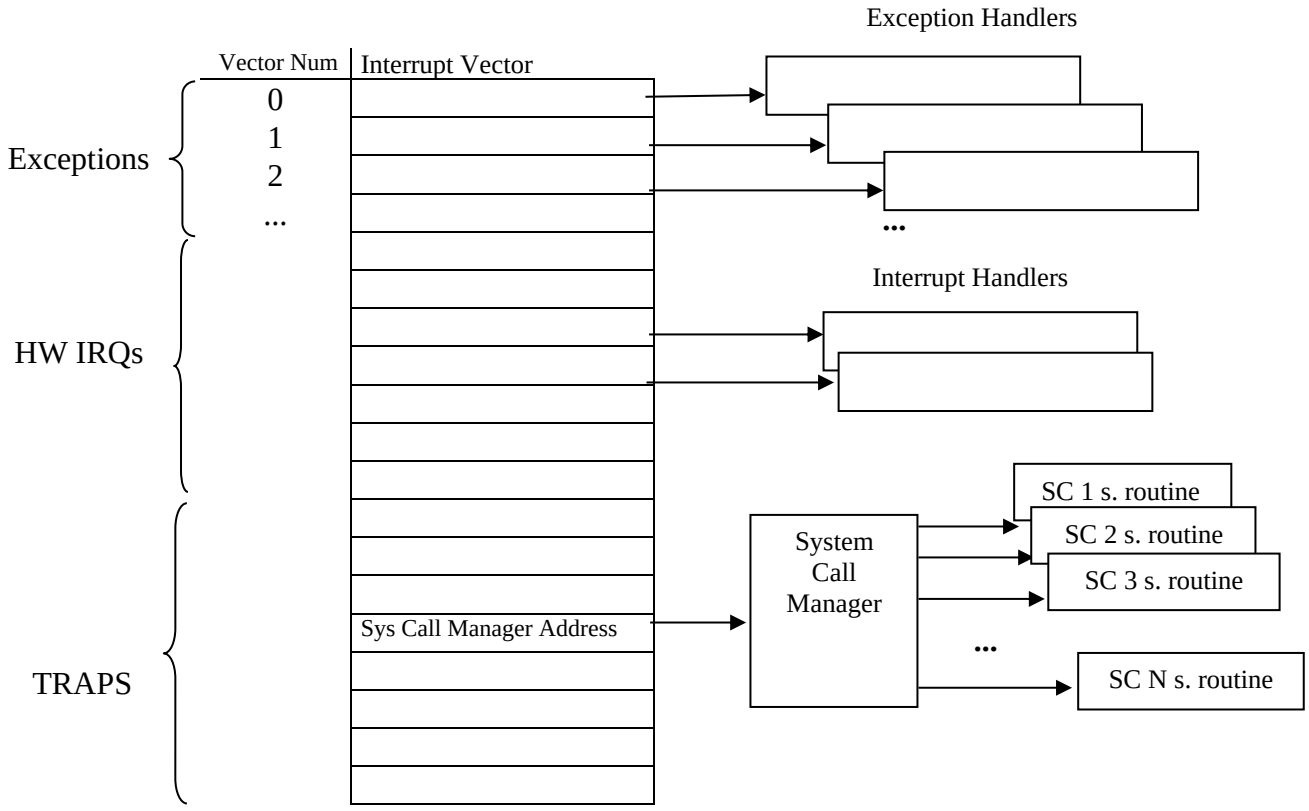

Figure 2. Generic Interrupt Vector Table.

This table must be configured during system initialization (in the case of an Operating System, it will be the one in charge of setting the initial configuration). In this way, when the applications are executed later, the system will be stable and will have a controlled response to any event that may occur.

A common configuration used by operating systems is to use a single table vector to manage all system calls. A routine, called the *system call manage*r, centralizes all requests, and based on a received parameter (either through a default register or through the stack) will identify the requested system call, and invoke its service routine.

For hardware interrupts, a common configuration is also one in which the system software provides a standard interrupt handler, with a prologue and an epilogue (written in assembler) given the specific characteristics of the architecture. This, in turn, is responsible for invoking the user interrupt handler associated with the generated hardware

interrupt. It is this handler that can be installed by the user, and a simple array can be used to store it.

