ECSE 324: Computer Organization
Lab 3: I/O

Due Tuesday, November 11, at 11:59 pm

Abstract

This lab builds on Lab 2 by beginning to investigate assembly implementations of I/O interactions. Subroutines for reading and writing I/O peripherals (i.e., *device drivers*) are often still written in assembly or low-level C. You will start by writing a library of functions for interacting with slider switches, LEDs, 7-segment displays, and push-buttons. You will first use this library to implement a simple interactive display; you will subsequently use the library to re-implement the interactive display using interrupts.

Summary of Deliverables

- Source code for:
  - An interactive rotating HEX display program using polling
  - An interactive rotation HEX display program using interrupts
- Performance analysis report, no longer than one page (10 pt font, 1” margins)

Please submit your source code to **MyCourses** in a single .zip archive, using the following file naming conventions:

- Archive: `StudentID_FullName_Lab3_src.zip`
- Code: `part1.s`, `part2.s`

Please submit your report to **Crowdmark** as a PDFs, using the following file naming convention:

- Report: `StudentID_FullName_Lab3_report.pdf`

Grading Summary

- 80% Software test cases
- 20% Report

Changelog

- 22-Oct-2025 Initial release.

Part 1: Interactive Display

For this part, it is necessary to refer to sections 2.9.1 - 2.9.4 (pp. 7 - 9) and 3.4.1 (p. 14) in the DE1-SoC Computer Manual.

Brief overview

The hardware setup of the DE1-SoC's I/O components is straightforward. The system has designated addresses in memory that are connected to hardware circuits on the FPGA through parallel ports, and these hardware circuits, in turn, interface with the physical I/O components. In most cases, the FPGA hardware simply maps the I/O terminals to the memory address designated to it. There are several parallel ports implemented in the FPGA that support input, output, and bidirectional transfers of data between the ARM A9 processor and I/O peripherals. For instance, the state of the slider switches is available to the FPGA on a bus of 10 wires which carry either a logical '0' or '1'. The state of the slider switches is then stored in the memory address reserved for the slider switches (`0xFF200040` in this case).

It is useful to have slightly more sophisticated FPGA hardware in some cases. For instance, in the case of the push-buttons, in addition to knowing the state of the button, it is also helpful to know whether a falling edge is detected, signaling a keypress. This can be achieved with a simple edge detection circuit in the FPGA.

Getting Started: Drivers for slider switches and LEDs

Access to the memory addresses designated for I/O interfaces is best facilitated by what are called *device drivers*. Drivers are subroutines that ease the process of reading from and writing to I/O interface addresses, thereby manipulating the state of, and data associated with, a given peripheral. When writing drivers, it is critical that you follow the subroutine calling conventions presented in this course.

Note: *not naming your functions exactly as follows may result in lost points during auto-grading!*

1- Slider Switches: Create a new subroutine labeled *read_slider_switches_ASM* that reads the value from the memory location designated for the slider switches' data (`SW_ADDR`) and stores it in A1, and then returns. Remember to use the subroutine calling convention and save processor state (by pushing and popping registers) if needed! *If there are fewer than four 32-bit arguments or return values, use registers, rather than the stack, for communication between caller and callee.*

2- LEDs: Create a new subroutine labeled *write_LEDs_ASM*. The *write_LEDs_ASM* subroutine writes the value in A1 to the LEDs' memory location (`LED_ADDR`), and then branches to the address contained in the LR.

To help you get started, the code for the slider switch and LED drivers have been provided below. Use them as templates for writing future drivers.

