Laboratory Exercise 8

Using Interrupts with Assembly Language and C code

The purpose of this exercise is to investigate the use of interrupts for the ARM A9 processor, using both assembly language and C code. To do this exercise you need to be familiar with the exceptions processing mechanisms of the ARM processor, and with the operation of the ARM Generic Interrupt Controller (GIC). These concepts are discussed in the tutorials *Introduction to the ARM Processor*, and *Using the ARM Generic Interrupt Controller*. You should also read the parts of the DE0-Nano-SoC Computer documentation that pertain to the use of exceptions and interrupts.

Part I

Consider the main program shown in Figure 1. The code first specifies the exceptions vector table for the ARM processor using a code section called *.vectors*. Only the IRQ vector is set up for this program. The .**word** Assembler directive is used to create placeholders (0's) for all of the vectors except IRQ, which is set to branch to an interrupt service routine called SERVICE_IRQ. In the *.text* section of the main program the stack pointers, for both interrupt mode and supervisor mode, are initialized, the generic interrupt controller (GIC) is configured, and the pushbutton KEYs port is set up to generate interrupts. Finally, the main program enables interrupts in the processor. You are to fill in the code that is not shown in the figure.

The main program in Figure 1 consists of an endless loop that comprises a few logical instructions. The code in this loop does not serve any useful purpose other than to provide a specific set of instructions that are being executed repeatedly: logical AND, Exclusive-OR, logical OR, and branch. When a pushbutton KEY is pressed to cause an interrupt, the main program may be executing any of the instructions in the loop. After this instruction is completed, the processor will branch to the interrupt handler. Figure 2 shows most of the code for the interrupt handler. It passes to the KEYs interrupt service routine, as a parameter in register R0, the machine code of the next instruction to be executed in the main program. You have to write the KEYs interrupt service routine. It should display on the Terminal window of the Monitor Program which instruction will be executed when the processor returns to the main program. An example of output on the Terminal window after several KEY presses might be:

AND
ORR
B
EOR
EOR
AND
. . .

```
                .section    .vectors, "ax"
                .word       0               // reset vector
                .word       0               // undefined instruction vector
                .word       0               // software interrrupt vector
                .word       0               // aborted prefetch vector
                .word       0               // aborted data vector
                .word       0               // unused vector
                B           SERVICE_IRQ     // IRQ interrupt vector
                .word       0               // FIQ interrupt vector

                .text
                .global     _start
_start:
                /* Set up stack pointers for IRQ and SVC processor modes */
                . . . code not shown
                BL          CONFIG_GIC      // configure the ARM generic interrupt controller

                /* Configure the pushbutton KEYs port to generate interrupts
                . . . code not shown
                /* Enable IRQ interrupts in the ARM processor */
                . . . code not shown
MAIN_LOOP:
                AND         R0, R1, R2
                EOR         R3, R4, R5
                ORR         R6, R7, R8
                AND         R8, R7, R6
                EOR         R5, R4, R3
                ORR         R2, R1, R0
                B           MAIN_LOOP

                .end
```

Figure 1: Main program and interrupt service routine.

Perform the following:

1. Create a new folder to hold your Monitor Program project for this part. Create a file, such as *part1.s*, and type the assembly language code for the main program into this file.

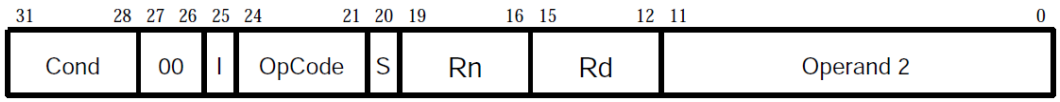
2. Create any other source code files you may want, and write the code for the *CONFIG_GIC* subroutine that initializes the GIC. Set up the GIC to send interrupts to the ARM processor from the pushbutton KEYs port.