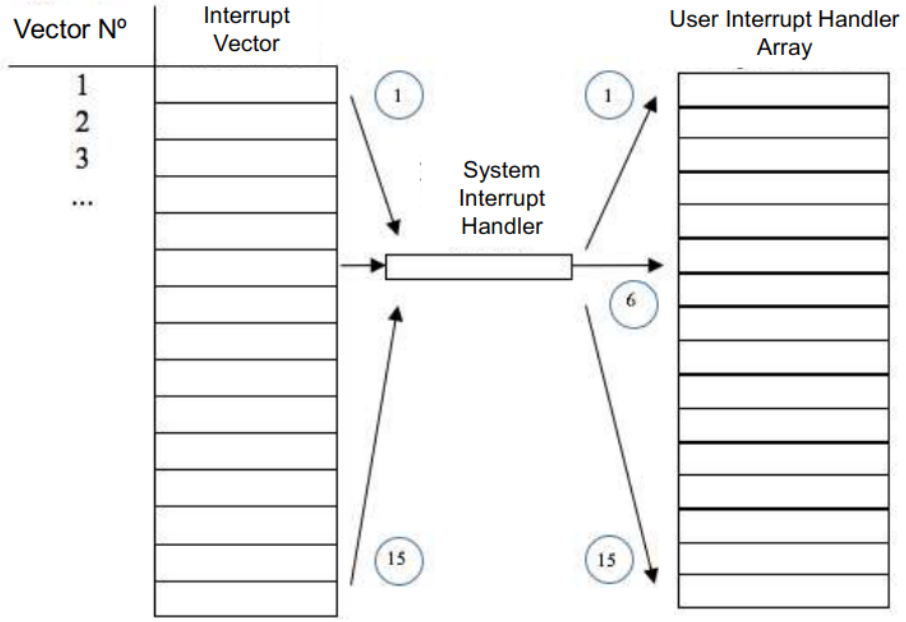

Figure 3. Configuration using a System Interrupt Handler (common for all interrupts) and a User Interrupt Handler.

4. Interrupt Vector Table of LEON3 processor

The LEON3 processor has an interrupt vector table structure in which each table element is not a direction to jump to, but a set of 4 instructions that the processor executes directly each time the event occurs.

vector Nº	Interrupt Vector			
X	Instruction-1	Instruction-2	Instruction-3	Instruction-4

This structure allows you to define a rapid response to the event. However, when the response is complex, it is necessary to jump to a routine in order to define the response. The following configuration of a vector, defines four instructions that allow to jump to a function called `handler`, saving in the register `%l0` the status register `%psr`, and in the register `%l3` the vector number. Both registers can be used by the `handler` para to complete the event handling.

Example of Interrupt Vector			
`rd %psr,%l0`	`sethi %hi(handler), %l4`	`jmp %l4 + %lo(handler)`	`mov vector,%l3`

The following figure shows the LEON3 processor interruption vector table, where **TT** indicates the vector number, **Trap** is the event to which the vector is associated (it can be an exception, a hardware interruption or a trap caused by an instruction) and **Pri** the

priority with which the event is attended. The first 15 inputs correspond to processor exceptions, the next 15 are external interrupts, while the range defined by 0x80-0xFF is reserved to be invoked through the *TRAPS* instruction