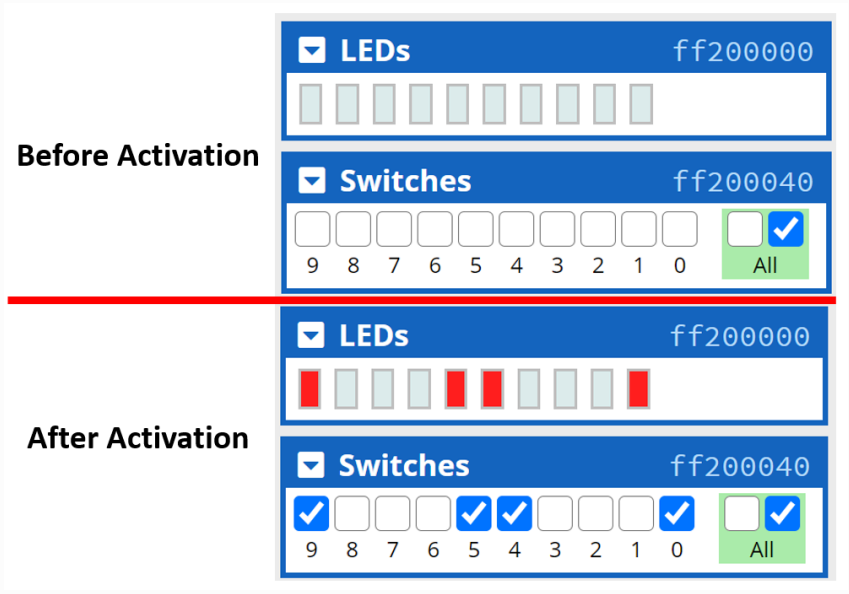
```
// Slider Switches Driver
// returns the state of slider switches in A1
// post- A1: slide switch state
.equ SW_ADDR, 0xFF200040
read_slider_switches_ASM:
    LDR A2, =SW_ADDR     // load the address of slider switch state
    LDR A1, [A2]         // read slider switch state
    BX  LR

// LEDs Driver
// writes the state of LEDs (On/Off) in A1 to the LEDs' control register
// pre-- A1: data to write to LED state
.equ LED_ADDR, 0xFF200000
write_LEDs_ASM:
    LDR A2, =LED_ADDR    // load the address of the LEDs' state
    STR A1, [A2]         // update LED state with the contents of A1
    BX  LR
```

Exercise

To acquaint yourself with using memory-mapped I/O, write an application that makes use of the *read_slider_switches_ASM* and *write_LEDs_ASM* subroutines to turn on/off the LEDs. Write an infinite loop that calls *read_slider_switches_ASM* and *write_LEDs_ASM* in order, turning on LEDs if the corresponding switch is on, and vice versa. Compile and Run your project, and then change the state of the switches in the online simulator to turn on/off the corresponding LEDs. Note that both the Switches and the LEDs panels are located on the top corner of your screen.

The figure below demonstrates the result of activating slider switches 0, 4, 5 and 9.



More Advanced Drivers: Drivers for HEX displays and push-buttons

Now that the basic structure of the drivers has been introduced, we can write more advanced drivers i.e., for HEX displays and push-buttons drivers.

1- HEX displays: There are six HEX (seven-segment) displays (HEX0 to HEX5) on the DE1-SoC Computer board.

Exercise

Write three subroutines to implement the functions listed below to control the HEX displays.

Note! The emulator will complain about word-aligned accesses if you use STRB to write to some HEX displays. This does not indicate an error has occurred. You may turn off *Device-specific warnings* in the settings, but be careful, as this may hide other warnings worth observing. If you leave this *Device-specific warnings* enabled, simply continue the execution of your code after the emulator-inserted breakpoint.

Note: *not naming your functions exactly as follows may result in lost points during auto-grading!*

HEX_clear_ASM: This subroutine turns off all the segments of the selected HEX displays. It receives the selected HEX display indices through register A1 as an argument.

HEX_flood_ASM: This subroutine turns on all the segments of the selected HEX displays. It receives the selected HEX display indices through register A1 as an argument.

HEX_write_ASM: This subroutine receives HEX display indices and an integer value, 0-15, to display. These are passed in registers A1 and A2, respectively. Based on the second argument (A2), the subroutine will display the corresponding hexadecimal digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) on the display(s).

The subroutines should check the first argument to determine which displays, HEX0-HEX5, have been selected, and write the value of the second argument to each. A loop may be useful here! The HEX display indices can be encoded based on a one-hot encoding scheme, e.g.,:

```
HEX0 = 0x00000001
HEX1 = 0x00000002
HEX2 = 0x00000004
HEX3 = 0x00000008
HEX4 = 0x00000010
HEX5 = 0x00000020
```

For example, you may pass `0x0000000C` to the *HEX_flood_ASM* subroutine to turn on all the segments of HEX2 and HEX3 displays, as 0xC = 0b1100:

```
mov A1, #0x0000000C
BL  HEX_flood_ASM
```

2- Pushbuttons: There are four pushbuttons (PB0 to PB3) on the DE1-SoC Computer board.

Exercise

Write seven subroutines to implement the functions listed below to control the pushbuttons:

Note: *not naming your functions exactly as follows may result in lost points during auto-grading!*

read_PB_data_ASM: This subroutine returns the indices of the pressed pushbuttons (the keys from the pushbuttons Data register). The indices are encoded based on a one-hot encoding scheme:

```
PB0 = 0x00000001
PB1 = 0x00000002
PB2 = 0x00000004
PB3 = 0x00000008
```

PB_data_is_pressed_ASM: This subroutine receives a pushbutton index as an argument. Then, it returns `0x00000001` if the corresponding pushbutton is pressed.

read_PB_edgecp_ASM: This subroutine returns the indices of the pushbuttons that have been pressed and then released (the edge bits from the pushbuttons' `Edgecapture` register).

