Lab 3: Basic I/O, Timers and Interrupts

For ECSE 324: Computer Organization

Report Written by

Antoine Wang (260766084)

Alfred Wang (260771591)

Due Nov 17th at 5PM
McGill University

1. Basic I/O

The first section of this lab is to implement basic input/output interfaces in assembly language for the system to access the input generated by the DEC-SO1 machine.

1.1 Slider switches

The algorithm of reading the input of slider switches is straight forward. Directly go to the memory location which holds the sliders' status (0xFF200040) and load it into the return register, the bit string in form of one-hot coding will be returned.

1.2 LEDs

The reading of LED lights' status is similar to what described in 1.1. For writing, a certain string of bit patterns indicating which light will be on needed to be stored to the memory location holding the LED status (0xFF200000) so that the light can be enabled. The writing is done by using STR in assembly.

1.3 HEX displays

Overall, the essence of writing to the HEX displays on the board is same as the writing for LEDs. However, since the HEX display has a more complicated arrangement of memory, the writing algorithm need to be slightly adjusted. There are two words in memory holding 6 HEX displays in total, illustrated by the code segment below.

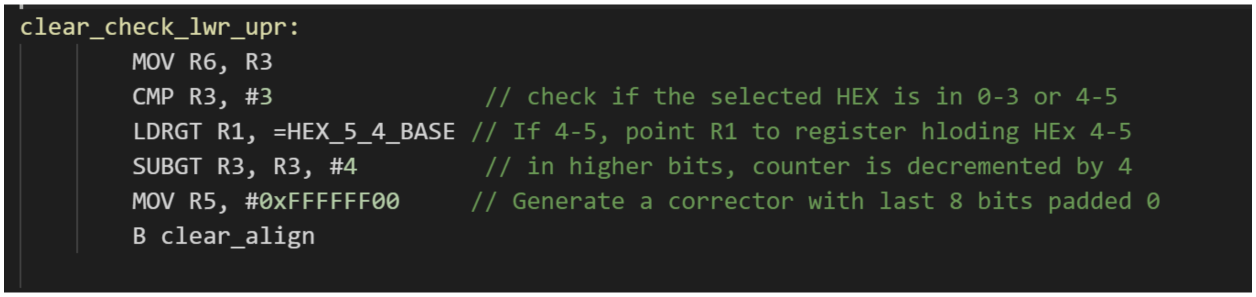
```
   .equ HEX_3_0_BASE, 0xFF200020
   .equ HEX_5_4_BASE, 0xFF200030
```

Another thing to notice is that the input, which selects which HEX display to change, is a enumeration using one-hot coding. Thus, if we want to change HEX0, HEX2 and HEX5 at the same time, we input "HEX0|HEX2|HEX5" as the argument and the corresponding bit pattern will be 100101. The ones on bit 0, 2 and 5 maps the choice. In the assembly implementation, we use this feature and designed a loop which shifts the input bit pattern one at a time to check through all 6 HEX displays. If a 1 exists, meaning the HEX is selected, then the program will enter a branch to find the corresponding 8 bits in memory. Then, the bit pattern will also be shifted to align with that 8 bits. Taking HEX_clear_ASM as an example.

To check which HEX is chosen, we use the following loop.

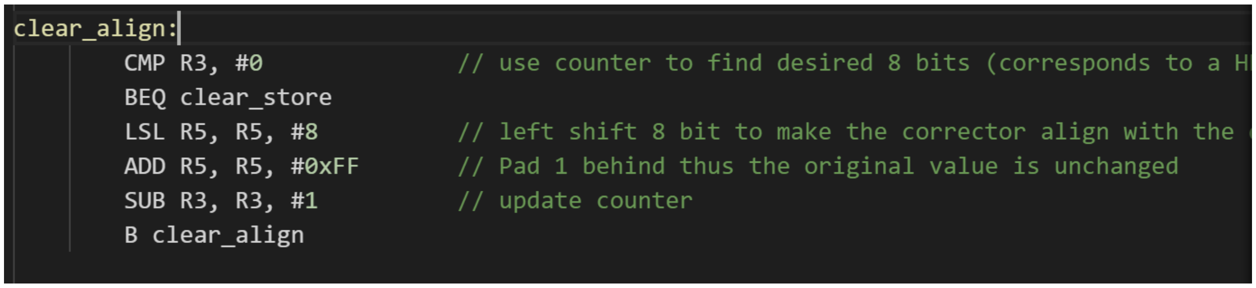
```
clear_LOOP:
        CMP R3, #6
        BEQ clear_check_lwr_upr
        AND R4, R0, #1          // compare the lower bit of R0 (one hot coded HEX)
        CMP R4, #1              // See if the Hex is selected
        BEQ clear_check_lwr_upr
        ASR R0, R0, #1          // SHift R0 right one bit to check the next input
        ADD R3, R3, #1          // update counter
        B clear_LOOP
```