Trap	TT	Pri	Description
reset	0x00	1	Power-on reset
write error	0x2b	2	write buffer error during data store
instruction_access_error	0x01	3	Error during instruction fetch
illegal_instruction	0x02	5	UNIMP or other un-implemented instruction
privileged_instruction	0x03	4	Execution of privileged instruction in user mode
fp_disabled	0x04	6	FP instruction while FPU disabled
cp_disabled	0x24	6	CP instruction while Co-processor disabled
watchpoint_detected	0x0B	7	Hardware breakpoint match
window_overflow	0x05	8	SAVE into invalid window
window_underflow	0x06	8	RESTORE into invalid window
register_hadrware_error	0x20	9	register file EDAC error (LEON-FT only)
mem_address_not_aligned	0x07	10	Memory access to un-aligned address
fp_exception	0x08	11	FPU exception
cp_exception	0x28	11	Co-processor exception
data_access_exception	0x09	13	Access error during data load, MMU page fault
tag_overflow	0x0A	14	Tagged arithmetic overflow
divide_exception	0x2A	15	Divide by zero
interrupt_level_1	0x11	31	Asynchronous interrupt 1
interrupt_level_2	0x12	30	Asynchronous interrupt 2
interrupt_level_3	0x13	29	Asynchronous interrupt 3
interrupt_level_4	0x14	28	Asynchronous interrupt 4
interrupt_level_5	0x15	27	Asynchronous interrupt 5
interrupt_level_6	0x16	26	Asynchronous interrupt 6
interrupt_level_7	0x17	25	Asynchronous interrupt 7
interrupt_level_8	0x18	24	Asynchronous interrupt 8
interrupt_level_9	0x19	23	Asynchronous interrupt 9
interrupt_level_10	0x1A	22	Asynchronous interrupt 10
interrupt_level_11	0x1B	21	Asynchronous interrupt 11
interrupt_level_12	0x1C	20	Asynchronous interrupt 12
interrupt_level_13	0x1D	19	Asynchronous interrupt 13
interrupt_level_14	0x1E	18	Asynchronous interrupt 14
interrupt_level_15	0x1F	17	Asynchronous interrupt 15
trap_instruction	0x80 - 0xFF	16	Software trap instruction (TA)

Figure 4 LEON3 Interrupt Vector Table

In order to better manage this interruption vector table, it is very useful to define functions that, from the vector number, and from a pointer to the routine, manage the way this table must be completed. A function prototype with these characteristics would be the following:

```
uint8_t leon3_set_event_handler( uint32_t vector_num ,
                                 void (* handler) (void))
```

The parameter `vector_num c`orresponds to the vector number, while `handler` is the pointer to the function we want to be invoked when the event is attended.

5. To-do. LEON3 Project creation

Creation of a new project called prac3 whose executable is for the Sparc Bare C platform. In this project, create two subdirectories: **include** and **src**. In the **src** directory add the files *leon3_bprint.c and leon_uart.c* from the previous practice and also the files *leon3_ev_handling.S leon3_ev_handling.c and leon3_hw_irqs.c* that you will find in the file linked as *prac3_sources* in the web page. The content of each of these files is determined below:

leon3_ev_handling.c

It implements the following system routines for event management:

- `leon3_install_hwirq_handler`. Function that installs a user routine for handling an external interrupt following the structure of Figure 3. The prototype of the function is shown below, where `hwirq_number` is the vector number and `handler` is a pointer to the routine to be installed:

```
uint8_t leon3_install_hwirq_handler(     uint8_t hwirq_number,
                                void (* handler) (void));
```

- `leon3_set_trap_handler`. Function for the management of events that directly installs the trap attention routine, indicating the vector number of the trap (`trap_vector_number`), and the `handler` that is a pointer to the routine to install. The prototype of this function is the following:

```
uint8_t leon3_set_trap_handler(    uint8_t trap_vector_number,
                                void (* handler) (void));
```

In this practice we will use both functions to control the response to different system events.

leon3_hw_irqs.c

This module implements user routines for the management of **external interrupt control registers** (not for exceptions or traps). It partially implements the following functions:

- `leon3_mask_all_irqs`. Function that allows masking the 15 external interrupt levels (from 1 to 15) by setting the IMASK register (located at address 0x80000240) to 0. Its prototype is as follows:

```
void leon3_maks_all_irqs();
```

- `leon3_unmask_all_irqs`. Function that allows to unmask the 15 external interrupt levels (from 1 to 15) by setting the IMASK register (located at address 0x80000240) to 0xFE. Its prototype is as follows:

```
void leon3_unmaks_all_irqs();
```

- `leon3_mask_irq`. Function that allows masking **one of the 15** external interrupt **levels** by setting the bit corresponding to the level to 0 in the IMASK register (**only that bit, the rest must remain with the value they have**). The level number must be supplied via the `irq_level` parameter. Its prototype is as follows:

```
uint8_t leon3_mask_irq(uint8_t irq_level);
```

- `leon3_unmask_irq`. Function that allows unmasking **one of the 15** external interrupt **levels** by setting the bit corresponding to the level level in the IMASK register to 1 (**only that bit, the rest must remain with the value they have**). The level number must be supplied via the `irq_level` parameter. Its prototype is as follows:

```
uint8_t leon3_unmask_irq(uint8_t irq_level);
```

