ECSE 324 Lab Report 2
Chenyi Xu
260948311

Task1:

Three subroutines for Hex displays, seven subroutines for pushbuttons, one subroutine for switch, and one subroutine for LED were required to implement the deliverable for Task 1. For task 1, the switches will be mapped directly to LEDs. In addition, SW0 to SW3 will be used to set the integer value for the hex displays. Pushbutton will be used to determine which hex display we will use. When the pushbutton is released, the number on the hex display should remain unchanged. All segments of HEX4 and HEX5 displays should be turned on. Subroutines for switch and LED were given by the instructor which are read_slider_switches_ASM and for write_LEDs_ASM.

Three subroutines required for HEX displays are HEX_clear_ASM, HEX_flood_ASM, and HEX_write_ASM. HEX_clear_ASM subroutine will turn off all the segments of the HEX displays passed in the argument. R0 was used as my input argument for this subroutine. The TST instruction was used to compare the value of input argument and HEX displays. If it does not equal to zero, the bits on the seven-segment display will be masked to zero by using an AND instruction as shown in *Figure 1*. Always remember to follow the callee convention to push and pop the registers used in the subroutine. HEX_flood_ASM subroutine will turn on all the segment of the HEX displays passed in the argument. R0 was used as my input argument. The main idea is similar to HEX_clear_ASM subroutine, the only difference here is I used ORR instruction to mask the desired bits to 1 in order to turn on the hex display. HEX_write_ASM receives integer input argument through R0 and hex display input argument through R1. First, I used CMP instruction to test out the integer input. Then, I moved the matching bit on the hex display to a register. In addition, applying the same idea as shown in Figure 1, I used TST to find the correct hex display. Using the AND instruction to clear the hex display first. Then used the ORR instruction to mask the seven-segment display with the bits saved in the register,

Seven subroutines required for pushbuttons are read_PB_data_ASM, PB_data_is_pressed_ASM, read_PB_edgecp_ASM, PB_edgecp_is_pressed_ASM, PB_clear_edgecp_ASM, enable_PB_INT_ASM, and disable_PB_INT_ASM. Read_PB_data_ASM has one-hot encoding based indices. Load the address of PB_DATA which is 0xFF200050 based on manual into register. Then load the value on this register into the output register R0. PB_data_is_pressed_ASM simply just returned a one wherever the corresponding pushbutton is pressed. Read_PB_edgecp_ASM is also one-hot encoding. Load the address of PB_EDGE which is 0xFF20005C into register. Then load the value on this register into the output register R0. PB_edgecp_is_pressed_ASM returns 1 whenever the corresponding pushbutton is pressed. PB_clear_edgecp_ASM clears the Edgecapture register by write what we just read back onto the register. Enable_PB_INT_ASM store the input indices R0 into the address of PB_MASK which is 0xFF200058. Disable_PB_INT_ASM is used to set the interrupt mask bit to 0. The BIC instruction was used here to achieve this function.

Task 1 deliverable was achieved by an infinite loop and aforementioned subroutines. The output from read_slider_switches_ASM was used as the input for write_LEDs_ASM. It was also used by a TST instruction to see if switch 9 was on which would require us to call the HEX_clear_ASM subroutine. HEX_flood_ASM was used to turn on display for HEX4 and HEX5. Read_PB_edgecp_ASM and PB_clear_edgecp_ASM were used to control the pushbuttons. Output R0 was moved to R1. The output from SW0 to SW3 was moved to R0. Those two registers were then used as input arguments for HEX_write_ASM. The produced output is the same with what is required.

More loops can be used instead of hard code everything. This way the code efficiency can be significantly improved. More time can also be saved as fewer instructions are executed.

```
hex_0_c:
  MOV R1, #0x00000001
  TST R0, R1
  BEQ hex_1_c

  LDR R2, =ADDR_7SEG1
  LDR R3, [R2]
  MOV R4, #0xFFFFFF00
  AND R3, R3, R4
  STR R3, [R2]
```

Figure 1 HEX_clear_ASM Partial code

ECSE 324 Lab Report 2
Chenyi Xu
260948311

Task 2:

Three more subroutines are required in this session to control the timer. The goal of task 2 is to create a simple stopwatch using the ARM A9 private timer. The stopwatch count in increments of 100 milliseconds. Milliseconds are displayed on HEX 0, seconds on HEX2-1, minutes on HEX 3-4, and hours on HEX 5. Pushbutton zero to two will be used to start, stop and reset the stopwatch. F bits are polled using an endless loop.

Three subroutines used to control the timer are ARM_TIM_config_ASM, ARM_TIM_read_INT_ASM, and ARM_TIM_clear_INT_ASM. ARM_TIM_config_ASM is used to configure the timer. R0 which is the load value and R1 which is the configurations bits were used as the input argument for this subroutine. The load value was stored into the address of Load which is 0xFFFEC600 and the configuration bits were stored into Control which is 0xFFFEC608 based on the manual. ARM_TIM_read_INT_ASM return the F value bit of the interrupt register from address of Interrupt_status which is 0xFFFEC60C based on the manual. The output is stored in register R0. ARM_TIM_clear_INT_ASM clears the F value by writing a one into the interrupt status register.