3. To implement the KEYs interrupt service routine, you need to know the ARM instruction-encodings that are shown in Figure 3. Part a of the figure gives the encoding of data processing instructions, which include AND, EOR, and ORR, and Figure 3b displays the encoding of branch instructions. For all of the relevant instructions used here the Cond field will have the value 1110, which represents the condition *always*. In part a of Figure 3 the I field indicates an *immediate* operand. This field

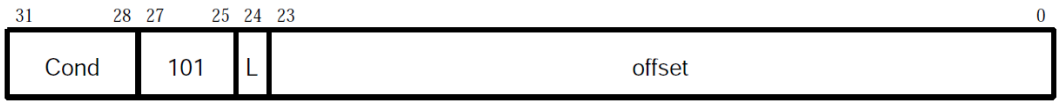
```
/* Define the IRQ exception handler */
SERVICE_IRQ:  PUSH    {R0-R7, LR}

              LDR     R4, =0xFFFEC100   // GIC CPU interface base address
              LDR     R5, [R4, #0x0C]   // read the ICCIAR in the CPU interface

CHECK_KEYS:   CMP     R5, #73           // check the interrupt ID
UNEXPECTED:   BNE     UNEXPECTED        // if not recognized, stop here

              /* load into register R0 the machine code of the instruction that will be
                 executed on return from this interrupt service routine */
              · · · code not shown
              BL      KEY_ISR

EXIT_IRQ:     STR     R5, [R4, #0x10]   // write to the End of Interrupt Register

              POP     {R0-R7, LR}
              SUBS    PC, LR, #4        // return from exception

              .end
```

Figure 2: The IRQ Exception handler.

is 1 if *Operand2* is an immediate constant, otherwise 0. The OpCode field specifies the type of data processing instruction: it has the values 0000 for AND, 0001 for EOR and 1100 for ORR. The S bit is 1 if the instruction modifies the condition codes, otherwise 0. Fields Rn and Rd specify the first operation register and destination register, respectively. Field Operand2 specifies the flexible second operand. When a register is used for this operand, it is identified in bits $3-0$. In Figure $3b$ the L bit is 0 for a Branch instruction and 1 for Branch-with-link. The offset field is a 24-bit signed number in 2's complement.



(a) Data processing instructions



(b) Branch instructions

Figure 3: The machine code format.

To display the name of an instruction mnemonic on the Terminal window you may wish to declare some strings of data as indicated below, and write the characters in these strings to the JTAG UART that is connected to the Terminal window:

```
AND:    .asciz    "AND\n"
ORR:    .asciz    "ORR\n"
EOR:    .asciz    "EOR\n"
BAL:    .asciz    "B\n"
```

4. Make a new Monitor Program project in the folder where you stored your source-code files. In the Monitor Program screen illustrated in Figure 10, choose Exceptions in the *Linker Section Presets* drop-down menu. In the screen of Figure 5, select JTAG_UART_for_ARM_0 as the *Terminal device*. Refer to Exercise 2, Part IV, for information on using the JTAG UART to communicate with the Monitor Program's Terminal window.

