Lab2 report

Wen Cui 260824815

Within this lab, the interaction with basic I/Os are implemented. More specifically, the ARMv7 simulator website was used to implement assembly drivers that interface with the I/O components (slider switches, pushbuttons, LEDs, 7-segment (HEX) displays), timers and interrupt. In part2 and part3 of this lab, I/O components, timers, and interrupts were used to demonstrate polling and interrupt-based applications.

Part1 – Basic I/O

Part 1-1 Drivers for sliders switches and LEDS

Description: In order to turn on and off LEDs using switches, 2 basic subroutines were used. They were provided in the lab instruction. More specifically, read_slider_switches_ASM is the slider switches driver subroutine which returns the state of the slider switches in R0; write_LEDs_ASM is the LED driver which writes the value of r0, which is the value of slider switches, to LED_MEMORY address (r1) using the store instruction. In order to constantly reading the value, I used an endless loop. In the loop, these 2 subroutines were called in order, namely first read then write then read again.

Approach: The main approach taken in the part is the use of device memory location to read and write to specific I/O device through ldr and str instruction. The other approach taken was the use of an endless loop to continuously read and write switch values to LEDs. This can be implemented by looping and unconditional branching.

Challenges faced: Since the template was given in the lab instruction, it was relatively easy to implement this part of the lab. The main challenge was to understand the template given to get a better understanding of implementing I/O device driver. Other than that, no main challenge has encountered and it took me relative short time compared to the other parts of this lab.

Possible improvements: Since the basic code are provided in the instruction, there were relatively limited ways to implement this part. Therefore, I did not come up with possible improvements for this part.

Part 1-2 Drivers for HEX displays and pushbuttons

Description: There are several steps to implement this part

1. HEX display subroutines. 3 subroutines are implemented to perform clear, flood, write to HEX displays.

- HEX_clear_ASM is used to turn off corresponding displays provided in r0. First, the 2 base addresses for HEX0_3 and HEX4_5 are stored in different registers since they have different base address. Then, since the display indices are hot encoded, tst instruction on r0 and display indices can be used to perform bitwise and operation on each bit in order to determine that the display to clear. Then if the desired indices are 4/5, we need to change the base address from HEX0_3 (0XFF200020) to HEX4_5 (0XFF200030). After that, 0 are written to corresponding indices of data registers of HEX display by using bitwise and (to clear the bits) and store instruction (to store the 0 bits in corresponding position of data register in order to display it).
- HEX_flood_ASM uses similar approach with the clear subroutine. The only difference is that it is used to turn on HEX displays with the corresponding indices passed in r0 instead of clearing it them. First, the 2 base addresses for display are stored in registers. Then, the same approach by tst instruction used to determine display indices with r0 are used. We also need to change the base address if the desired indices are 4/5. After that, F are written to corresponding indices of data registers of HEX display by using bitwise orr (to flood the bits) and store instruction (to store the F bits in corresponding position of data register in order to display it).
- HEX_write_ASM performs write value provided in r0 to display indices provided in r1. The first part is to determine display indices as before. Then, another register (r6) was used to store the HEX value for the number we want to display, it was determined by looking up the manual. After that, depending on the display indices, corresponding and instruction are performed to only display the number on the desired indices and store instruction was also used to display the number.

2. Push button subroutines. 7 subroutines shown in the following are implemented to meet the requirements and provide a better understanding of how push button works.

- Read_PB_data_ASM: this subroutine returns the indices of pressed push buttons. To achieve this, we simply need to ldr the data, which is the value of push buttons to register r0 from the base address (0xFF200050).