To find corresponding memory location, we used a code block shown below.



```
clear_check_lwr_upr:
        MOV R6, R3
        CMP R3, #3               // check if the selected HEX is in 0-3 or 4-5
        LDRGT R1, =HEX_5_4_BASE // If 4-5, point R1 to register hloding HEx 4-5
        SUBGT R3, R3, #4         // in higher bits, counter is decremented by 4
        MOV R5, #0xFFFFFF00     // Generate a corrector with last 8 bits padded 0
        B clear_align
```

Then we use AND, the bit wise operation to clear all bits in the corresponding position. The rest bits holding the status for other HEX displays, which should not be changed, are ANDed with 1 to maintain its status. Code below shows how the zeros are targeted to the right position.

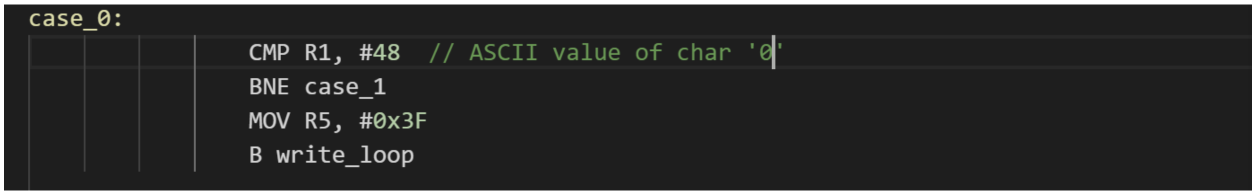


```
clear_align:
        CMP R3, #0              // use counter to find desired 8 bits (corresponds to a H
        BEQ clear_store
        LSL R5, R5, #8          // left shift 8 bit to make the corrector align with the
        ADD R5, R5, #0xFF       // Pad 1 behind thus the original value is unchanged
        SUB R3, R3, #1          // update counter
        B clear_align
```

Finally, we store the pattern (which is in R5) to the memory location and finish the writing.

The rest two methods HEX_flood_ASM and HEX_write_ASM are almost identical operations as clearing. The only difference is that for flood, instead of changing all bits to zero, we need to enable all segments by setting those to ones. Opposite to AND with zero, for flood we use ORR with one.

What happens in HEX_write_ASM is that we first clear the chosen HEX, then we ORR the desired pattern into the memory using a similar loop. The way we transfer the input character (an ASCII value) into the HEX pattern is that we implement a switch loop listing all 16 possibilities. Take input '0' as example:

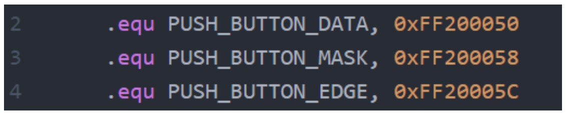


```
 case_0:
                CMP R1, #48  // ASCII value of char '0'
                BNE case_1
                MOV R5, #0x3F
                B write_loop
```

To sum up, the manipulation of the HEX display is straightforward. What makes it more challenging is that, first, we have to have a loop so that multiple selection of HEX is viable. Then, we constantly encounter errors when we use the command "STRB" (store byte) in assembly. We assume that partially tweak the values in a word causes internal errors or interruptions which causes random numbers contaminates the memory. This is the reason why we move the clear pattern step by step, each times by 8 bit, to align with the chosen HEX and always store the entire 32 bit pattern.

