G21_Lab3_Report

The basic I/O features carried by the DE1-SOC computer FPGA board is introduced in this lab, including 10 slider switches, 10 LEDs, 4 pushbuttons and 6 7-Segment displays. Assembly drivers interfacing with such I/O devices were written to test basic operations on the FPGA board. Furthermore, with working I/O drivers, one "Stopwatch" program was required to be written in C language with the implementation of the I/O drivers.

1. Basic I/O

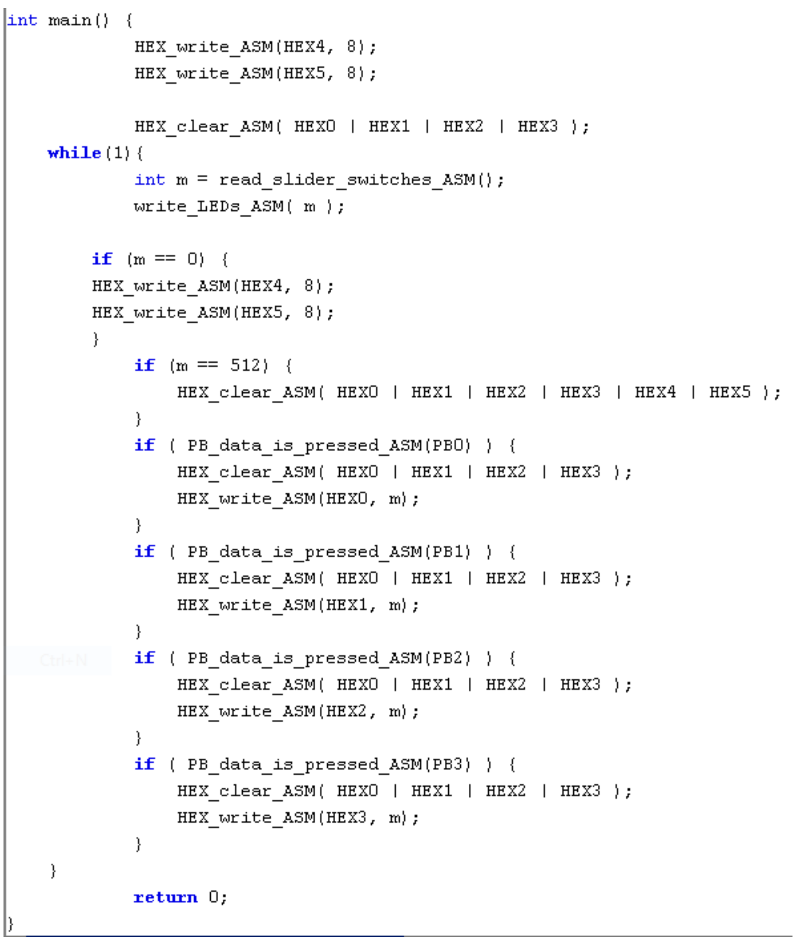
For slider switches, the slider_switches.s file contains the read_slider_switches_ASM subroutine, the slider_switches.h file declares the function, make it ready to be used in main.c. read_slider_switches_ASM reads the value stored at address 0xFF200040 in the main memory, and load that value into R0 register. For LEDs, there are two functions implemented: read_LEDs_ASM and write_LEDs_ASM, these functions load the value at 0xFF200000 to R0 and write R0 value to 0xFF200000 respectively, enables the control of the on/off on the LEDs. The value loaded or stored is a 10-bit binary number, matching the LEDs 0-9. Now with all that, write_LEDs_ASM (read_slider_switches_ASM()) putting in an infinity while loop in main.c will light up the respective LEDs when the matching slider switches are turned on.