PB_edgecp_is_pressed_ASM: This subroutine receives a pushbutton index as an argument. Then, it returns `0x00000001` if the corresponding pushbutton has been pressed and released.

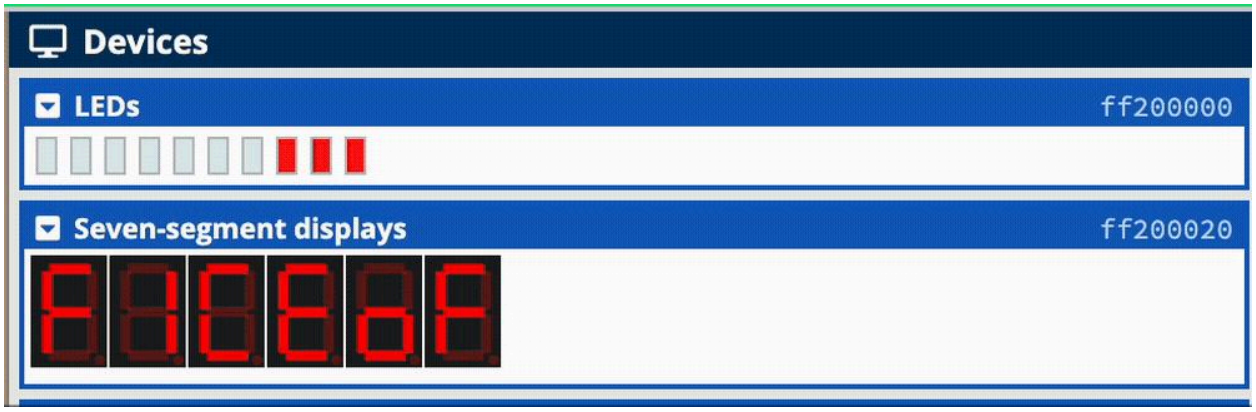
PB_clear_edgecp_ASM: This subroutine clears the pushbutton `Edgecapture` register. You can read the `edgecapture` register and write what you just read back to the `edgecapture` register to clear it.

enable_PB_INT_ASM: This subroutine receives pushbutton indices as an argument. Then, it enables the interrupt function for the corresponding pushbuttons by setting the interrupt mask bits to '1'.

disable_PB_INT_ASM: This subroutine receives pushbutton indices as an argument. Then, it disables the interrupt function for the corresponding pushbuttons by setting the interrupt mask bits to '0'.

Part 1 Deliverable: *Interactive display using polling*

In this task you will write assembly code that utilizes your drivers to implement a polling-based interactive display where characters move from one side to the other, stepping in response to push-button press-and-release, fall off the end, and wrap around to the other side.



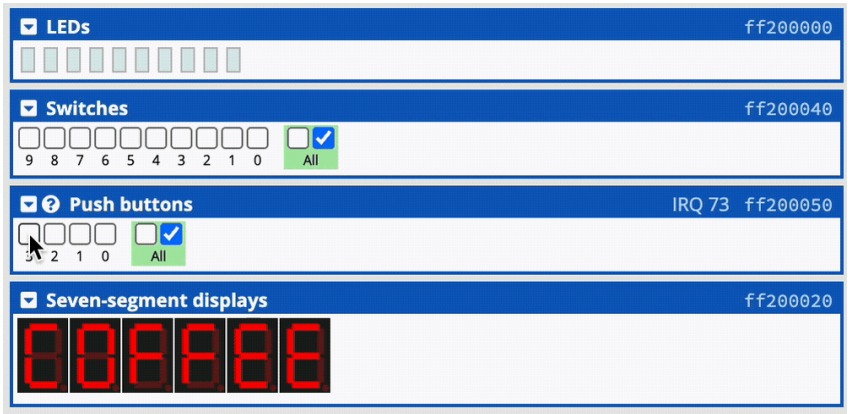
Using the following library of characters, the HEX display will show different messages based on slider switch input and button presses.

```
HEX_CODES:
    .byte   0b00111111, 0b00000110, 0b01011011, 0b01001111 // 0, 1, 2, 3
    .byte   0b01100110, 0b01101101, 0b01111101, 0b00000111 // 4, 5, 6, 7
    .byte   0b01111111, 0b01101111, 0b01110111, 0b01111100 // 8, 9, A, b
    .byte   0b00111001, 0b01011110, 0b01111001, 0b01110001 // C, d, E, F
```