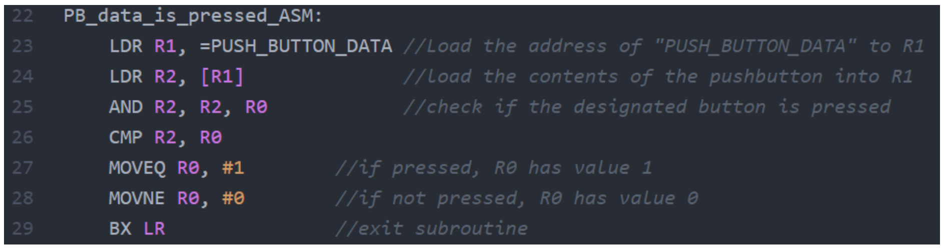
1.4 Push Button

There are three main parts for the section of pushbuttons. The first part is to access the data register of the pushbuttons. The second part is to access the pushbutton edge-capture register. And the third part is to access the pushbutton interrupt mask register. There are corresponding addresses for data, mask and edge of the four pushbuttons on the DE1-SoC board (shown below). The general ideal for implementation of all these functionalities is to access and rewrite the data in corresponding memories.



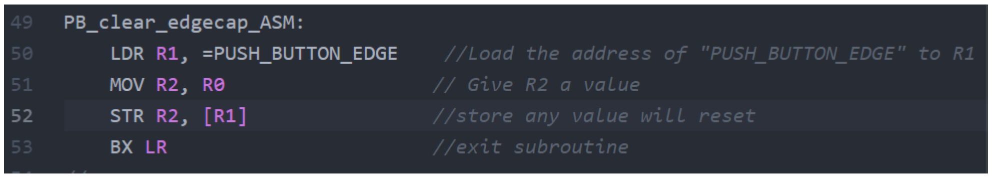
```
    .equ PUSH_BUTTON_DATA, 0xFF200050
    .equ PUSH_BUTTON_MASK, 0xFF200058
    .equ PUSH_BUTTON_EDGE, 0xFF20005C
```

For the push button data, there are two subroutines in this part, which are read_PB_data_ASM and PB_data_is_pressed_ASM. The implementation of PB_data_ASM is straightforward, we simply load the pushbutton data from PUSH_BUTTON_DATA address and get the first four one-hot bits of the data loaded to get the status of each pushbutton. The implementation of PB_data_is_pressed_ASM is slightly more complex. As the code shown below, the subroutine will take an input (R0) of desired pushbutton status. Then we get the current status of the pushbuttons and compare it with the desired status. If current status is the desired status, return 1; otherwise, return 0.



```
PB_data_is_pressed_ASM:
    LDR R1, =PUSH_BUTTON_DATA //Load the address of "PUSH_BUTTON_DATA" to R1
    LDR R2, [R1]              //Load the contents of the pushbutton into R1
    AND R2, R2, R0            //check if the designated button is pressed
    CMP R2, R0
    MOVEQ R0, #1        //if pressed, R0 has value 1
    MOVNE R0, #0        //if not pressed, R0 has value 0
    BX LR               //exit subroutine
```

For PB edge, there are three subroutines in this part, which are read_PB_edgecap_ASM, PB_edgecap_is_pressed_ASM and PB_clear_edgecap_ASM. The implementation of read_PB_edgecap_ASM is almost the same as PB_data_ASM, the only difference is that, instead of reading data from PUSH_BUTTON_DATA, data is loaded from PUSH_BUTTON_EDGE. The implementation of PB_edgecap_is_pressed_ASM and PB_data_is_pressed_ASM are similar in the same manner. For the implementation of PB_clear_edgecap_ASM, a little trick is used to make the code much shorter. According to the manual of the board, "writing any value to the Edgecapture register [can set] all bits of the Edgecapture register to zero". Therefore, as the code shown below, instead of setting each bit back to 0, we simply write any value to PUSH_BUTTON_EDGE and edgecapture register will be cleared automatically.



