EL 6483 F24 - MIDTERM PAPER

Real Time Embedded Systems (New York University)

ECE6483 Quiz Fall 2024 (Take home, Open book, notes and Laptop only)

Name: ARYAN AJMERA

ID: aa12904

NYU Tandon School of Engineering

Question 1: Consider the following ARM assembly code segment.

a. Accurately comment each line of code

b. Describe what parameters r0 and r1 (passed into the function) are used for.

c. What are each of the local variables r4-r8 used for?

d. What is the purpose of this function?

e. Explain in detail the specific purpose of “stmfd” and “ldmfd” in this function.

```
.globl _MyFunc
.text
_MyFunc:
 stmfd sp!, {r4, r5, r6, r7, r8, lr}
 cmp r1, #1
 ble end_outer

 sub r5, r1, #1
 mov r4, r0
 mov r6, #0

loop_start:
 ldr r7, [r4], 4
 ldr r8, [r4]
 cmp r7, r8
 ble no_go

 mov r6, #1
 sub r4, r4, 4
 swp r8, r8, [r4]
 str r8, [r4, 4]!
no_go:
 subs r5, r5, #1
 bne loop_start

end_inner:
 cmp r6, #0
 beq end_outer

 mov r6, #0
 mov r4, r0
 sub r5, r1, #1
 b loop_start

end_outer:
 ldmfd   sp!, {r4, r5, r6, r7, r8, pc}
```

Answer:

```
.globl _MyFunc                 ; Declare _MyFunc as a global
function for external access.
.text                           ; Start of the code section.
MyFunc:                         ; Function entry point label.
stmfd sp!, {r4, r5, r6, r7, r8, lr} ; Push registers r4-r8 and lr
onto the stack, adjusting sp.
cmp r1, #1                      ; Check if number of elements > 1.
ble end_outer                   ; If r1 <= 1, branch to end_outer.

sub r5, r1, #1                  ; Set r5 to r1 - 1.
mov r4, r0                      ; Copy r0 (array pointer) into r4.

mov r6, #0                      ; Initialize r6 to 0 (serves as a
flag).
loop_start:                     ; Start of the loop.
ldr r7, [r4], 4                 ; Load the value at r4 to r7, then
increment r4 by 4.
ldr r8, [r4]                    ; Load the next element from r4 into
r8.
cmp r7, r8                      ; Compare values in r7 and r8.
ble no_go                       ; If r7 <= r8, skip to no_go.

mov r6, #1                      ; Set flag r6 to 1 (indicating a
swap).
sub r4, r4, 4                   ; Adjust r4 back by 4.
swp r8, r8, [r4]                ; Swap r8 with memory at r4.
str r8, [r4, 4]!                ; Store r8 to r4 and increment r4 by
4.
no_go:                          ; Destination for branch.
subs r5, r5, #1                 ; Decrement r5 by 1 (loop counter).
bne loop_start                  ; If r5 != 0, repeat loop.

end_inner:                      ; End of inner loop.
cmp r6, #0                      ; Check if any swaps occurred.
beq end_outer                   ; If r6 == 0, branch to end_outer.

mov r6, #0                      ; Reset swap flag r6 to 0.
mov r4, r0                      ; Reset r4 to the array's start
(r0).
sub r5, r1, #1                  ; Reinitialize r5 with r1 - 1.
```

```
b loop_start                    ; Go back to loop_start for another
pass.

end_outer:                      ; End of outer loop.
ldmfd sp!, {r4, r5, r6, r7, r8, pc} ; Restore registers r4-r8 and
return by loading pc.
```

a. **r0**: This register holds the pointer to the start of an array or list of integers that the function will operate on.
   r1: This register holds the number of elements in the array (size of the array).