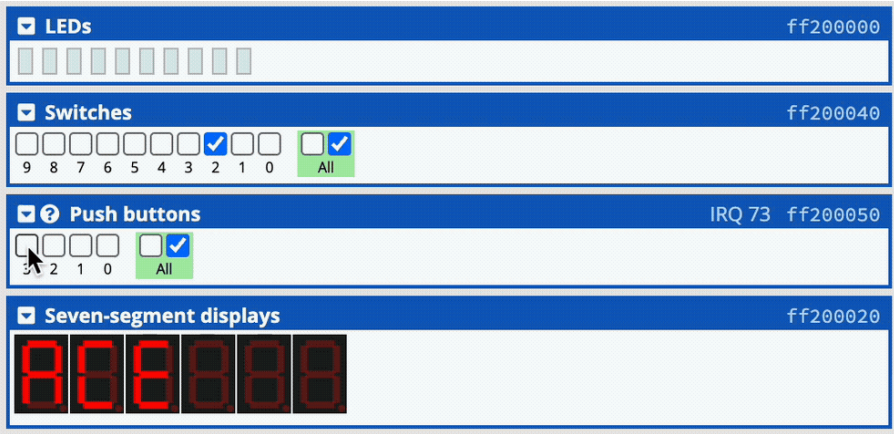
Note! The autograder will depend upon your implementing messages based on the above characters; deviating from it will result in lost points.

Specifications

- By default, a pre-programmed sequence, C0FFEE, is displayed.
- Slider switches change the characters in the message. Implement the following 4 messages, assuming the following one-hot encoding:
    1. 0x00: C0FFEE
    2. 0x01: CAFE5
    3. 0x02: CAb5
    4. 0x04: ACE
- Push-buttons rotate the sequence and modify the direction of movement:
    - A press and release of PB3 rotates the characters one position (to the left, by default); and,
    - A press and release of PB2 reverses direction of movement.
- LEDs display the number of times the sequence has been rotated, in binary, starting with 0 (no LEDs lit), and the least significant bit on the right.
    - When the message changes, the count should reset.
    - If the count ever exceeds 2047, all LEDs should remain lit until the count resets.



- All other switch combinations values should result in a blank display.
    - If no message is displayed, pressing and releasing PB2 and PB3 should not have any visible effect. (The count should not change.)
- If a message shorter than six characters is displayed, pad the message up to six characters with blank (unlit) HEX displays. (I.e., ACE above is followed by three blanks, which rotate through the display along with the printed characters.)



- When a new message is displayed, it should be left-justified (the first character starts in the left-most HEX display).

Summary

1. Implement a library of I/O functions and, using polling, an interactive display application (**`part1.s`**).

Part 2: Timers and Interrupts

Timers

For this part, it is necessary to refer to sections 2.4.1 (p. 3) and 3.1 (p. 14) in the DE1-SoC Computer Manual.

Timers are simply hardware counters that are used to measure time and/or synchronize events. They run on a known clock frequency that is programmable in some cases. Timers are usually (but not always) down counters, and by programming the start value, the time-out event (when the counter reaches zero) occurs at fixed time intervals.

ARM A9 Private Timer drivers

There is one ARM A9 private timer available on the DE1-SoC board. The timer uses a clock frequency of 200 MHz. You need to configure the timer before using it. To configure the timer, you need to pass three arguments to the “configuration subroutine”. The arguments are:

1- Load value: The ARM A9 private timer is a down counter and requires an initial count value. Use A1 to pass this argument.
2- Configuration bits: Use A2 to pass this argument. Read sections 2.4.1 (p. 3) and 3.1 (p. 14) in the De1-SoC Computer Manual carefully to learn how to handle the configuration bits. The configuration bits are stored in the Control register of the timer.

Exercise

Write three subroutines to implement the functions listed below to control the timers.

Note: *not naming your functions exactly as follows may result in lost points during auto-grading!*

ARM_TIM_config_ASM: This subroutine is used to configure the timer. Use the arguments discussed above.

ARM_TIM_read_INT_ASM: This subroutine returns the “F” value (`0x00000000` or `0x00000001`) from the ARM A9 private timer interrupt status register.

ARM_TIM_clear_INT_ASM: This subroutine clears the “F” value in the ARM A9 private timer Interrupt status register. The F bit can be cleared to 0 by writing a `0x00000001` to the interrupt status register.

Exercise

To test the functionality of your subroutines, write assembly that uses the ARM A9 private timer. Use the timer to count from 0 to 15 and show the count value on the HEX display (HEX0). You must increase the count value by 1 every time the “F” value is asserted (“F” becomes '1'). The count value must be reset when it reaches 15 (1, 2, 3, …, E, F, 0, 1, …). The counter should be able to count in increments of **0.25 second**. Remember, you must clear the timer interrupt status register each time the timer sets the “F” bit in the interrupt status register to 1 by calling the *ARM_TIM_clear_INT_ASM* subroutine. This code is especially important for the Part 2 deliverables.