Driver for HEX displays were implemented after the previous part. In HEX_displays.s file, there are in total 3 functions, HEX_flood_ASM, HEX_clear_ASM and HEX_write_ASM. HEX_flood_ASM would turn on all the 7-Segment displays by storing an eight-bit binary #0b11111111 to the address corresponding with HEX0-5. Reversely, HEX_clear_ASM would clear out all 7-Segment displays by storing #0b00000000 to the correct addresses. Stated by the DE1-SOC user manual, HEX0-3 have the address 0xFF200020 and HEX4-5 have 0xFF200030, and each single HEX display occupies 8-bit on each address. This is realised by implementing a loop in the assembly, writing #0b11111111 or #0b00000000 while iterating from HEX0-3, and jump to another loop which only in charge for HEX4-5 since they have a different and discontinuous address than HEX0-3.

One strange bug was encountered in HEX_flood_ASM. When the input for HEX_flood_ASM is HEX1 | HEX4, HEX0 | HEX5, or any other input combination containing HEX1 | HEX4 or HEX0 | HEX5, HEX4 and HEX5 will always be light up. There was no error occurred with any other HEX input combinations. Logic failures could no be found in the c program or the assembly program, the loop in assembly program only iterates through HEX0-5 once, the reason for the bug could be some random overflow on the memory digits when iterating from HEX0 to HEX5. To fix this, but without any logic fails found, #0b00000000 was forcibly written on HEX4 or HEX5 if they were in combination with HEX1 or HEX0 using an extra loop.

Finally, one simple pushbutton application was written in C, with the use of pushbuttons.s and pushbuttons.h. read_PB_data_ASM returns a binary string by reading the bits at 0xFF200050, where the final 4 bits hold the status of the buttons (1 for pressed and 0 for not). PB_data_is_pressed_ASM checks if the indicated buttons are pressed by taking each pushbutton of PB0-3 as input; if yes return 1; otherwise return 0. This is carried out by an AND operation between the input and the value stored at 0xFF200050, if the AND result is 0, return false on R0, otherwise return true on R0. read_PB_edgecap_ASM returns a binary string, where the final 4 bits hold the edgecap bits. PB_edgecap_is_pressed_ASM checks if the indicated buttons are pressed. PB_clear_edgecap_ASM writes input string into the edge capture memory location. enable_PB_INT_ASM writes the input string to the interrupt mask memory location, and lastly disable_PB_INT_ASM writes the input's opposite to the interrupt mask location.

The application simply requires LEDs to be light up when corresponding slider switches are turned on, while pushbuttons0-3 controls which one of HEX0-3 is activated showing the hexadecimal number 0-F whose input comes from the 4-bit binary number represented by slider switches0-3 (HEX4-5 is always activated); with slider switch9 controls the clearance of all HEX displays.



```
int main() {
            HEX_write_ASM(HEX4, 8);
            HEX_write_ASM(HEX5, 8);

            HEX_clear_ASM( HEX0 | HEX1 | HEX2 | HEX3 );
    while(1){
            int m = read_slider_switches_ASM();
            write_LEDs_ASM( m );

        if (m == 0) {
        HEX_write_ASM(HEX4, 8);
        HEX_write_ASM(HEX5, 8);
        }
            if (m == 512) {
                HEX_clear_ASM( HEX0 | HEX1 | HEX2 | HEX3 | HEX4 | HEX5 );
            }
            if ( PB_data_is_pressed_ASM(PB0) ) {
                HEX_clear_ASM( HEX0 | HEX1 | HEX2 | HEX3 );
                HEX_write_ASM(HEX0, m);
            }
            if ( PB_data_is_pressed_ASM(PB1) ) {
                HEX_clear_ASM( HEX0 | HEX1 | HEX2 | HEX3 );
                HEX_write_ASM(HEX1, m);
            }
            if ( PB_data_is_pressed_ASM(PB2) ) {
                HEX_clear_ASM( HEX0 | HEX1 | HEX2 | HEX3 );
                HEX_write_ASM(HEX2, m);
            }
            if ( PB_data_is_pressed_ASM(PB3) ) {
                HEX_clear_ASM( HEX0 | HEX1 | HEX2 | HEX3 );
                HEX_write_ASM(HEX3, m);
            }
    }
            return 0;
}
```

As shown in the figure, in the while loop HEX4-5 are always flooded, if the slider switch9 is on while others are off, HEX0-5 are cleared. The following 4 ifs captures which pushbutton is pressed and write the input value. The reason for clearing out HEX0-3 before writing values is because if doesn't clear before write, some random segment would light up (this might because some unknown overflow of 1s within the memory).