- PB_data_is_pressed_ASM: this subroutine returns 1 when pushbutton indices passed is pressed. To achieve this, I used ldr instruction to read the PB data and tst instruction to compare it with the indices passed. If the result is larger than 0, 1 is stored in r0.
- Read_PB_edgecp_ASM: this subroutine returns the corresponding indices when push button indices passed are released. When a push button is released, the edge capture register is updated with the released indices. Therefore, this subroutine can be achieved by simply loading the value from edge capture register's base address (0xFF20005C)
- PB_edgecp_is_pressed_ASM: this subroutine returns 1 when the push button indices passed in has been asserted. This can be achieved by loading the value of edge capture register then compare it with indices passed using tst instruction. If the result is not equal to 0, store 1 into r0.
- PB_clear_edgecp_ASM: this subroutine clears edge capture register. This can be achieved by writing a 1 to the edge capture register's base address.
- Enable_PB_INT_ASM: this subroutine enables the interrupt function for push button indices passed as arguments. This can be achieved by storing 3 (011) into the interrupt mask register using str instruction on the Interrrupt mask register's base address (0xFF200058)
- Disable_PB_INT_ASM: this subroutine disables the interrupt function for push button indices passed as arguments. This can be achieved by storing 0 into interrupt mask register

Implement the desired functions described in the instruction using previous subroutine. This can be achieved by combing the subroutines described before and integrate them into a endless loop.

- Before starting the loop, all registers are cleared and initialized. Then, we clear HEX0-3 since it needs to be cleared regardless the slider switch value. This can be achieved by passing 15 (the sum of indices of HEX0-3) to HEX_clear_ASM.
- Inside the endless loop: At the beginning of the loop, we check if SW9 is asserted, which can be achieved by loading the slider switch's value to r0 by using subroutines in part1-1. If r0 is greater than 512 (SW9 is asserted), we set r0 to the sum of all indices, then branch to HEX_clear_ASM. Else, we pass 30 (the sum of HEX4 and HEX5 indices) to HEX_flood_ASM. Then in order to display the number only when push button is released, "bl PB_edgecap_is_pressed_ASM" is called to check whether to display. In order to simplify the program, I modified the PB_edgecap_is_pressed_ASM so that it stores the push button indices that has just been released into r1 and returns 1 if r1 is not 0. After calling this subroutine, if r0 is equal to 1, which means we are now required to display, the slider switch value is loaded to r0 as before and since the push button indices is already loaded into r1, we have gathered all parameter needed, HEX_write_ASM is called to display. After that, PB_clear_edgecp_ASM is called to reset the interrupt bit so that it will not affect future interrupts.

Main approach:

Subroutine part: the main approach taken in HEX display was the use of tst instruction to determine display indices since the indices are hot encoded. The other approach in HEX subroutines is the use of and/orr instruction to clear/flood/write specific bits inside data register. Combining these 2 approaches, the display subroutines can be easily implemented. For push button subroutines, the main approach is the use of load/store instruction on different base addresses. Once I understood the work inside the push buttons, the subroutines are easy to implement.

Main program part: the first approach taken is same as part1-1, which is the use of endless loop to keep polling from interrupt register in push buttons and slider switch values to perform desired function. Inside the endless loop, the main approach is to make the use of register, mainly r0, to communicate among different subroutines. The return value is always stored in r0. So r0 is used to detect interrupt, and if interrupt is detected, it stores the value of slider switches. Moreover, in this program, r1 is also used to store the push button indices if it has been released.

Main challenge:

Subroutine part: the main challenge I faced when implementing subroutines was understanding how HEX and push button work. For the HEX display, at first, I didn't distinguish between HEX0_3 and HEX4_5. Then after using the debug mode and multiple tests, the bug was detected and fixed. For the push button part, it took me a while to understand the given manual. At first, I didn't realize that the value returned is required to store in r0. Instead I used some other registers, it gave me hard time when implementing the main program since I must poll different registers to check result.

Main program part: the main challenge I encountered was the order of subroutines calls based on the return value. At first, I read the value of slider switches before checking push button edge capture register, which caused the loss of switch values since both subroutines return in r0. After adding multiple breakpoints, the error was detected and corrected.

Possible improvements:

One possible improvement is to use interrupt instead of endless loop for detecting push button releases. This can be implemented using GIC. In this way, an interrupt can be generated as soon as push button is released, so that we do not have to wait until the next loop to detect the interrupt.

Part 2 – Timers

Part2-1 ARM A9 private timer

Description: In this part, the ARM A9 private timer, which is stored in 0xfffec600, is used to measure time.

First, 3 subroutines are implemented that can be used in further implementation and provide a better understanding for the timer.

