Lab 2

This lab introduces the basic I/O capabilities of the DE1-SoC computer, more specifically, the slider switches, pushbuttons, LEDs, 7-Segment (HEX) displays and timers. After writing assembly drivers that interface with the I/O components, timers and interrupts are used to demonstrate polling and interrupt based applications.

Part 1- Basic I/O

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 I/O components is fairly simple to understand. The ARM cores have 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 of the basic I/Os, 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 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 a slightly more sophisticated FPGA hardware. 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 by a simple edge detection circuit in the FPGA. This section will deal with writing assembly code to control the I/O components by reading from and writing to the memory.

Getting Started: Drivers for slider switches and LEDs

To access the memories designated to the I/O interfaces you need **drivers**. In other words, you need to write subroutines (derivers) in order to write to or read from the I/O interface memories. Therefore, you must follow the conventions you have learned in this course when describing your drivers in assembly language.

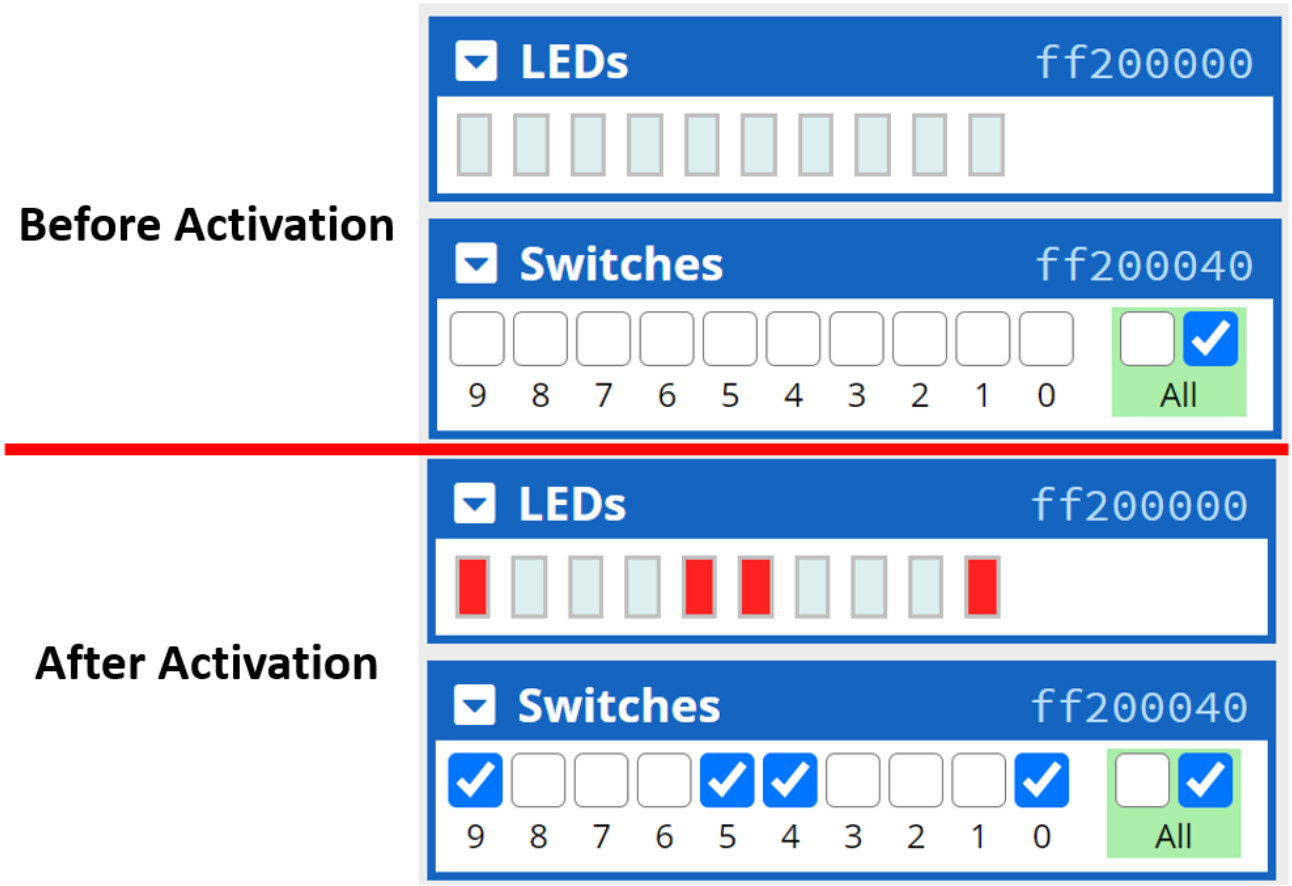
1. **Slider Switches:** Create a new subroutine labelled **read_slider_switches_ASM**, which reads the value from the memory location designated for the slider switches data (SW_MEMORY) and stores it into the R0 register, and then branches to the address contained in the link register (`LR`). Remember to use the subroutine calling convention, and save the context (Registers) if needed!
2. **LEDs:** Create a new subroutine labelled **write_LEDs_ASM**. The write_LEDs_ASM subroutine writes the value in `R0` to the LEDs memory location (LED_MEMORY), and then branches to the address contained in the `LR` .