5. Compile, download, and test your program.

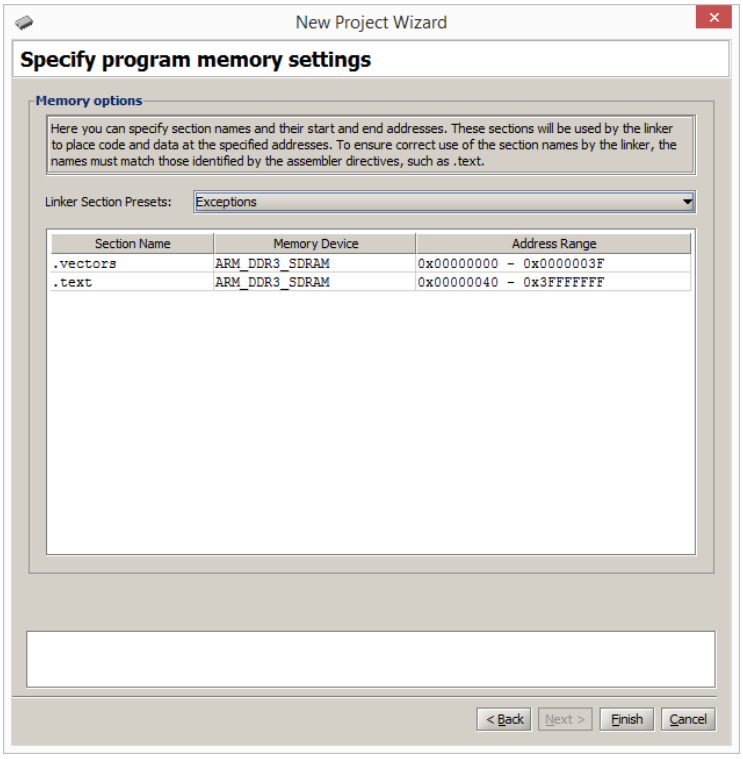


Figure 4: Selecting the Exceptions linker section.

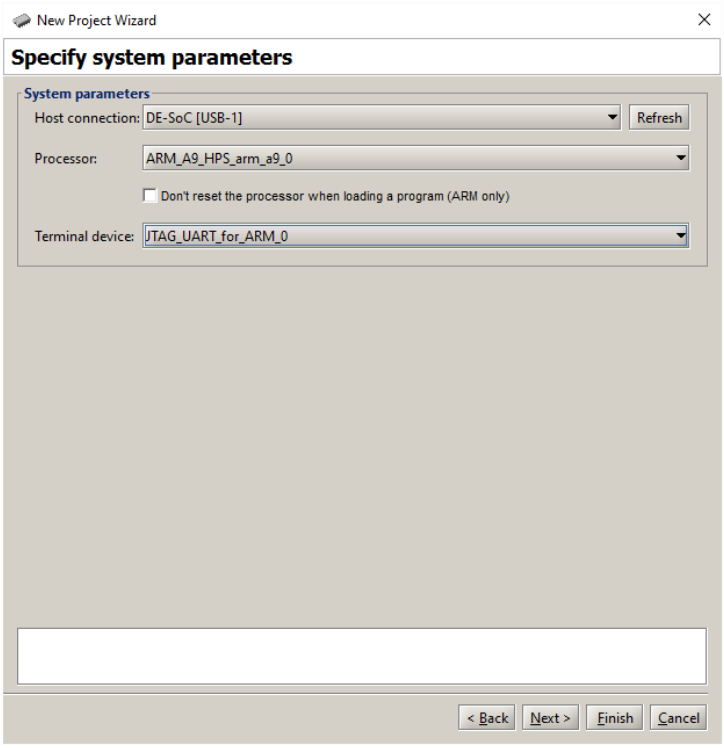


Figure 5: Specifying the *Terminal device*.

Part II

For this part you are to modify your code from Part I so that additional information in displayed on the Terminal window when a pushbutton KEY is pressed. All of your changes for this part should be done in the interrupt service routine for the pushbutton KEYs. You should not have to change any other files. For Part I you were required to display only the mnemonic of the next instruction, such as AND or EOR, that is to be executed in the main program on return from an interrupt. For this part you are to extend your solution so that it also displays the names of the register arguments of each instruction. Use the machine encodings shown in Figure 3 to find the required information. An example of output produced on the Terminal window after several KEYs have been pressed might be:

```
ORR     R2, R1, R0
EOR     R3, R4, R5
AND     R0, R1, R2
B       0xFFFFF8
ORR     R6, R7, R8
AND     R8, R7, R6
...
```

For logical instructions you can assume that only the register numbers used in the main program from Figure 1 have to be supported. Thus, your code only has to be able to display register indices which are single-digit numbers from 0 to 9. For a branch instruction you are to show the 24-bit branch offset as a 2's-complement number, as indicated in the example output that is provided above.

