SYSC 2001 LAB5

This week, you will use indirect addressing modes to access elements of a record from an `array` of `struct`'s. You will also learn how to call subroutines and save & restore register states using the stack. Follow the instructions below. Write down your answers to any questions that appear **in bold**.

Part 0: Review Calling Subroutines, Parameter Passing Using Registers, and Saving & Restoring Registers

a) Calling subroutines using `CALL` & `RET`

Calling a subroutine in ASM is easy. You just write some code that starts with a label (i.e. the name of the subroutine) and ends with a `RET` instruction. The `RET` instruction is used at the end of the subroutine so that the `IP` will jump back to the calling code when the subroutine is done. See a sample subroutine below:

```
;;;;;;;;;;;;;;;;
; newLine: Subroutine to print a newline and a linefeed character
; Input parameters:
;       None.
; Output parameters:
;       None.

; Constants for this subroutine:
s_NL .EQU 0Dh          ; ASCII value for carriage return
s_LF .EQU 0Ah          ; ASCII value for line feed
Display .EQU 04E9h     ; address of Libra display

newLine:
        MOV AL, s_CR           ; Load carriage return (CR) into AL
        out DX,AL              ; print the char
        MOV AL, s_LF           ; Load line feed (LF) into AL
        out DX,AL              ; print the char

        RET                    ; Return to the calling code
```

Which would be called from code that looks like this:

```
main:
        ...
        CALL newLine    ; Print newline and linefeed characters
        ...             ; This instruction will be executed
                        ;  when we RETurn from newLine()
```

b) Parameter Passing Using Registers

Two ways to pass information into a subroutine is to pass by value or by reference using registers. The subroutine knows in which register to look for its input value (or the *address of* its input value when passing by reference).

Consider the following subroutine that prints a decimal digit between 0-9. It receives its input digit by value using register AL:

```
Display .EQU 04E9h     ; address of Libra display

;;;;;;;;;;;;;;;;
; printDigit: Subroutine to print a single decimal digit
; Input parameters:
;       AL: Unsigned decimal digit (between 0-9) to be printed
; Output parameters:
;       None.
printDigit:
        MOV DX, Display
        ADD AL, '0'             ; Convert number to ASCII code
        OUT DX,AL               ; Print it

        RET
```

This subroutine would be invoked using code such as the following:

```
num: .DB 5

main:
        ...
        MOV AL,[num]            ; Set up the input parameter
                                ;  (VALUE of digit to be printed goes in AL)
        CALL printDigit         ; Print '5' to the screen

        ...                     ; This instruction will be executed when
                                ;  we RETurn from printDigit()
```

Now consider the following subroutine that prints a string and receives its input by reference. It looks for the starting address of the string to be in register BX:

```
Display   .EQU 04E9h    ; address of Libra display

; printStr: Subroutine to print a '$'-terminated string
; Input parameters:
;    BX: Address of start of string to be printed
; Output parameters:
;    None.
printStr:
     MOV DX, Display
```

```
mainLoop:
     MOV AL, [BX]  ; Load next char to be printed - USING INPUT PARAMETER BX
     CMP AL, '$'   ; Compare the char to '$'
     JE quit       ; If equal, then quit subroutine and return to calling code
     OUT DX,AL     ; If it is not equal to '$', then print it
     INC BX        ; Point to the next char to be printed
     jmp mainLoop  ; Jump back to the top of the loop
quit:
     RET
```

This subroutine would be invoked using code such as the following:

```
str: .DB 'Hello New York!$'

main:
     ...
     MOV BX,str    ; Set up the input parameter (REFERENCE TO str goes in BX)
     CALL printStr ; Print 'Hello New York!' to the screen
     ...           ; This instruction will be executed
                   ; when we RETurn from printStr()
```

c) Saving and Restoring Registers

One problem with calling subroutines, such as the one above, is that the subroutine is likely to require registers in order to do its job. That means that the register values will likely be changed during the course of the subroutine. What about the values that were in those registers before the subroutine was called?? (i.e. the values that the 'main' or 'calling' code was using the registers to hold) These values would be lost, if we did not take care to save their original values at the start of each subroutine, and restore them at the end of each subroutine. In this way, the subroutine is free to modify any registers it likes in order to do its job, then restore the registers back to their original unmodified values when it is done with them, before returning to the calling code. The calling code doesn't need to know that the registers were saved, modified, then restored by the subroutine. This promotes code modularity, which allows each subroutine to be independent from each other (and potentially written by a different person!).