Subroutines from task 1 was also used in this part. Registers R4 to R9 were used as counters for HEX 0-5 respectively. Use HEX_write_ASM to initialize all display to 0. Move to the polling branch. The output from read_PB_edgecp_ASM was used to determine which button was pressed. If pushbutton 0 was pressed, branch to label pb0_start. If pushbutton 1 was pressed, branch to label pb1_stop. If pushbutton 2 was pressed, branch to label pb2_reset. The code for branch pb0_start is shown in *Figure 2*. The initial count is written to the load register R0. Configuration has three bits which are interrupt, auto and enable respectively. Setting the enable bit E in the control register to 1 can start the timer. It can be stopped by set E back to 0. When the auto bit is set to 1, the timer will reload the value. When the timer decrements its count value and reaches 0, the timer sets the F bit in the interrupt status register. Therefore, I masked bits to 011 in pb0_start, 000 in pb1_stop, and 011 in pb2_reset based on the aforementioned information. Then, use ARM_TIM_read_INT_ASM to return the interrupt register. Instruction TST is used to check whether the interrupt register is 0. If it is zero, branch to polling. Otherwise, move on to increase the counter. Since we are creating a timer, the hex display follows the convention of the regular timer. Hex 0 can only go up to 9. Once it reaches 9, Hex 1 will increase by 1. Once Hex 1 reaches 9, Hex 2 will count up by 2. Hex 2 has to be smaller than 6. Once it reached that, Hex 3 will increase by 1. Once Hex 3 reaches 9, Hex 4 will increase by 1. Hex 4 has to be smaller than 6 as well. Once it reaches that, Hex 5 will increase. Hex 5 can only go up to 9. Call the HEX_write_ASM subroutine to display the counter every time it updated. Then branch back to polling to achieve an endless loop. The sample output is shown in Figure 3.

One improvement is to add more comments. While writing the lab, I found out I would get confused by more own code sometimes because of the lack of comments. Comments makes the code more clear and more understandable. By improving this skill, I would significantly improve my own time efficiency and provides reader with a better understanding of my work.

```
pb0_start:
  LDR R0, TLoad
  MOV R1, #0b011
  BL ARM_TIM_config_ASM
  B polling
```

Figure 2 Partial code for Task 2

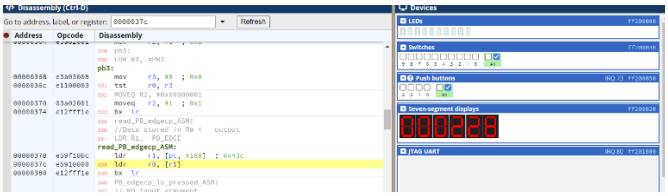

Figure 3 Task 2 Sample Output

ECSE 324 Lab Report 2
Chenyi Xu
260948311

Task 3:

The output of task 3 is the same with task 2. However, instead of polling, interrupts are used. When the process receives the interrupt, I will pause the current code execution, handles the interrupt by the Interrupt Service Routine, and resumes its normal execution. Six parts of the given template from the instructor were modified.

To begin with, call the enable_PB_INT_ASM subroutine where the input pushbutton indices is 1111. Then call the subroutine ARM_TIM_config_ASM with 110 as configuration bits and initial count as load value.

Under the IDLE session, the stopwatch function is implemented. The main idea of the function here is similar to the one from previous task. The bits pb0_start, pb1_stop, and pb2_reset were masked differently. For pb0_start, the configuration bits are masked to 111, For pb1_stop, the configuration bits are masked to 100. For pb2_reset, the configuration bits are masked to 110. Instead of pollig, interrupt were used in this part of the lab. Figure 4 shows how I check the interrupt bit. If it equals 0, branch back to the interrupt_loop. If it equals to 1, the function can continue the execution. The rest are the same with Task 2.

For the SERVICE_IRQ part, the function is modified to check both ARM A9 private timer and pushbutton interrupts and call the corresponding interrupt service routine. CMP instruction was used to compare with the interrupt IDs to see if the interrupt has occurred. The interrupt ID is 29 for timer and 73 for the pushbutton. If the ID is not recognized branch to the next check. If it is recognized, call the corresponding interrupt service routine.

The subroutine was modified to configure the ARM A9 private timer and pushbuttons interrupt in CONFIG_GIC part by passing the required interrupt ID. Use MOV instruction to move 29 into R0 and 1 into R1. Then branch link to CONFIG_INTERRUPT.

I rewrote the KEY_ISR part from scratch. To begin with, I called the read_PB_edgecp_ASM subroutine. The contents of pushbuttons edgecapture register is then written into the PB_int_flag memory. Last but not the least, I called the PB_clear_edgecp_ASM to clear the interrupt.

The ARM_TIM_ISR writes the value 1 into the tim_int_flag memory when an interrupt is received. The MOV instruction is used to store the value into tim_int_flag. Then it clears the interrupt by calling the subroutine ARM_TIM_clear_INT_ASM.

For this part, I added a lot of comments to help myself keep track of everything I wrote. It significantly improved my time efficiency and provided reader with a clear and logical idea of how the function flows.

```
LDR R0, =tim_int_flag
 LDR R2, [R0]
 TST R2, #1
 BEQ interrupt_loop
```

Figure 4 Task 3 Partial Code