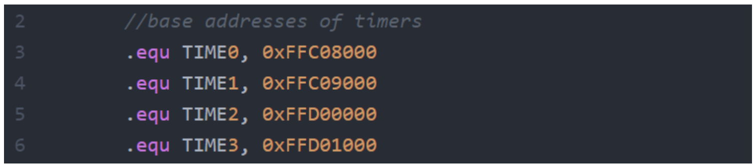
```
PB_clear_edgecap_ASM:
    LDR R1, =PUSH_BUTTON_EDGE    //Load the address of "PUSH_BUTTON_EDGE" to R1
    MOV R2, R0                  // Give R2 a value
    STR R2, [R1]                //store any value will reset
    BX LR                       //exit subroutine
```

For PB interruption, there are two subroutines in this part, which are enable_PB_INT_ASM and disable_PBINT_ASM. The implementation of these two subroutines are straightforward and similar. The general idea is to get the data in PUSH_BUTTON_MASK and change corresponding bits to 1 for enabling or 0 for disabling. The corresponding bits are determined from the input, which is in R0 in these cases.

2. Timers

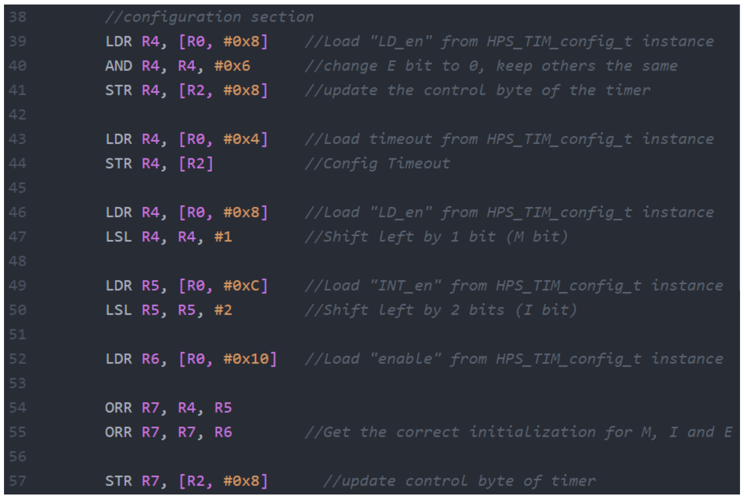
There are three main parts for this section, each has one subroutine. The first part is used to config timers using a subroutine called HPS_TIM_config_ASM. The second part is responsible for reading the interrupt status of the timer using a subroutine called HPS_TIM_read_INT_ASM. The third part is responsible for resetting the interrupt status of the timer using a subroutine called

HPS_TIM_clear_INT_ASM. The general idea for all these subroutines is to access the correct memory location and rewrite the data to desired ones. There are 4 timers on the board, with the addresses shown below.



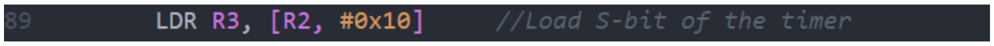
```
	//base addresses of timers
	.equ TIME0, 0xFFC08000
	.equ TIME1, 0xFFC09000
	.equ TIME2, 0xFFD00000
	.equ TIME3, 0xFFD01000
```

To config timers (**HPS_TIM_config_ASM**), all four timers will be checked if they are in use or not. As the code shown below, if the timer is in use, we load the timer into R2, and then we initialize the correct M, I, E bits to the designated timer.



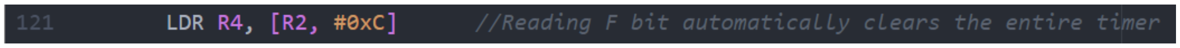
```
	//configuration section
	LDR R4, [R0, #0x8]	//Load "LD_en" from HPS_TIM_config_t instance
	AND R4, R4, #0x6	//change E bit to 0, keep others the same
	STR R4, [R2, #0x8]	//update the control byte of the timer

	LDR R4, [R0, #0x4]	//Load timeout from HPS_TIM_config_t instance
	STR R4, [R2]		//Config Timeout

	LDR R4, [R0, #0x8]	//Load "LD_en" from HPS_TIM_config_t instance
	LSL R4, R4, #1		//Shift left by 1 bit (M bit)

	LDR R5, [R0, #0xC]	//Load "INT_en" from HPS_TIM_config_t instance
	LSL R5, R5, #2		//Shift left by 2 bits (I bit)

	LDR R6, [R0, #0x10]	//Load "enable" from HPS_TIM_config_t instance

	ORR R7, R4, R5
	ORR R7, R7, R6		//Get the correct initialization for M, I and E

	STR R7, [R2, #0x8]	  //update control byte of timer
```