To help you get started, the codes for the slider switches and LEDs drivers have been provided below. Use them as templates for writing future drivers.

```
// Sider Switches Driver
// returns the state of slider switches in R0
.equ SW_MEMORY, 0xFF200040
/* The EQU directive gives a symbolic name to a numeric constant,
a register-relative value or a PC-relative value. */
read_slider_switches_ASM:
    LDR R1, =SW_MEMORY
    LDR R0, [R1]
    BX  LR
```

```
// LEDs Driver
// writes the state of LEDs (On/Off state) in R0 to the LEDs memory location
.equ LED_MEMORY, 0xFF200000
write_LEDs_ASM:
    LDR R1, =LED_MEMORY
    STR R0, [R1]
    BX  LR
```

Your objective for this part of the Lab is to use the read_slider_switches_ASM and the write_LEDs_ASM subroutines to turn on/off the LEDs. To do so, write an endless loop. In the loop, call the the read_slider_switches_ASM and the write_LEDs_ASM subroutines in order. Compile and Run (Continue) 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. 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., HEX displays and push-buttons drivers.

1- HEX displays: There are 6 HEX displays (HEX0 to HEX5) on the DE1-SoC Computer board. You are required to write three subroutines to implement the functions listed below to control the HEX displays:

- HEX_clear_ASM: The subroutine will turn off all the segments of the HEX displays passed in the argument. It receives the HEX displays indices through `R0` register as an argument.
- HEX_flood_ASM: The subroutine will turn on all the segments of the HEX displays passed in the argument. It receives the HEX displays indices through `R0` register as an argument.
- HEX_write_ASM: The subroutine receives the HEX displays indices and an integer value between 0-15 through `R0` and `R1` registers as an arguments, respectively. Based on the second argument value (`R1`), 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 argument for all the displays HEX0-HEX5, and write to whichever ones have been asserted. A loop may be useful here! The HEX displays indices can be encoded based on a one-hot encoding scheme:

```
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:

```
mov R0, #0x0000000C
BL  HEX_flood_ASM
```

2- Pushbuttons: There are 4 Pushbuttons (PB0 to PB3) on the DE1-SoC Computer board. You are required to write seven subroutines to implement the functions listed below to control the pushbuttons:

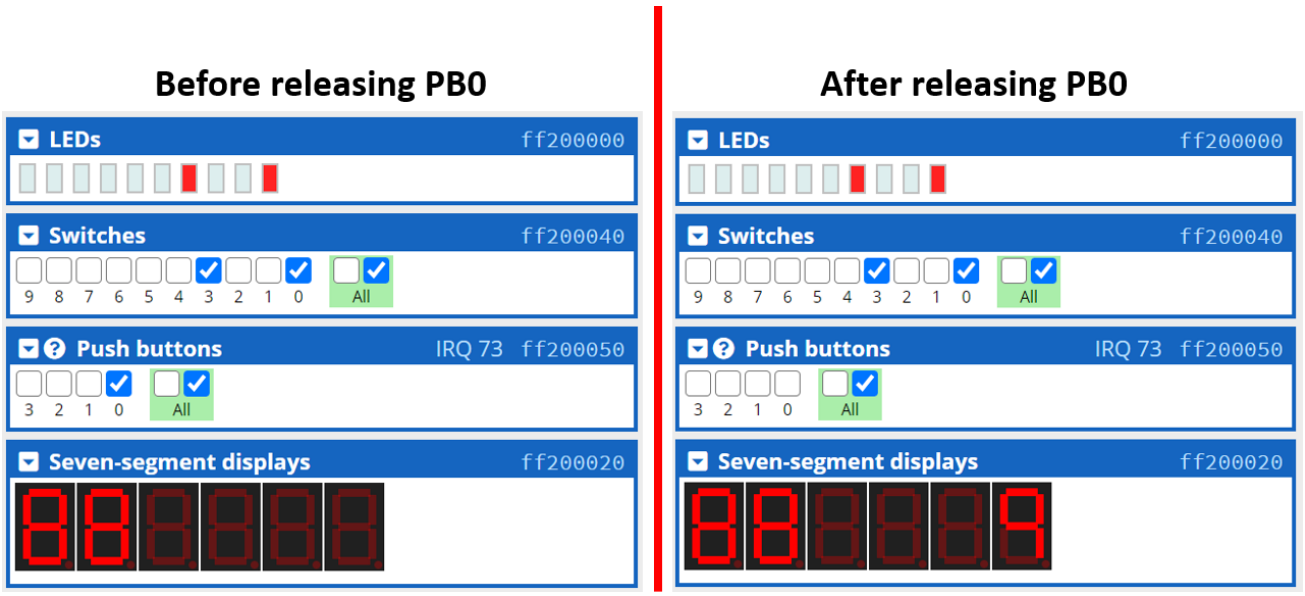
- read_PB_data_ASM: The subroutine returns the indices of the pressed pushbuttons (the keys form the pushbuttons Data register). The indices are encoded based on a one-hot encoding scheme:

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

- PB_data_is_pressed_ASM: The subroutine receives pushbuttons indices as an argument (One index at a time). Then, it returns `0x00000001` when the the corresponding pushbutton **is** pressed.
- read_PB_edgecp_ASM: The subroutine returns the indices of the pushbuttons that have been pressed and then released (the edge bits form the pushbuttons Edgecapture register).
- PB_edgecp_is_pressed_ASM: The subroutine receives pushbuttons indices as an argument (One index at a time). Then, it returns `0x00000001` when the the corresponding pushbutton **has been** asserted.
- PB_clear_edgecp_ASM: The subroutine clears the pushbuttons 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: The subroutine receives pushbuttons 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: The subroutine receives pushbuttons indices as an argument. Then, it disables the interrupt function for the corresponding pushbuttons by setting the interrupt mask bits to `'0'`.

Write an application that uses the appropriate drivers (subroutines) created so far to perform the following functions. As before, the state of the slider switches will be mapped directly to the LEDs. Additionally, the state of the last four slider switches SW3-SW0 (SW3 corresponds to the most significant bit) will be used to set the value of a number from 0-15. This number will be displayed on a HEX display when the corresponding pushbutton is pressed. For example, pressing PB0 will result in the number being displayed on HEX0. When the pushbutton is released, the value displayed on the HEX display should remain unchanged, even if you change the state of the slider switches. Since there are no pushbuttons to correspond to HEX4 and HEX5, you must turn on all the segments of the HEX4 and HEX5 displays. Finally, asserting slider switch SW9 should clear all the HEX displays. Figure below demonstrates the result of activating slider switches 0 and 3 and pressing pushbutton 0 (PB0). Remember, you have to release the pushbuttons to see the results as the Edgecapture register is updated once the pushbuttons are released (unchecked).



Part 2- 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.

Brief overview

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 (by using a phase-locked loop). 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 Computer 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: ARM A9 private timer is a down counter and requires initial count value. Use `R0` to pass this argument.

2- Configuration bits: Use `R1` 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.

You are required to write three subroutines to implement the functions listed below to control the timers:

- ARM_TIM_config_ASM: The subroutine is used to configure the timer. Use the arguments discussed above to configure the timer.
- ARM_TIM_read_INT_ASM: The subroutine returns the “F” value (`0x00000000` or `0x00000001`) from the ARM A9 private timer Interrupt status register.
- ARM_TIM_clear_INT_ASM: The 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` into the Interrupt status register.

To test the functionality of your subroutines, write an assembly code 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 1 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.

Creating an application: Polling based Stopwatch!

Create a simple stopwatch using the ARM A9 private timer, pushbuttons, and HEX displays. The stopwatch should be able to count in increments of 10 milliseconds. Use the ARM A9 private timer to count time. Display milliseconds on HEX1-0, seconds on HEX3-2, and minutes on HEX5-4.

PB0, PB1, and PB2 will be used to start, stop and reset the stopwatch, respectively. Use an endless loop to poll the pushbutton edgecapture register and the “F” bit from the ARM A9 private timer interrupt status register.

Part 3- 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 current code execution, 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 processors time and use it only when required.

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 sources of interrupts. When a peripheral device sends its IRQ signal to the GIC, then 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 such interrupts can be used, software code has to perform a number of steps:

1. Ensure that IRQ 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 interrupt requests to the GIC.
4. Enable IRQ interrupts in the A9 processor, by setting the IRQ disable bit in the CPSR to 0.

An example assembly language program is given below. This program demonstrates use of interrupts with assembly language code. 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 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
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 processor's modes have their own stack pointers and link registers (seed fig 3 in "Using the ARM Generic Interrupt Controller" document). As 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 the 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 shows how to set the stack to the A9 on-chip memory in IRQ mode.