Note! The emulator is not real-time capable (0.25 second of emulation of your software may take more than 0.25 second of wall-clock time). You may not be able to compare the passage of wall-clock time with your program’s behavior to determine whether it is functioning correctly.

Interrupts

For this part, it is necessary to refer to section 3 (pp. 13-17) in the DE1-SoC Computer Manual. Furthermore, detailed information about the interrupt drivers is provided in the “Using the ARM Generic Interrupt Controller” document available here.

Interrupts are hardware or software signals that are sent to the processor to indicate that an event has occurred that needs immediate attention. When the processor receives an interrupt, it pauses the code currently executing, handles the interrupt by executing code defined in an Interrupt Service Routine (ISR), and then resumes normal execution.

Apart from ensuring that high priority events are given immediate attention, interrupts also help the processor to utilize resources more efficiently. Consider the polling application from the previous section, where the processor periodically checked the pushbuttons for a keypress event. Asynchronous events such as this, if assigned an interrupt, can free the processor to do other work between events.

ARM Generic Interrupt Controller

The ARM generic interrupt controller (GIC) is a part of the ARM A9 MPCORE processor. The GIC is connected to the IRQ interrupt signals of all I/O peripheral devices that are capable of generating interrupts. Most of these devices are normally external to the A9 MPCORE, and some are internal peripherals (such as timers). The GIC included with the A9 MPCORE processor in the Altera Cyclone V SoC family handles up to 255 interrupt sources. When a peripheral device sends its IRQ signal to the GIC, the GIC can forward a corresponding IRQ signal to one or both of the A9 cores. Software code that is running on the A9 core can then query the GIC to determine which peripheral device caused the interrupt, and take appropriate action.

The ARM Cortex-A9 has several main modes of operation and the operating mode of the processor is indicated in the current processor status register **CPSR**. In this Lab, we only use **IRQ mode**. A Cortex-A9 processor enters IRQ mode in response to receiving an IRQ signal from the GIC.

Before interrupts can be used, software has to perform a number of steps:

1. Ensure that interrupts are disabled in the A9 processor by setting the IRQ disable bit in the CPSR to 1.
2. Configure the GIC. Interrupts for each I/O peripheral device that is connected to the GIC are identified by a unique interrupt ID.
3. Configure each I/O peripheral device so that it can send IRQ requests to the GIC.
4. Enable interrupts in the A9 processor, by setting the IRQ disable bit in the CPSR to 0.

An example assembly program that demonstrates use of interrupts follows. The program responds to interrupts from the pushbutton KEY port in the FPGA. The interrupt service routine

for the pushbutton KEYs indicates which KEY has been pressed on the HEX0 display. You can use this code as a template when using interrupts in the ARM Cortex-A9 processor.

First, you need to add the following lines at the beginning of your assembly code to initialize the exception vector table. Within the table, one word is allocated to each of the various exception types. This word contains branch instructions to the address of the relevant exception handlers.

```
.section .vectors, "ax"
B _start            // reset vector
B SERVICE_UND       // undefined instruction vector
B SERVICE_SVC       // software interrupt vector
B SERVICE_ABT_INST  // aborted prefetch vector
B SERVICE_ABT_DATA  // aborted data vector
.word 0             // unused vector
B SERVICE_IRQ       // IRQ interrupt vector
B SERVICE_FIQ       // FIQ interrupt vector
```

Then, add the following to configure the interrupt routine. Note that the different processor modes have their own stack pointers and link registers (see Figure 3 in “Using the ARM Generic Interrupt Controller”). At a minimum, you must assign initial values to the stack pointers of any execution modes that are used by your application. In our case, when an interrupt occurs, the processor enters IRQ mode. Therefore, we must assign an initial value to the IRQ mode stack pointer. Usually, interrupts are expected to be executed as fast as possible. As a result, on-chip memories are used in IRQ mode.
The following code illustrates how to set the stack to the A9 on-chip memory in IRQ mode, and call the subroutine to configure the GIC. After initialization, the program enters an infinite loop at IDLE; this is where you implement any functionality not implemented in interrupt handlers.

```
.text
.global _start

_start:
    /* Set up stack pointers for IRQ and SVC processor modes */
    MOV R1, #0b11010010      // interrupts masked, MODE = IRQ
    MSR CPSR_c, R1           // change to IRQ mode
    LDR SP, =0xFFFFFFFF - 3  // set IRQ stack to A9 on-chip memory
    /* Change to SVC (supervisor) mode with interrupts disabled */
    MOV R1, #0b11010011      // interrupts masked, MODE = SVC
    MSR CPSR, R1             // change to supervisor mode
    LDR SP, =0x3FFFFFFF - 3  // set SVC stack to top of DDR3 memory
    BL  CONFIG_GIC           // configure the ARM GIC
    // NOTE: write to the pushbutton KEY interrupt mask register
```

```
    // Or, you can call enable_PB_INT_ASM subroutine from previous task
    // to enable interrupt for ARM A9 private timer,
    // use ARM_TIM_config_ASM subroutine
    LDR R0, =0xFF200050      // pushbutton KEY base address
    MOV R1, #0xF             // set interrupt mask bits
    STR R1, [R0, #0x8]       // interrupt mask register (base + 8)
    // enable IRQ interrupts in the processor
    MOV R0, #0b01010011      // IRQ unmasked, MODE = SVC
    MSR CPSR_c, R0
IDLE:
    B IDLE // This is where you write your main program task(s)
```