- `leon3_force_irq`. Function that allows forcing the triggering of one of the 15 external interrupt levels by setting to 1 in the IFORCE register (register located at address 0x80000208) the bit corresponding to the level (**only that bit, the rest must remain with the value they have**). The level number must be supplied via the `irq_level` parameter. Its prototype is as follows:

```
uint8_t leon3_force_irq(uint8_t irq_level);
```

- `leon3_clear_irq`. Function that allows clearing the *pending* bit of one of the 15 external interrupt levels by setting 1 in the ICLEAR register (register located at address 0x8000020C). The function changes only the bit corresponding to the level (**the rest of the bits are set to zero so as not to change the status of its *pending* bit**). The level number must be supplied via the `irq_level` parameter. Its prototype is as follows:

```
uint8_t leon3_clear_irq(uint8_t irq_level);
```

leon3_ev_handling_asm.S

An assembler file that implements the following event handling functions:

- `leon3_trap_handler_enable_irqs a` **trap handler routine** that enables all interrupts that are not masked in the IMASK register. It does this by setting the Priority Interrupt Level (PIL) of the Processor Status Register (PSR) to 0.

- `leon3_trap_handler_disable_irqs a` **trap handler routine** that disable all interrupts, regardless of how their mask is configured in the IMASK register. It does this by setting the Priority Interrupt Level (PIL) of the Processor Status Register (PSR) to 15.

- `leon3_sys_call_enable_irqs` **system call**, made through a *TRAP*, which allows the `leon3_trap_handler_enable_irqs handler` routine to be executed in supervisor mode.

- `leon3_sys_call_disable_irqs(void)` **system call**, made through a *TRAP*, which allows the `leon3_trap_handler_disable_irqs handler` routine to be executed in supervisor mode.

To complete the file configuration, in the **include** directory, add the files *leon3_bprint.h and leon_uart.h* from the previous practice. Also add the files *leon3_asm.h leon3_ev_handling.h and leon3_hw_irqs.h* that you will also find in the file linked as *prac3_sources* in the web page. The content of these last three files is as follows:

- *leon3_asm. h* Utility assembler macros for the definition of the functions implemented in *leon3_ev_handling_asm.S*

- *leon3_hw_irqs.h* File for the declaration of the functions defined in *leon3_hw_irqs. c*

- *leon3_ev_handling.h* File for the declaration of the functions defined in *leon3_ev_handling. c* and *leon3_ev_handling_asm.S*

6. Prac3. Task to complete.

1. Complete the `leon3_mask_irq, leon3_unmask_irq and leon3_force_irq` functions to properly handle masks and the register that forces interrupt triggering.

2. Implement in the file *leon3_bprint.c* the function `leon3_print_uint32` with the following prototype. This function implements a function analogous to the

`leon3_print_uint8` function performed in the previous practice, but working with a 32-bit integer. (Add this declaration to *leon3_bprint.h* so that the function can be used.)

```
 int8_t leon3_print_uint32(   uint32_t i);
```

3. Check the validity of the implementation with the following main programme by completing the missing parts:

```
#include "leon3_uart.h"
#include "leon3_bprint.h"
#include "leon3_hw_irqs.h"
#include "leon3_ev_handling.h"

void device_hw_irq_level_1_handler(void)

      leon3_print_string("Device HW IRQ user handler \n");
}

int main()
{

  //Install as trap 0x83 handler the routine
  // that enables interrupts
  leon3_set_trap_handler(0x83,leon3_trap_handler_enable_irqs);

  //Install trap handler that 0x83 the routine
  // that disables interrupts
  leon3_set_trap_handler(0x84,leon3_trap_handler_disable_irqs);


  //system call to disable interrupts
  leon3_sys_call_disable_irqs();

   //COMPLETE
   //

     //Mask all interrupts


     //Install device_hw_irq_level_1_handler as
     // level 1 interrupt user handler


      //Unmask level 1 interrupt


      //System call to enable interrupts


     //Force level 1 interrupt


   //END COMPLETE

    return 0;

}
```

Check that the output given on the screen is as follows:

```
System HW IRQ handler
Irq level = 1;
Device HW IRQ user handler
```

4. Repeat the execution putting a breakpoint in the first instruction of the `leon3_trap_handler_enable_irqs` function defined in *leon3_ev_handling_asm.S* and another one in the **call** to the `leon3_sys_call_enable_irqs` function **that is in the main**, and that has been defined as *wrapper* of trap 3 (vector 0x83). Execute **step-by-step** (Step-into, F5 key) to check the behaviour. Which function does the program jump to after the assembler instruction **ta** ?