```
.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 onchip 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
    // To DO: 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 objective task
```

Then, you need to define the exception service routines 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

/* To Do: 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 you are required to add the following 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)); */
/* To Do: 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 use the pushbutton Interrupt Service Routine (ISR) given below. This routine checks which KEY has been pressed and writes 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    // based 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
```

Interrupt based stopwatch!

Before attempting this section, get familiarized with the relevant documentation sections provided in the introduction.

Modify the stopwatch application from the previous section to use interrupts. In particular, enable interrupts for the ARM A9 private timer (**ID: 29**) used to count time for the stopwatch. Also enable interrupts for the pushbuttons (**ID: 73**), and determine which key was pressed when a pushbutton interrupt is received.

In summary, you need to modify some parts of the given template to perform this task:

- **_start**: activate the interrupts for pushbuttons and ARM A9 private timer by calling the subroutines you wrote in the previous tasks (Call enable_PB_INT_ASM and ARM_TIM_config_ASM subroutines)
- **IDLE**: You will describe the stopwatch function here.

- **SERVICE_IRQ**: modify this part so that the IRQ handler checks both ARM A9 private timer and pushbuttons interrupts and calls the corresponding interrupt service routine (ISR). Hint: The given template only checks the pushbuttons 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 pushbuttons interrupt. You must modify this subroutine to configure the ARM A9 private timer and pushbuttons interrupts by passing the required interrupt IDs.
- **KEY_ISR**: The given pushbuttons 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 pushbuttons edgecapture register in 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'` in to 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)

Grading and Report

Your grade will be evaluated through the deliverables of your work during the demo (**70%**) (basically showing us the working programs), your answers to the questions raised by the TA's during the demo (**10%**), and your lab report (**20%**).

Grade distribution of the demo:

- Part 1.1: Slider switches and LEDs program (**5%**).
- Part 1.2: HEX displays and pushbuttons (**5%**).

- Part 2.1: Counters based on ARM A9 private timers (**15%**).
- Part 2.2: Polling based stopwatch (**25%**).
- Part 3: Interrupt based stopwatch (**20%**).

Write up a short report (max 5 pages in total) that should include the following information.

- A brief description of each part completed (do not include the entire code in the body of the report).
- The approach taken (e.g., using subroutines, stack, etc.).
- The challenges faced, if any, and your solutions.
- Possible improvement to the programs.

Your final submission should be submitted on myCourses. The deadline for the submission and the report is **Friday, 6 November 2020**. A **single compressed folder** should be submitted in the **.zip** format, that contains the following files:

- Your lab report in **pdf** format: **StudentID_FullName_Lab2_report.pdf**
- The assembly program for Part 1.1: **part1_1.s**
- The assembly program for Part 1.2: **part1_2.s**
- The assembly program for Part 2.1: **part2_1.s**
- The assembly program for Part 2.2: **part2_2.s**
- The assembly program for Part 3: **part3.s**

> **Important**
>
> Note that we will check every submission (code and report) for possible plagiarism. All suspected cases will be reported to the faculty. Please make sure to familiarize yourself with the course policies regarding Academic Integrity and remember that all the labs **are to be done individually**.

The **demo** will take place via Zoom between **2-6 Nov 2020** on the day of you assigned lab session day. We will provide a registration system for the demo the week before. You will need to answer live questions during the demo with your screen shared to onstrate the working program.