- ARM_TIM_config_ASM: this subroutine starts the timer.
    1. Load the initial load value in load register by storing it into the base address of timer
    2. Set the control register to the value passed by r1 by storing it into the base address of timer +8 (the address for control register)
- ARM_TIM_read_INT_ASM: this subroutine returns 1 when interrupt is asserted in the interrupt status register (the current value comes down to 0). This can be achieved by reading the interrupt bit (base address+12). Then it stores 1 in r0 if the interrupt bit is 1.
- ARM_TIM_clear_INT_ASM: this subroutine clears the F value in timer's interrupt status register. This can be achieved by writing 1 in it (base address+12) using store instruction.

After implementing the 3 subroutines, counter for 1 sec can be realized by using these subroutines.

First, I used ARM_TIM_config_ASM to configure the timer. In order to do this, 7 is passed into r1 to set enable bit, interrupt bit and auto bit to 1. The initial value is loaded to 200000000 since the frequency of timer is 200Mhz so that we can get an interrupt generated every 1 sec. Then an endless loop is used to poll the interrupt status register using ARM_TIM_read_INT_ASM. If the interrupt is generated (r0==1), which means 1 sec has passed, the program branches to subroutine "count". Inside the subroutine, we can reuse the HEX_write_ASM in the previous part by passing 1 to r1 (display on HEX0) and count to r0. Then, increment the counter, clear the interrupt using "ARM_TIM_clear_INT_ASM", wait for the next interrupt to be generated.

Main approach:

Subroutine part: the main approach is simply making the use of load and store instruction on different base addresses (read from the manual) to configure and detect interrupt of the counter.

Main program part: the main approach is to make the use of interrupt bit to measure time. Also, the endless loop is used to detect an interrupt. I also used the return value (r0) to communicate among different subroutines to ensure writing operation is only performed when an interrupt was generated.

Main challenge: At first, I forgot to clear the interrupt bit after each time an interrupt is generated, which gave a false performance on the counter. After adding multiple breakpoints, the error was fixed. Other than that, the program was relatively easy.

Possible Improvement: One possible improvement is that the branch "count" where I used to update display can be omitted and integrated into the main loop. More specifically, if we detect an interrupt, the loop body will be executed; else, the program will branch back to poll interrupt status register. In this way, the complexity for this program might be reduced.

Part2-2 Polling-based stopwatch

Description: In this part, we are required to create a stopwatch which counts every 10 milliseconds using ARM A9 private timer to count time and HEX0-5 to display the time in minutes, seconds, milliseconds. The following steps are performed to implement this.

1. Display time: First, the timer is configured, by setting the initial load value to 2000000 to increment the counter every 10 milliseconds. Several registers are initialized to store each bit for HEX display. R6 is initialized to store the total count value. Each time an interrupt is generated, r6 is incremented by 1 using the same strategy as part2-1. The main difference is how to display. To achieve this, I implemented a loop. More specifically, every time the count reaches 10, I increment the registers that store value for HEX1 by1, subtract the count by 10, then branch back to the loop until the count is less than 10 so that it can be displayed on HEX0. Similarly, every time the count reaches 100, update the registers that store value for

HEX2, subtract the count by 100 then branch back to the loop. The other registers can be updated similarly to store the right value for corresponding display. To ensure we check MSB first, they are called in reverse order. After updating registers, HEX_write_ASM will be called to write register values to their corresponding display by setting r0 and r1 repeatedly.

2. Push button interrupt: The next part for this program is to enable start/stop/restart for the stopwatch. This can be achieved by introducing push button interrupts from part1.
3. The application can then be achieved by combing the previous implementations using the endless loop to constantly check the edge capture register. Inside the loop, the push button edge capture register is polled, and the "switch" branch is called to perform different functionalities depends on types of interrupt. More specifically, if the value is 1 the subroutine for configurating timer is called; when the edge capture register is 2, the enable bit in control register of the timer is cleared to 0 so that the timer is stopped. Finally, when the edge capture register is 4, the restart function can be achieved by first stopped the timer, write all HEX display to 0, clear the registers that were used to store value for each register, then move the total count value to 0, reconfigure the timer. After that, we branched back to the main loop and display the time as described above.