2. Polling based stopwatch

In this part, we need to build a HPS timer driver and test it by creating a stop watch C file.

There are three subroutines in the assembly file which are configuration, read and clear. Since there are four timers (0-3), we need four different address for them (0xFFC08000, 0xFFC09000, 0xFFD00000 and 0xFFD01000).
The configuration subroutine is the foundation of the rest of the two subroutines since they all share the similar structure.

The main idea of this part is to go through each timer and configurator them with the corresponding parameters. Since there is only four of them, we didn’t use loop and instead, we write the configuration four times. This is the part that we believe need to be improved. It can be replaced by using a loop to check the configuration.
We set a counter R1 to be initially 1. And load input value into R3. Then use TST operation on R1 and R3. It the value is not zero, we start the configuration for the first timer. First, we load R2 to be the base address of the first timer. Before configuration, we disable the timer first and than, loading time-out, M, I, E bits from input to the timer. One thing needed to be noticed is that since MIE are three bits so we use ORR operation to configure at the same time. When the configuration for the first timer is done, we left shift the counter R1 one bit and let R2 to be the base address of the next timer. Similar as the previous process, we use test bit to compare new R1 and R3. We repeated the same procedure four times until all timers are configured.
For the read part, it’s basic the same ideal as the configuration, except that we read from the timer. Similarly, we build a counter R1 and load address of the first timer to R2 and read the S-bit of that timer. After this is done, left shift R1 by 1 bit and move on to the next timer. Finally, for the clear, everything is the same as the read procedure but except this time, we don’t read S-bit from the timer, we reset F-bit and S-bit in the operation for each timer.

For this subroutine, the major issue is how to find which timer need to be configured, cleared, or read. It will be much easier to use a loop and a pointer to figure out the timer. This is the part we can improve for this code.

For the stop watch in main(C code), we first configure two timers, one is for push-button to see if the push-button is activated. And the other one is for the time showing on the FPGA board.
First we need to configure two timers. Load corresponding parameters into each timer respectively. Then we build three counters for the display timer: millisecond, second

and minute to increase their value. Each counter need to update each time if any of them hit their limit. For the pushbutton, we set a time counter to check wheter the push button activate the timer or not. we use edgecapture to to check whether the corresponding pushbutton is pressed. For PB0, we reset the time counter to be 1 to start the stop watch and clear the edgecapture. For PB2, it stops the timer so we set the time counter to be 0. And for the reset button (PB2), we simply reset the three display counter ms, sec, min and the time counter to be 0 to reset.

For this part, the main challenge is to figure out how to set the range for each counter and how to convert them to char. Also, understanding how edgecapture works is cruitial for this part.

3. interrupt

The last part shares the similar idea with part 3. In the stop watch part, there are two timers to check the display and the pushbutton. In this part, when the pushbutton is not pressed, it sends an interrupt for start, stop and restart of the timer.

In ISR.s file, we only add few things. the FPGA_PB_KEYS_ISR subroutine, is a interrup service routine. First get the pushbutton which is pressed and than set the flag and when the interrupt is done, reset the interrupt.
For c file, the structure is very similar to the stop watch, except this time, we don’t set the timer for pushbutton but for the interrupts flag. When the flag is set to notice that PB1 is pressed, the timer starts. And same thing for the stop and rest.

Since the main structure is mostly the same as part 3, the only challenge we face is to figure out how the interrupt work with the pushbutton. The difference between the interrupt and the polling stop watch is that the interrupt timer only one time when the button is not hold. But the polling is activated when the button is pressed.