5. Repeat the execution by masking the level 1 interrupt (use `leon3_mask_irq`) before `leon3_force_irq(1)` and check that no message is generated on the screen.

6. Repeat commenting out the call to `leon3_sys_call_enable_irqs()`, checking that no message is generated either.

7. Execute the following code at the end of the program. What happens? Knowing that when a division by 0 occurs the library routine that implements the division calls trap 0x82, how would you use the `leon3_set_trap_handler` function to make the program not hang and instead print a message saying "error, division by zero"?

```
uint8_t i;
uint8_t j;
for(i=10; i>0; i--)
  j=j/(i-9);
```

7. Prac3b. Task to be completed.

In this part of the practice we are going to modify the UART-A driver from practice 2 in order to be able to trigger interrupts when receiving data. These interrupts are associated with external interrupt level 2. In addition, we are going to learn how to configure the UART-A in LOOP-BACK mode to force the data that we are trying to transmit to be redirected back to the input, so that we can simulate the reception of data through the UART-A and its management by means of interruptions.

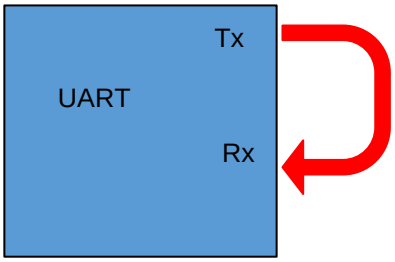

Figure 5. UART-A in LOOP-BACK mode

1. Create a new project called Prac3b, in which we will copy all the files of the Prac3 project, following the same directory structure as in the Prac3 project (if you have the project configured in SVN, be careful with copying directories because you will have problems when uploading Prac3b).

2. Comment out the line defining the VERBOSE_TRAP_HANDLER macro in the *leon3_ev_handling. h* file (This will prevent the interrupt handler from sending characters over the UART).

```
#ifndef TRAPS_H_
#define TRAPS_H_

//#define VERBOSE_TRAP_HANDLER
```

3. Add to the *leon3_uart.h* file the declaration of the functions `leon3_getchar`, `leon3_uart_ctrl_rx_enable`, `leon3_uart_ctrl_rx_irq_enable` and `leon3_uart_ctrl_config_rxtx_loop` as shown below:

```
char leon3_getchar();

void leon3_uart_ctrl_rx_enable();

void leon3_uart_ctrl_rx_irq_enable();

void leon3_uart_ctrl_config_rxtx_loop(uint8_t set_rxtxloop);
```

4. Add to the file *leon3_uart.c* the code for the functions `leon3_getchar`, `leon3_uart_ctrl_rx_enable`, `leon3_uart_ctrl_rx_irq_enable` and `leon3_uart_ctrl_config_rxtx_loop`, according to the following definition:

   - `leon3_getchar` returns the value (converted to uint8_t) found in the Data register of the UART-A (see description of this register in practice 2).

   - `leon3_uart_ctrl_rx_enable`: sets the `Receiver_enable` field of the UART-A control register to 1 **without modifying the rest of the fields of that register** enabling the reception of data through the UART. (See the following figure to locate the position of the field).

   - `leon3_uart_ctrl_rx_irq_enable` sets the UART-A control register field `Receiver_interrupt_enable` to 1 **without modifying the other fields in that register** enabling interrupts after receiving data via the UART. (See the figure below to locate the position of the field).

   - `leon3_uart_ctrl_config_rxtx_loop (uint8_t set_rxtxloop)` function that receives as parameter the value to be set in the `loop_back` field of the UART-A control register (see figure 5), so that if `set_rxtxloop` is 1, the LOOP_BACK mode of the UART is enabled and if it is 0, it is disabled.