Part III

Consider the C code shown in Figure 6. It contains an endless loop that repeatedly calls a function named *KEY_pressed*, not shown in the figure, which checks to see if a KEY is currently being pressed. If so, it waits for the KEY to be released and then returns 1. Otherwise, if no KEY was pressed, 0 is returned. When *KEY_pressed* returns 1, the main program calls a function named *doit*. This function is supposed to print, on the Terminal window, the mnemonic of the instruction that will be executed on return from the *doit* function. In the main program *inline assembly* commands are used to insert specific instructions following each call to *doit*. For example, the first call to *doit* is followed by the instruction AND R0, R1, R2, the second call to *doit* is followed by the instruction EOR R3, R4, R5, and so on.

Part of the code for the *doit* function is given in Figure 6. It uses two inline assembly commands. The command

```
asm("LDR R0, [LR]" : : : "r0")
```

loads the machine code of the instruction pointed to by the link register into register R0. The argument "r0" in this command informs the C compiler that the contents of register R0 will be changed as a result of executing the instruction. The command

```
asm("MOV %0, R0" : "=r" (machine_code) : : )
```

copies the contents of register R0 into the variable *machine_code*. The argument %0 in this command is set to whichever register, for example register R1, is chosen by the C compiler to hold the value of the variable *machine_code*.

Make sure that you understand how the inline assembly commands work. Documentation on these commands can be found by searching on the web for *gnu inline assembly ARM*, or something similar.

Perform the following:

1. Type the C code for the main program into a file, for example *part3.c*. Also, add your own code for the *KEY_pressed* function.
2. Complete the C code for the *doit* function. You should display on the Terminal window the mnemonic of the instruction corresponding to the *machine_code* variable. Refer to Figure 3 for the machine code formats. You should use the *printf* library function to display the result produced by the program. To use the *printf* function include the *stdio.h* library as shown in Figure 6.
3. Make a new Monitor Program project for this part of the exercise. In the Monitor Program screen shown in Figure 7 set the *Terminal device* to Semihosting. This setting causes the output of library functions like *printf* to appear in the *Terminal* window of the Monitor Program graphical user interface.
4. Compile, download, and test your program.

```
#include <stdio.h>

/* Function prototypes */
int KEY_pressed(void);
void doit(void);

/* This program demonstrates the use of inline assembly code in C code */
int main(void)
{
    while (1)                                          // endless loop
    {
        if (KEY_pressed ()) doit ();
        asm("AND R0, R1, R2");
        if (KEY_pressed ()) doit ();
        asm("EOR R3, R4, R5");
        if (KEY_pressed ()) doit ();
        asm("ORR R6, R7, R8");
        if (KEY_pressed ()) doit ();
        asm("AND R8, R7, R6");
        if (KEY_pressed ()) doit ();
        asm("EOR R5, R4, R3");
        if (KEY_pressed ()) doit ();
        asm("ORR R2, R1, R0");
        if (KEY_pressed ()) doit ();
    }
}

int KEY_pressed( )
{
    ··· code not shown
}

void doit( )
{
    unsigned int machine_code;
    // get the machine code of the next instruction on return from this subroutine
    asm("LDR R0, [LR]" : : : "r0");                    // read machine code into R0
    asm("MOV %0, R0" : "=r" (machine_code) : : );      // copy R0 into variable machine_code

    ··· code not shown
}
```

Figure 6: Main program for Part III.