For subroutine **HPS_TIM_read_INT_ASM**, we will check the interrupt status of a given timer. To implement this, we first find out which timer is needed to be check, and then get the S-bit of the timer, and finally return it as the output of the method. The code below shows the access of the S-bit of a timer. (R2 holds the timer address)



```
	LDR R3, [R2, #0x10]	//Load S-bit of the timer
```

Subroutine **HPS_TIM_clear_INT_ASM** is responsible for clearing designated timers given by the input from R0. According to the manual of the board, there is a short path to clear each timer. Instead of changing each bit to desired values for clearing the timer, we can simply read the F-bit of each timer, and the timer will be cleared automatically. The code below shows the access of the F-bit of a timer. (R2 holds the timer)

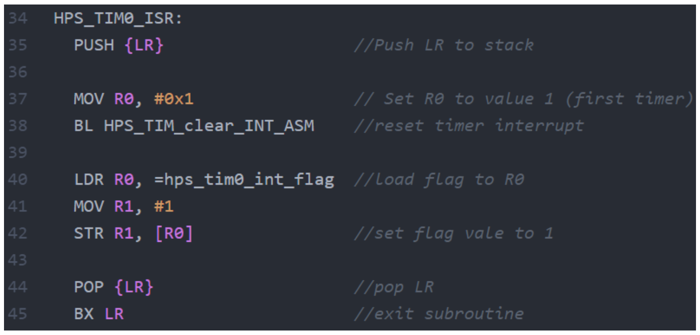


```
	LDR R4, [R2, #0xC]	//Reading F bit automatically clears the entire timer
```

3. Interruption

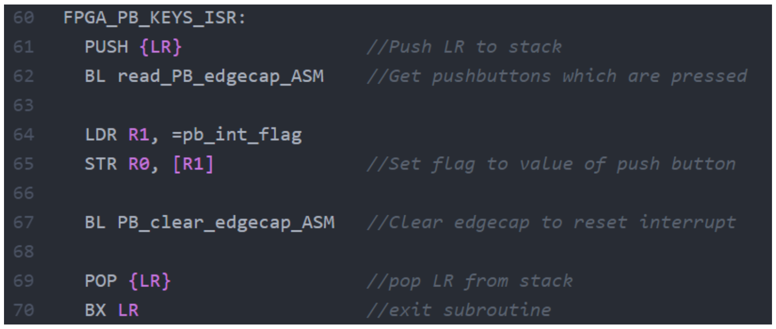
In order to handle interrupts, we need to tell the program what to execute when an interruption occurs, which is why there exists the Interrupt Service Routine (ISR). For this lab, interrupt will only occur when the timer needs to be updated or a push button is pushed. Therefore, only HPS_TIM0_ISR and FPGA_PB_KEYS_ISR need to be modified.

As the code for **HPS_TIM0_ISR** shown below, each time there is an interrupt for the timer, we clear the interrupt and set the interrupt flag to 1, so that the main.c file can know that there is an interrupt for the timer.



```
HPS_TIM0_ISR:
  PUSH {LR}                 //Push LR to stack

  MOV R0, #0x1              // Set R0 to value 1 (first timer)
  BL HPS_TIM_clear_INT_ASM  //reset timer interrupt

  LDR R0, =hps_tim0_int_flag //load flag to R0
  MOV R1, #1
  STR R1, [R0]              //set flag vale to 1

  POP {LR}                  //pop LR
  BX LR                     //exit subroutine
```

As the code for **FPGA_PB_KEYS_ISR** shown below, each time there is an interrupt for the pushbutton (press or release), we will read the edge status of the pushbutton, set the pushbutton flag to 1, and clear the status of the button for future uses.