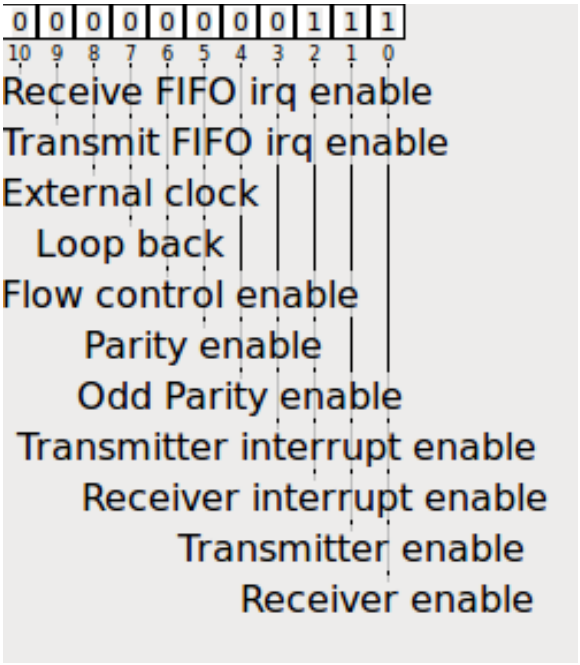

Figure 6. Description of the UART-A Control register

5. Complete the following main program, so that for the external level interrupt2, associated with the reception of a character by the UART-A, the `uart_rx_irq_handler` is installed. In addition, once the handler is installed, and the UART is configured with the functions `leon3_uart_ctrl_rx_enable`, `leon3_uart_ctrl_rx_irq_enable` and `leon3_uart_ctrl_config_rxtx_loop`, the interrupts must be enabled, and the level 2 interrupt must be unmasked.

```
#include "leon3_uart.h"
#include "leon3_bprint.h"
#include "leon3_hw_irqs.h"
#include "leon3_ev_handling.h"

uint8_t irq_counter=0;
char RxChars[8];

/*This function shall be used as a handler for the interrupt
generated by the reception of a data through the serial port.
The function stores in RxChars the first 8 characters that are
received, and forwards the successor on the same channel Tx*/.

void uart_rx_irq_handler(void){

      char aux;
      aux=leon3_getchar();

      if(irq_counter< 7){
            RxChars[irq_counter]=aux;
            aux++;
            leon3_putchar(aux);
      }
      irq_counter++;
}
```

```
int main()
{
      uint8_t i;

  //Install as trap 0x83 handler the routine // that enables
interrupts
  leon3_set_trap_handler(0x83,leon3_trap_handler_enable_irqs);

  //Install trap handler that 0x83 the routine // that disables
interrupts
  leon3_set_trap_handler(0x84,leon3_trap_handler_disable_irqs);


  //system call to disable interrupts
leon3_sys_call_disable_irqs();

      //COMPLETE by installing     the uart_rx_irq_handler
routine as  //level 2 interrupt handler following the same
      //pattern as      in practice 3a









      //END COMPLETE

   //Enable loop-back
   leon3_uart_ctrl_config_rxtx_loop(1);

   //Enable receive interrupt by UART
   leon3_uart_ctrl_rx_irq_enable();

   //Enable reception by the UART
   leon3_uart_ctrl_rx_enable ();


   //COMPLETE enabling interrupts and //unmasking level 2
interrupt


   //END COMPLETE
   leon3_putchar('A');

   //Sounding if all 8 characters have been received
   while(irq_counter < 7)
            ;

   //After receiving the 8 characters, I configure the UART
    // no loop-back
   leon3_uart_ctrl_config_rxtx_loop(0);

   //Send back the 8 characters that were received,
   //but without loop-back, so they will appear on the screen
   for(i=0;i<8;i++)
            leon3_putchar(RxChars[i]);
```

```
    leon3_putchar('\n');

    //Wait until all characters have been sent.
    while(!leon3_uart_tx_fifo_is_empty())
            ;
    return 0;
}
```

6. Check that once all the changes indicated in the previous points have been made, the screen output shows the string ABCDEFGH as shown below.

```
allocated 2048 KiB ROM memory
icache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
dcache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
gdb interface: using port 1234
Starting GDB server. Use Ctrl-C to stop waiting for connection.
connected
ABCDEFGH
gdb: disconnected
gdb interface: using port 1234
Starting GDB server. Use Ctrl-C to stop waiting for connection.
```

7. Which line would you have to change so that the screen output would be 12345678, instead of ABCDEFGH?