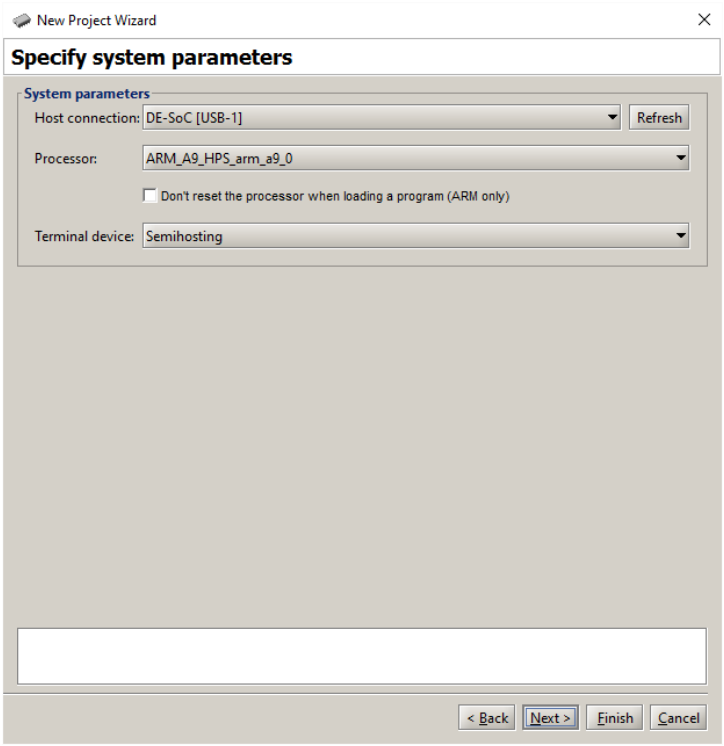


Figure 7: Specifying the *Terminal device*.

Part IV

In this part you are to repeat the tasks given in Parts I and II, but using C code rather than assembly-language code.

Consider the main program shown in Figure 8. The program first initializes the ARM A9 stack pointer for IRQ (interrupt) mode by calling a subroutine named *set_A9_IRQ_stack()*. This step is necessary because, although the C compiler automatically generates code that initializes the SVC-mode (supervisor mode) stack pointer, the C compiler does not generate code to initialize the IRQ-mode stack pointer. The main program then calls subroutines *config_GIC()* to initialize the generic interrupt controller (GIC), and *config_KEYs()* to initialize the pushbutton KEYs port so that it will generate interrupts. Finally, a subroutine *enable_A9_interrupts()* is called to unmask IRQ interrupts in the ARM processor.

After completing the initialization steps described above, the main program executes an endless loop that consists of several logical instructions. Inline assembly commands, described in Part III, are used to specify the logic instructions.

Perform the following:

```
int main(void)
{
    set_A9_IRQ_stack ();            // initialize the stack pointer for IRQ mode
    config_GIC ();                  // configure the general interrupt controller
    config_KEYs ();                 // configure pushbutton KEYs to generate interrupts

    enable_A9_interrupts ();        // enable interrupts in the A9 processor

    while (1)                       // wait for an interrupt
    {
        asm("AND R0, R1, R2");
        asm("EOR R3, R4, R5");
        asm("ORR R6, R7, R8");
        asm("AND R8, R7, R6");
        asm("EOR R5, R4, R3");
        asm("ORR R2, R1, R0");
    }
}

/* Initialize the banked stack pointer register for IRQ mode */
void set_A9_IRQ_stack(void)
{
    ... code not shown
}
/* Configure the Generic Interrupt Controller (GIC) */
void config_GIC(void)
{
    ... code not shown
}
/* Set up the pushbutton KEYs port in the FPGA */
void config_KEYs(void)
{
    ... code not shown
}
/* Turn on interrupts in the ARM processor */
void enable_A9_interrupts(void)
{
    ... code not shown
}
```

Figure 8: Main program for Part IV.

1. Create a new folder to hold your Monitor Program project for this part. Create a file, such as *part1.c*, for your main program, and create any other source-code files that you may wish to use. Write the code for the subroutines that are called by the main program. For the *config_GIC()* subroutine set up the GIC to send interrupts to the ARM processor from the pushbutton KEYs port.