b. **r4**: Holds the pointer to the array. It is initialized with r0 and used throughout the function to iterate over the array elements.
   r5: Acts as a loop counter. It starts with r1 - 1 and is decremented in each iteration of the loop.
   r6: A flag indicating whether any swaps occurred in the current pass (set to 1 if a swap happens, otherwise it remains 0).
   r7: Temporarily holds one array element loaded from the address in r4.
   r8: Temporarily holds the next array element (or the swapped value) from the array during the comparison and swap.
c. The function sorts an array of integers in descending order using a modified bubble sort algorithm. It iterates through the array multiple times, comparing adjacent pairs of elements and swapping them if they are in the wrong order.
d. **stmfd sp!, {r4, r5, r6, r7, r8, lr}:** This instruction pushes the values of registers r4, r5, r6, r7, r8, and the link register (lr) onto the stack. The ! modifier indicates that the stack pointer is decremented before the values are pushed. This is used to save the current state of these registers before the function's execution.
   ldmfd sp!, {r4, r5, r6, r7, r8, pc}: This instruction pops the values of the same registers from the stack and restores them to their original values. The pc register is also popped, which causes the program to return to the instruction following the function call.

Question 2: Suppose I have the following C code snippet, which simply sums 6 numbers:

```
int sum6(int a1, int a2, int a3, int a4, int a5, int a6);

int main()
{
        int t;
        t=sum6(1,2,3,4,5,6);
        while(1);
}


int sum6(int a1, int a2, int a3, int a4, int a5, int a6)
{
        int total;
        total =a1+a2+a3+a4+a5+a6;
        return total;
}
```

a. Write the equivalent ARM assembly code in the following structure. Be sure to comment each line.

```
     AREA sum,
     CODE EXPORT
     main ALIGN
     ENTRY

__main PROC
     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     ; PUT YOUR CODE HERE
     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
stop B stop
     ENDP


sum6 PROC

     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     ; PUT YOUR CODE HERE
     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ENDP
END
```

Answer:

```
AREA sum, CODE
EXPORT __main
ALIGN
ENTRY

__main PROC
   MOV R0, #1         // Load 1st parameter
   MOV R1, #2         // Load 2nd parameter
   MOV R2, #3         // Load 3rd parameter
   MOV R3, #4         // Load 4th parameter
   PUSH{R1}            // Store 2nd parameter in the stack so that R1 can be used again as
                         only  r0-r3 registers can be used between function calls
   PUSH{R2}            // Store 3rd parameter in stack, freeing up R2 register
   MOV R1, #5         // Load fifth parameter
   MOV R2, #6         // Load sixth parameter
   BL sum6            // Call sum6 function

stop B stop
   ENDP

sum6 PROC

   ADD R0, R0, R1      // Add 1st and 5th numbers
   ADD R0, R0, R2      // Add 6th  number
   ADD R0, R0, R3      // Add 4th  number
   POP{R1}             // Get 2nd number and store it in R1
   POP{R2}             // Get 3rd number and store it in R2
   ADD R0, R0, R1      // Add 2nd number
   ADD R0, R0, R2      // Add 3rd  number
   BX LR               // Return with sum in R0

ENDP
END
```

Question 3: GPIO

We learned in class how to set up GPIO pins for input and output. We also know that the vendor specific HAL provides libraries like DigitalWrite, DigitalRead etc. But often these libraries are bloated, since they need to support so many different possible configurations and need to be generic. This question requires us to write these two functions ourselves, so that we can do GPIO more efficiently.