Main approach: The main approach used in this part displaying the time by implementing a loop and keep decrementing the count until we can the value to be written to each HEX display. The other main approach is the use of endless loop to constantly check push button interrupts and timer interrupt bit to perform desired stopwatch functions and get a measurement of time. Additionally, to realize the start/stop/restart functions for the stopwatch, I took advantage of the configuration of the timer, made use of the load register, enable bit to perform desired functionality.

Main challenge: the main challenge I faced was how to display the value on corresponding displays given the total count value. After multiple tries, I figured out one solution using looping and decrementing technique as described above. Moreover, when implementing the restart function, at first, I forgot to clear the registers that were used to store value for each register, which caused the timer to keep counting instead of restarting. After adding multiple breakpoints, the problem was solved finally.

Possible improvement: there are several possible improvements for this part

1. Instead of using polling, a better approach using GIC can be implemented to give a better performance, which will be implemented in the next section for this lab
2. Instead of using the looping & decrementing method to display the value, a simpler way is to using mod 10 operation and division operation. In this way, we can get each digit easily and get rid of looping. However, I did not find the mod 10 operation or division operation in ARMv7. I will try to implement them in the next lab.

Part 3 Interrupt

Description: The goal is to use GIC to implement the same stopwatch as previous part. By modifying the code provided in the instruction, the stopwatch function can be implemented more efficiently using the IRQ mode of GIC. The following shows the details of implementation

- _start: enable_PB_INT_ASM and ARM_TIM_config_ASM subroutines implemented before are called with bl instruction.
- SERVICE_IRQ: this can be modified by adding another subroutine, timer_check, inside which we compare the timer ID 29 with the ICCIAR. If it is equal, which means an interrupt is generated for the timer, we branch to ARM_TIM_ISR.
- CONFIG_GIC: in order to configure interrupt for timer, we can simply repeat the lines provided for key interrupt and replace the id with 29.
- KEY_ISR: 1. write the content of edge capture register into PB_int_flag. This can be achieved by store/load instruction on base address of edge capture register. 2. clear the interrupt. This can be achieved by writing "1" in the base address of edge capture register.
- ARM_TIM_ISR: first, it checks the interrupt status register by load instruction. Then, if it is equal to 1, tim_int_flag is changed and 1 is stored in interrupt status register.
- beginning this starts the main program that implements stopwatch.

The main structure is like the previous implementation. The main difference is that instead of keep polling the registers and checking return values, two global variables are used to store the interrupt bits. By KEY_ISR and ARM_TIM_ISR, these bits will be updated automatically when a corresponding interrupt is raised.

1. the beginning subroutine is used to start the program. More specifically, program is started as soon as start/restart signal is generated, else it will branch back to itself.

2. Inside the prog subroutine, the stop signal can be detected by reading PB_int_flag. If the value is equal to 1, the count subroutine will simply be called. If the value is equal to 2, it will branch back to the beginning of prog and the rest of loop will stay unexecuted. If the value is equal to 4, we first clear all displays by HEX_write_ARM, then set the value of registers that store in corresponding displays to 0, then the count subroutine can be called to start the counter.
3. The count subroutine is the same as the previous part. The only difference is that instead of polling timer interrupt, we read the tim_int_flag to detect interrupt and update counter, write to corresponding displays when an interrupt is generated.

Main approach: the main approach used in this part is the GIC interrupt. Instead of polling, GIC can be used to determine the interrupt and take the appropriate action. This can be achieved by configure interrupts for each device, implement ISR subroutines which indicates the action to be take upon interrupt, then the program will be resumed after the ISR subroutine.

Main challenge: Since the main program is like the previous part, the only challenge I faced was understanding how does GIC work. It took me some time to implement the start/stop/restart functions by constantly checking values in global variables using while loop instead of polling. After understanding the internal work for GIC, the remaining parts are relatively easy to implement.

Possible improvements: One possible improvement for this part is that instead of using the looping & decrementing method, use mod 10 operation and division operation to display the corresponding value. In this way, we can get each digit easily and get rid of looping. However, I did not find the mod 10 operation or division operation in ARMv7. I will try to implement them in the next lab. The other possible improvements is to better organize the main loop by combining several subroutines to provide a better readability on the program.