

Lab2 report

Wen Cui 260824815

Within this lab, the interaction with basic I/Os is 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:

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. 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 devices 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 the I/O device driver. Other than that, no main challenge has been encountered, and it took me a relatively short time compared to the other parts of this lab.

Possible improvements:

Since the basic code is 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. Then, since the display indices are not encoded, `TST` instruction on R0 and display indices can be used to perform bitwise AND operation on each bit to determine that the display 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 the corresponding position of data register to display it).
 - `HEX_flood_ASM` uses a similar approach with a 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. 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 was used. We also need to change the base address if the desired indices are 4/5. After that, F is 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 the corresponding position of data register 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 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 the 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 pushbuttons 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 the 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 the 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 the 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 first using ORR R0, R0, R2 where r2 is the value of interrupt mask register followed by "str r0, [r1]" where r1 is the base address of Interrupt mask register (0xFF200058).
 - Disable_PB_INT_ASM: this subroutine disables the interrupt function for push-button indices passed as arguments. This can be achieved by first using BIC R0, R0, R2 where r2 is the value of interrupt mask register followed by "str r0, [r1]" where r1 is the base address of Interrupt mask register (0xFF200058).

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

- Before starting the loop, all registers are cleared and initialized. Then, we clear HEX0-3 since it needs to be cleared regardless of 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 to display the number only when the push button is released, "bl PB_edgecap_is_pressed_ASM" is called to check whether to display. To simplify the program, I modified the PB_edgecap_is_pressed_ASM so that it stores the push button indices that have 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 are 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 the 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, also "ORR/BIC" instruction is also used to determine to recognize specific push buttons passed as parameter. Once I understood the work inside the pushbuttons, the subroutines are easy to implement.

Main program part: the first approach taken is the same as part1-1, which is the use of an endless loop to keep polling from interrupt register in pushbuttons and slider switch values to perform the 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 an 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 the result.

Main program part: the main challenge I encountered was the order of subroutine calls based on the return value. At first, I read the value of slider switches before checking the 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 an endless loop for detecting push-button releases. This can be implemented using GIC. In this way, an interrupt can be generated as soon as the 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 0xffec600, is used to measure time.

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

- ARM_TIM_config_ASM: this subroutine starts the timer.
 1. Load the initial load value in the load register by storing it into the base address of the 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 the control register)
- ARM_TIM_read_INT_ASM: this subroutine returns 1 when an 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 the 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, a counter for 1 sec can be realized by using these subroutines.

First, I used `ARM_TIM_config_ASM` to configure the timer. 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 the 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 "counter". 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.

Additionally, as indicated in the instruction, when the counter reaches 15 (F), the counter needs to restart. This can be achieved by comparing counter to 15. If it is larger than 15, a loop is implemented to keep subtracting counter value by 16 until the value is lower than or equal to 15. In this way, the counter can be restarted.

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. Moreover, the strategy of restart the counter by keep subtracting using a conditional loop is used so that the counter can be restarted from 0 after it reaches 15.

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. Another issue I faced was to restart the counter after reaching F. At first, I subtracted 15 instead of 16 from the total counter, however, 0 is always omitted in this case. After adding breakpoints, the error was detected and corrected. Other than that, the program was relatively easy.

Possible Improvement:

One possible improvement is to use a modulo 16 operation by shifting operation to keep the counter between 0-15 instead of keep subtracting using the loop. In this way, the efficiency of the program can be increased by getting rid of keep subtracting and branching.

Part2-2 Polling-based stopwatch

Description: In this part, we are required to create a stopwatch that 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: This is implemented inside the "count" subroutine. 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 the 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 the corresponding display. Other than that, `HEX_write_ASM` is also called inside "count" to write register values to their corresponding display by setting r0 and r1 repeatedly. Most importantly, all these can be done only if an interrupt from the timer is generated, else it will branch back to the head and stop executing the write and increment operations.
2. Push-button interrupt: The next part for this program is to enable start/stop/reset for the stopwatch. This can be achieved by introducing push-button interrupts from part1.

3. The application can then be achieved by combining 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 first, and then the value of it, which is stored in r1 is compared with 1, 2, 4 to perform different functionalities depends on the types of interrupt. More specifically, if the value is 1, the counter is started by calling the configure timer subroutine with r1=3 (enable timer); when the edge capture register is 2, the enable bit in the control register of the timer is cleared to 0 so that the timer is stopped, this is implemented by `disable_timer` branch. Finally, when the edge capture register is 4, the reset function can be achieved by first stopped the timer (`disable_timer` branch), 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. we branched back to the main loop and display the time as described above.
4. After checking the push button indices, the count subroutine is called inside which the register values are written on display, and counter value is incremented when a timer interrupt is generated as described in step1.

Main approach:

The main approach used in this part is 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 an 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 the 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 the looping and decrementing technique as described above. Moreover, at first, I forget to set r1=1 back after setting it to 3 when starting the stopwatch, since r1 is passed as a parameter to successfully configure the timer. In this way, the timer was never started since it is disabled right after when it is compared with 2 to detect pushbutton 1. However, after adding multiple breakpoints, the bug was detected and solved in the end.

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 future.

Part 3 Interrupt

Description: The goal is to use GIC to implement the same stopwatch as the 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 the 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`: to configure interrupt for the timer, we can simply repeat the lines provided for key interrupt and replace the id with 29.

- KEY_ISR: 1. write the content of the edge capture register into PB_int_flag. This can be achieved by store/load instruction on the base address of the edge capture register. 2. clear the interrupt. This can be achieved by writing "1" in the base address of the edge capture register.
- ARM_TIM_ISR: first, it checks the interrupt status register by the load instruction. Then, if it is equal to 1, tim_int_flag is changed and 1 is stored in the 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 keeping 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 by detecting push-button release. More specifically, the program is started as soon as the start signal is generated, else it will branch back to itself. The stop/reset/start signal can be detected by reading PB_int_flag. If the value is equal to 1, the timer will be configured and then the count subroutine will be called. If the value is equal to 2, it will branch back to the beginning of the loop and the rest of the 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, disable the timer via passing 6 (set enable bit to 0) to configuration subroutine, after that, it will branch back to the beginning of the loop and the rest of loop will stay unexecuted.
2. The count subroutine is the same as the previous part. The only difference is that instead of the 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 configuring interrupts for each device, implement ISR subroutines which indicates the action to be taken 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 works. It took me some time to implement the start/stop/reset 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 future. The other possible improvements are to better organize the main loop by combining several subroutines to provide better readability on the program.