SPECIFICALLY USING THE REGISTERS IN LECTURE 6 (for Arm Cortex M0+), implement the following functions:

```
Int MyPortWrite(int MyPort, int WriteMask, int MyPinValues)
{
//My port is 0 or 1 selecting the port
//WriteMask holds a 32 bit int where there are 1’s in the bits/pins you wish to write
//MyPinValues has a 32 bit int with the write values for the pins indicated in WriteMask
//Example: WritePort = 0, WriteMask=0x00002401=Pins 0, 10, and 13 are being written
//MyPinValues = 0x00002400 = Write Port 0 Pin 0=0, Pin 10=1 and Pin 13 = 1 (ignore all
//others)
//Don’t forget to set DIR, INEN etc….And check for erroneous parameters
//Return 1 if successful, 0 if failed
}
```

```
Int MyPortRead(int MyPort, int *PullEnable)
{
//My port is 0 or 1 selecting the port
//PullUpEnable holds the address of a 32 bit integer that returns the bit mask of those
//pins already  configured with the Pull resistor enabled.
//Example: MyPort = 0,
//MyPortRead = 0x00002401 = All pins on Port 0 are 0, except Pin 0, 10 and 13
//*PullEnable returns (for example) 0x01200480, Pins 7, 10, 21, and 24 have pull resistor
//enabled
//Don’t forget to set DIR, INEN etc….And check for erroneous parameters
//You may assume the port has already been configured.
//Return 1 if successful, 0 if failed
}
```

Answer:

```
#include <stdint.h>
#include <stdio.h>

// Define the GPIO register structure
typedef struct {
    volatile uint32_t DIR;     // Direction register to set pins as input or
output
    volatile uint32_t OUT;     // Register to control output values on pins
    volatile uint32_t IN;      // Register to read input values from pins
    volatile uint32_t PULLEN;  // Register to enable pull-up or pull-down
resistors
    volatile uint32_t INEN;    // Register to enable input functionality on
pins
} GPIO_TypeDef1;

// Define base addresses for GPIO Port 0 and Port 1
#define GPIO0_BASE ((uint32_t)0x40020C00) // Base address for GPIO Port 0
#define GPIO1_BASE ((uint32_t)0x40020C04) // Base address for GPIO Port 1 with
a 4-byte offset

// Create pointers to GPIO Port 0 and Port 1 for direct access
#define GPIO0 ((GPIO_TypeDef1 *) GPIO0_BASE)
#define GPIO1 ((GPIO_TypeDef1 *) GPIO1_BASE)

int MyPortWrite(int MyPort, int WriteMask, int MyPinValues) {
    // Ensure the port number is valid (0 or 1)
    if (MyPort != 0 && MyPort != 1) {
        return 0; // Return 0 for an invalid port
    }

    // Get the GPIO port address based on the specified port number
    GPIO_TypeDef1 *GPIO = (MyPort == 0) ? GPIO0 : GPIO1;

    // Set specified pins as outputs by updating the DIR register
    GPIO->DIR |= WriteMask;

    // Disable input functionality for the specified pins by clearing bits in
INEN
    GPIO->INEN &= ~WriteMask;

    // Write values to output register for specified pins only
    GPIO->OUT = (GPIO->OUT & ~WriteMask) | (MyPinValues & WriteMask);

    return 1; // Indicate successful operation
}

int MyPortRead(int MyPort, int *PullEnable) {
    // Validate port number
    if (MyPort != 0 && MyPort != 1) {
        return 0; // Return 0 for invalid port
    }

    // Check if PullEnable pointer is valid
    if (PullEnable == NULL) {
        return 0; // Return 0 if PullEnable is a null pointer
    }

    // Get the GPIO port address based on the specified port number
    GPIO_TypeDef1 *GPIO = (MyPort == 0) ? GPIO0 : GPIO1;

    // Read the current input values from the IN register
    int pinValues = GPIO->IN;

    // Retrieve the pull-up/pull-down enable status from PULLEN
    *PullEnable = GPIO->PULLEN;

    return pinValues; // Return input pin values
}

int main() {
    int pullEnableMask;

    // Set values for pins 0, 10, 13 on Port 0 with specified WriteMask and
MyPinValues
    MyPortWrite(0, 0x00002401, 0x00002400);

    // Read the input pin states and pull resistor configuration from Port 0
    int pinStates = MyPortRead(0, &pullEnableMask);

    // Display the results (adjust output method as needed)
    printf("Pin States: 0x%08X, Pull Enable Mask: 0x%08X\n", pinStates,
pullEnableMask);
```

```
    return 0;
}
```