Next, add the following subroutines to configure the Generic Interrupt Controller (GIC):

```
CONFIG_GIC:
    PUSH {LR}
/* To configure the FPGA KEYS interrupt (ID 73):
* 1. set the target to cpu0 in the ICDIPTRn register
* 2. enable the interrupt in the ICDISERn register */
/* CONFIG_INTERRUPT (int_ID (R0), CPU_target (R1)); */
/* NOTE: you can configure different interrupts
   by passing their IDs to R0 and repeating the next 3 lines */
    MOV R0, #73            // KEY port (Interrupt ID = 73)
    MOV R1, #1             // this field is a bit-mask; bit 0 targets cpu0
    BL CONFIG_INTERRUPT

/* configure the GIC CPU Interface */
    LDR R0, =0xFFFEC100    // base address of CPU Interface
/* Set Interrupt Priority Mask Register (ICCPMR) */
    LDR R1, =0xFFFF        // enable interrupts of all priorities levels
    STR R1, [R0, #0x04]
/* Set the enable bit in the CPU Interface Control Register (ICCICR).
* This allows interrupts to be forwarded to the CPU(s) */
    MOV R1, #1
    STR R1, [R0]
/* Set the enable bit in the Distributor Control Register (ICDDCR).
* This enables forwarding of interrupts to the CPU Interface(s) */
    LDR R0, =0xFFFED000
    STR R1, [R0]
    POP {PC}
```

```
/*
* Configure registers in the GIC for an individual Interrupt ID
* We configure only the Interrupt Set Enable Registers (ICDISERn) and
* Interrupt Processor Target Registers (ICDIPTRn). The default (reset)
* values are used for other registers in the GIC
* Arguments: R0 = Interrupt ID, N
* R1 = CPU target
*/
CONFIG_INTERRUPT:
    PUSH {R4-R5, LR}
/* Configure Interrupt Set-Enable Registers (ICDISERn).
* reg_offset = (integer_div(N / 32) * 4
* value = 1 << (N mod 32) */
    LSR R4, R0, #3    // calculate reg_offset
    BIC R4, R4, #3    // R4 = reg_offset
    LDR R2, =0xFFFED100
    ADD R4, R2, R4    // R4 = address of ICDISER
    AND R2, R0, #0x1F // N mod 32
    MOV R5, #1        // enable
    LSL R2, R5, R2    // R2 = value
/* Using the register address in R4 and the value in R2 set the
* correct bit in the GIC register */
    LDR R3, [R4]      // read current register value
    ORR R3, R3, R2    // set the enable bit
    STR R3, [R4]      // store the new register value
/* Configure Interrupt Processor Targets Register (ICDIPTRn)
* reg_offset = integer_div(N / 4) * 4
* index = N mod 4 */
    BIC R4, R0, #3    // R4 = reg_offset
    LDR R2, =0xFFFED800
    ADD R4, R2, R4    // R4 = word address of ICDIPTR
    AND R2, R0, #0x3  // N mod 4
    ADD R4, R2, R4    // R4 = byte address in ICDIPTR
/* Using register address in R4 and the value in R2 write to
* (only) the appropriate byte */
    STRB R1, [R4]
    POP {R4-R5, PC}
```