We will use PUSH to save a register, and POP to restore a register. This makes use of an area of memory called the stack. Due to the nature of the stack, registers must be popped in the reverse order that they were pushed. See the sample code below which demonstrates the principle of saving & restoring registers:

```
;;;;;;;;;;;;;;;;
; newLine: Subroutine to print a newline and a linefeed character
; Input parameters:
;       None.
; Output parameters:
;       None.

; Constants for this subroutine:
s_NL   .EQU 0Dh        ; ASCII value for carriage return
```

```
s_LF   .EQU 0Ah        ; ASCII value for line feed
Display .EQU 04E9h     ; address of Libra display

newLine:
        ; Save registers modified by this subroutine
        PUSH AX
        PUSH DX

        MOV AL, s_CR           ; Load carriage return (CR) into AL
        out DX,AL              ; print the char
        MOV AL, s_LF           ; Load line feed (LF) into AL
        out DX,AL              ; print the char

        ; Restore registers (in reverse order!)
        POP DX
        POP AX

        RET
```

Part 1: Saving & Restoring Registers

Download the source file lab5-P1.asm and save-as to your new "Lab5" directory on your M-drive. This file contains a number of subroutines that you will need to complete Part 2 below. The subroutines all work, except that the registers used in each subroutine need to be **saved and restored**.

Add the appropriate commands to each subroutine to save and restore the registers modified by each subroutine (and only those registers!).

Complete the `main` code in lab5-P1.asm to call each of the subroutines to test them. Use the comments in the code as hints for what needs to be added to `main()`...

Set a breakpoint in the middle of the `printDigit` subroutine *(at the* `ADD AL,'0'` *line).*

What is the value of the `SP` when the breakpoint is reached for the first time? Sketch the current contents of the stack (i.e. what register values are at which memory addresses on the stack). A partial table is given below: *Hint: Recall that `CALL` uses the stack to store the return address and `PUSH` uses the stack to store register values...*

Address	Value	Description
…	…	…
FFFBh		
FFFCh		
FFFDh		
FFFEh	?	Return address from `CALL printInt` (low byte)
FFFFh	?	Return address from `CALL printInt` (high byte)

Stop and demonstrate your answers above to the TA before proceeding to Part 2

Part 2: Printing a Record from an Array of Structures

Consider the following snippet of C-style code illustrating the Employee record format and the invocation of a subroutine, printEmployee():

```
Struct Employee
{
      char* name;         // 2-byte pointer to string of chars
      bool gender;        // 1-byte Boolean (zero-->male, else-->female)
      float salary;       // 1-byte unsigned integer (salary in $1000's)
      unsigned short id;  // 1-byte unsigned integer ID
};

Employee dayShift[1000];    // Array of Employee objects (dayShift DB)
Employee nightShift[100];   // Array of Employee objects (nighShift DB)
...
printEmployee(dayShift, 47);// Print the 48th dayShift worker to screen
printEmployee(nightShift, 0); // Print the 1st nightShift worker to screen
```

You are to implement "printEmployee" by filling in the blanks in the program provided in `lab5-P2.asm`. Assume you are given the starting address of the array of records in `BX` and the record number (i.e. array index) in `AL`.

Download the source file **`lab5-P2.asm`** and save-as to your new "Lab5" directory on your M-drive. This file contains all the comments for a program to print a record from an array of structures.

Copy and paste all the subroutines from your `lab5-P1.asm` file into the appropriate place holders in `lab5-P2.asm`. Note that you should NOT copy the `main` code to `lab5-P2.asm`, only the 5 subroutines.

Look at the `printEmployee()` subroutine and complete all the missing instructions. Use the comments as clues. Note that you should not have to add any additional lines to the program, just fill in the missing instructions.

Add instructions to the `main()` code to call your `printEmployee()` subroutine to implement the following C-style call:

```
printEmployee(dayShift, 0);    // Print the 1st dayShift worker to screen
printEmployee(dayShift, 3);    // Print the 4th dayShift worker to screen
printEmployee(nightShift, 0);  // Print the 1st nightShift worker to screen
```

Demonstrate and explain your code to a TA.

Remember to add comments to your program to explain any lines you change or add!
(Replace the "FIX ME" comments with actual comments)

Use the submit program **to submit your final `lab5-P2.asm` file. Good Luck!**