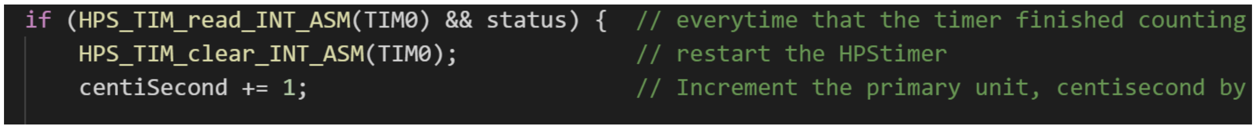


```
FPGA_PB_KEYS_ISR:
  PUSH {LR}                 //Push LR to stack
  BL read_PB_edgecap_ASM    //Get pushbuttons which are pressed

  LDR R1, =pb_int_flag
  STR R0, [R1]              //Set flag to value of push button

  BL PB_clear_edgecap_ASM   //Clear edgecap to reset interrupt

  POP {LR}                  //pop LR from stack
  BX LR                     //exit subroutine
```

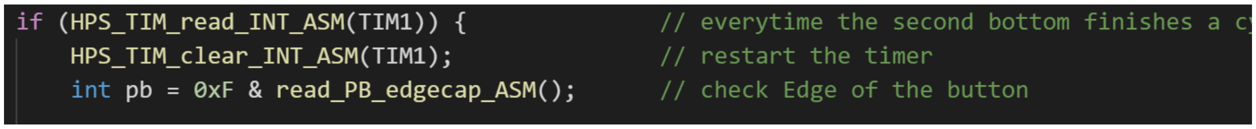
4. Overall Implementation in C

For the timer implementation in C, we use the internal counter implement on the DEC-SO1 machine. We configure the 100MHz machine to have a period of 1M so that every 0.01s a cycle is complete. Then we increment the variable for the stopwatch.

The main difference between two stopwatch configurations is that one is asynchronous while the other has a continuous polling loop. For the first timer, we list the two internal timers we used (TIM0 and TIM1) which the first operating at 0.01s and the second one polling at 5 microseconds. In another word, every 5 microseconds TIM1 will finish its countdown and check the push button.

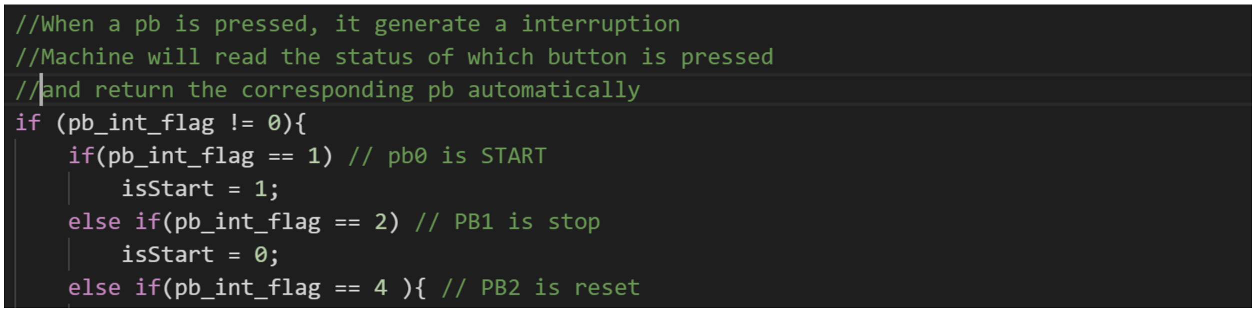


```
if (HPS_TIM_read_INT_ASM(TIM0) && status) {  // everytime that the timer finished counting
    HPS_TIM_clear_INT_ASM(TIM0);             // restart the HPStimer
    centiSecond += 1;                        // Increment the primary unit, centisecond by
```



```
if (HPS_TIM_read_INT_ASM(TIM1)) {          // everytime the second bottom finishes a c
    HPS_TIM_clear_INT_ASM(TIM1);           // restart the timer
    int pb = 0xF & read_PB_edgecap_ASM();  // check Edge of the button
```

Using interruption, we discover that the behavior of the timer becomes asychronous, meaning that the system needs not to poll the push button reading at any time. Instead, the program will be halted when a button press exists and the timer will respond correspondingly. In this way, the system's resource is highly saved since the 5-microsecond polling is revoked. The interruption timer thus has higher performance.



```
//When a pb is pressed, it generate a interruption
//Machine will read the status of which button is pressed
//and return the corresponding pb automatically
if (pb_int_flag != 0){
    if(pb_int_flag == 1) // pb0 is START
        isStart = 1;
    else if(pb_int_flag == 2) // PB1 is stop
        isStart = 0;
    else if(pb_int_flag == 4 ){ // PB2 is reset
```