Then you need to define the interrupt service routines (ISR) using the following:

```
/*--- Undefined instructions ---------------------------------------*/
SERVICE_UND:
    B SERVICE_UND
/*--- Software interrupts ------------------------------------------*/
SERVICE_SVC:
    B SERVICE_SVC
/*--- Aborted data reads -------------------------------------------*/
SERVICE_ABT_DATA:
    B SERVICE_ABT_DATA
/*--- Aborted instruction fetch ------------------------------------*/
SERVICE_ABT_INST:
    B SERVICE_ABT_INST
/*--- IRQ ----------------------------------------------------------*/
SERVICE_IRQ:
    PUSH {R0-R7, LR}
/* Read the ICCIAR from the CPU Interface */
    LDR R4, =0xFFFEC100
    LDR R5, [R4, #0x0C] // read from ICCIAR
/* NOTE: Check which interrupt has occurred (check interrupt IDs)
   Then call the corresponding ISR
   If the ID is not recognized, branch to UNEXPECTED
   See the assembly example provided in the DE1-SoC Computer Manual
   on page 46 */
Pushbutton_check:
    CMP R5, #73
UNEXPECTED:
    BNE UNEXPECTED      // if not recognized, stop here
    BL KEY_ISR
EXIT_IRQ:
/* Write to the End of Interrupt Register (ICCEOIR) */
    STR R5, [R4, #0x10] // write to ICCEOIR
    POP {R0-R7, LR}
    SUBS PC, LR, #4
/*--- FIQ ----------------------------------------------------------*/
SERVICE_FIQ:
    B SERVICE_FIQ
```

Then use the pushbutton ISR below. This routine checks which KEY has been pressed and writes its corresponding index to the HEX0 display:

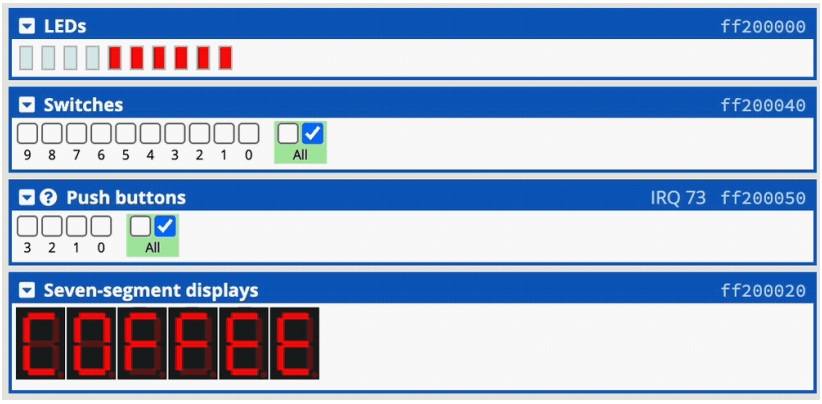
```
KEY_ISR:
    LDR R0, =0xFF200050    // base address of pushbutton KEY port
    LDR R1, [R0, #0xC]     // read edge capture register
    MOV R2, #0xF
    STR R2, [R0, #0xC]     // clear the interrupt
    LDR R0, =0xFF200020    // base address of HEX display
CHECK_KEY0:
    MOV R3, #0x1
    ANDS R3, R3, R1        // check for KEY0
    BEQ CHECK_KEY1
    MOV R2, #0b00111111
    STR R2, [R0]           // display "0"
    B END_KEY_ISR
CHECK_KEY1:
    MOV R3, #0x2
    ANDS R3, R3, R1        // check for KEY1
    BEQ CHECK_KEY2
    MOV R2, #0b00000110
    STR R2, [R0]           // display "1"
    B END_KEY_ISR
CHECK_KEY2:
    MOV R3, #0x4
    ANDS R3, R3, R1        // check for KEY2
    BEQ IS_KEY3
    MOV R2, #0b01011011
    STR R2, [R0]           // display "2"
    B END_KEY_ISR
IS_KEY3:
    MOV R2, #0b01001111
    STR R2, [R0]           // display "3"
END_KEY_ISR:
    BX LR
```

Part 2 Deliverable: *Interactive display using interrupts*

In this task you will revise your Part 1 interactive display to use timers and interrupts: the message will rotate automatically, and change direction, or speed, in response to push-button press-and-release.

Specifications

- By default, C0FFEE flows to the left, shifting one position every 0.25 seconds of simulated time, and wrapping around from left to right. *Reminder: the emulator does not guarantee real-time performance, and emulating 0.25 s may take more than 0.25 s.*
- Like in Part 1, slider switches change the characters in the message. In addition to the Part 1 messages, include the following. Note that for messages longer than six characters, six characters of the message must always be displayed.
    - 0x08: 70Ad570015
    - 0x10: CAFE bEEF C0FFEE
- Push-buttons modify the direction and rate of flow of characters:
    - PB3 pauses and resumes character movement,
    - PB2 reverses direction of movement,
    - PB1 makes movement faster, and
    - PB0 makes movement slower.
- Implement five rates, with one the message moving one position every {1/16, 1/8, 1/4, 1/2, 1} s. Light the two least significant LEDs for the lowest speed; add two more lit LEDs for each increase.
- LEDs display the speed of movement relative to min and max; when paused, all LEDs should be off. When at max speed, all LEDs should be on.
- While paused, PB0-2 have no effect; the timer interrupt has no effect. Upon resume, the system should have the same state as when it was paused.
- When at minimum speed, PB0 has no effect.
- When at maximum speed, PB1 has no effect.