2. Figure 9 gives part of the C code required for the interrupt handler. It is declared with the __**attribute**__ specification *interrupt*, and has the special name *__cs3_isr_irq*. Using this declaration allows the C compiler to recognize the code as being the IRQ interrupt handler. The compiler generates an entry that branches to this code in the ARM exception-vector table. Complete the missing part of this code. You have to obtain the machine code of the next instruction to be executed on return from the interrupt, and pass this machine code to the interrupt service routine for the pushbutton KEYs.

   You have to write the code for the *pushbutton_isr()* interrupt service routine. Your code should perform the same task as in Part II of this exercise. That is, you are to print on the Terminal window the mnemonic, and register indices, of the instruction that will be executed on return from the interrupt. You can make the same assumptions as in Parts I and II, namely that only the logical instructions, plus the branch instruction, have to be handled by your code. For the branch instruction display the 24-bit branch offset as its argument.

3. Make a new Monitor Program project in the folder where you stored your source-code files. As discussed for Part III, in the Monitor Program screen shown in Figure 7 set the *Terminal device* to Semihosting. Also, in the Monitor Program screen illustrated in Figure 10, make sure to choose Exceptions in the *Linker Section Presets* drop-down menu.

4. Compile, download, and test your program.

```
/* Define the IRQ exception handler */
void __attribute__ ((interrupt)) __cs3_isr_irq (void)
{
    /* Read the ICCIAR from the CPU interface in the GIC */
    int address = MPCORE_GIC_CPUIF + ICCIAR;
    int int_ID = *((int *) address);

    if (int_ID == KEYS_IRQ)          // check if interrupt is from the KEYs
    {
        ... code not shown
        pushbutton_ISR ( . . . );
    }
    else
        while (1);                   // if unexpected, then stay here

    /* Write to the End of Interrupt Register (ICCEOIR) */
    address = MPCORE_GIC_CPUIF + ICCEOIR;
    *((int *) address) = int_ID;

    return;
}
```

Figure 9: IRQ Exception handler.

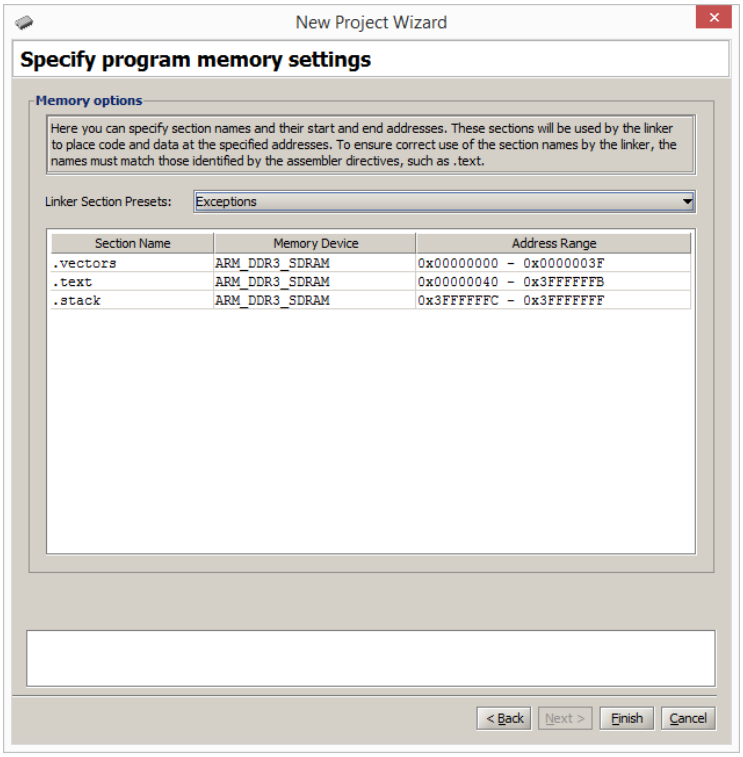


Figure 10: Selecting the Exceptions linker section.