Configuring Interrupts and Timer

Enable interrupts for the ARM A9 private timer (**ID: 29**) used to count time. Also enable interrupts for the pushbuttons (**ID: 73**), and determine which key was pressed when a

pushbutton interrupt is received. Polling will still need to be used to check switch states; interrupts are not supported for this peripheral.

You need to modify some parts of the given template to perform this task:

- **_start**: activate the interrupts for pushbuttons and the ARM A9 private timer by calling the subroutines you wrote in the previous tasks. (Call the *enable_PB_INT_ASM* and *ARM_TIM_config_ASM* subroutines.)
- **IDLE**: You will implement your program logic here.
- **SERVICE_IRQ**: Modify this part so that the IRQ handler checks both the ARM A9 private timer and pushbutton interrupts and calls the corresponding interrupt service routines (ISR). Hint: The given template only checks the pushbutton interrupt and calls its ISR (KEY_ISR). Use labels **KEY_ISR** and **ARM_TIM_ISR** for pushbuttons and ARM A9 private timer interrupt service routines, respectively.
- **CONFIG_GIC**: The given CONFIG_GIC subroutine only configures the pushbutton interrupts. You must modify this subroutine to configure the ARM A9 private timer and pushbutton interrupts by passing the required interrupt IDs.
- **KEY_ISR**: The given pushbutton interrupt service routine (KEY_ISR) performs unnecessary functions that are not required for this task. You must modify this part to only perform the following functions: 1- write the content of the pushbutton edgecapture register to the **PB_int_flag** memory and 2- clear the interrupts. In your main code (see **IDLE**), you may read the **PB_int_flag** memory to determine which pushbutton was pressed. Place the following code at the top of your program to designate the memory location:

```
PB_int_flag:
    .word 0x0
```

- **ARM_TIM_ISR**: You must write this subroutine from the scratch and add it to your code. The subroutine writes the value `'1'` into the **tim_int_flag** memory when an interrupt is received. Then it clears the interrupt. In your main code (see **IDLE**), you may read the **tim_int_flag** memory to determine whether the timer interrupt has occurred. Use the following code to designate the memory location:

```
tim_int_flag:
    .word 0x0
```

Make sure you have read and understood the user manual before attempting this task. For instance, you may need to refer to the user manual to understand how to clear the interrupts for different interfaces (i.e., ARM A9 private timer and pushbuttons)

Performance Analysis

When programming with interrupts, it is often important to determine how much time is spent servicing interrupts versus running user code. Use breakpoints to measure what fraction of time is spent: a) servicing interrupts; b) polling slider switches; c) in IDLE or user code (excluding polling of slider switches). Submit a one (1) page report (10 pt font, 1” margins) to Crowdmark detailing both your measurement approach as well as your results.

Summary

1. Implement an assembly program that uses interrupts to display character sequences as described above (**`part2.s`**).
2. Write a performance analysis report on your implementation.

Deliverables

Your code will be graded automatically to test the functionality of your I/O library and applications built with it. To ensure you receive full marks, define `.global _start` and `start:`, and implement your library functions using the naming specified above.

Your report is limited to one page for the performance analyses related to Part 2. Use no smaller than 10 pt font, no narrower than 1” margins, submit no cover page. The report will be graded by assessing your report’s clarity, organization, and technical content. Report grades will be modulated based on the correct operation of the submitted software. (You cannot receive full marks on a report about broken code.)

Grading

Source code

- 50% Part 1: Interactive display using polling
- 30% Part 2: Interactive display using interrupts

Note that multiple tests will be used to evaluate each submitted deliverable; the value of each test case is weighted equally.

Report

- 20% Performance analysis report (Part 2)

Each part of the report will be graded for: (a) clarity, (b) organization, and (c) technical content:

- 1pt *clarity*: grammar, syntax, word choice
- 1pt *organization*: clear narrative flow from brief overview to results, analysis, conclusions, etc.
- 3pt *technical content*: appropriate use of terms, description of results, analysis, etc.

Submission

Please submit your source code to MyCourses in a single .zip archive, using the following file naming conventions:

- Archive: `StudentID_FullName_Lab3_src.zip`
- Code: `part1.s`, `part2.s`

Please submit your reports to Crowdmark as a PDF, using the following file naming convention:

- `StudentID_FullName_Lab3_report.pdf`