Text I/O & Functions with TASM
Due: February 3, 2020 on MyCourses by 23:55

In this lab we want to practice INTEL text I/O assembly programming.

You may want to become familiar with interrupt 21: http://spike.scu.edu.au/~barry/interrupts.html . You will likely be using functions 1, 6 or 8 for keyboard input, functions 2, 6, 8 or 9 for console output and function 4C for exiting the program. You will also want to review Lab 2.

In this lab you will be asked to write your first function.

PART 1 is not the lab. It is a review of assembler functions in INTEL. You will find that it is similar to other assembler languages but with unique INTEL syntax and architecture differences. Please take the time to review PART 1.

PART 2 is the actual assignment. Please use the SMALL memory model for this lab.

PART 1: INTEL Function programming review

Goal: Understand how functions are created in assembly, and what are the standard ways of passing parameters and returning values

STACK OVERVIEW

The stack is just a part of the RAM that is pointed to by the SS segment. Just like in RAM, the smallest addressable amount is of 1 byte, though you will mostly be manipulating the stack using the PUSH and POP instructions, which only work with words (2 bytes). One important thing to remember about the stack is that it grows in the opposite direction of the rest of the memory - that is, it grows from high memory locations to lower ones. If you want to add a value to the stack, you would use the PUSH instruction, and if you want to retrieve the last value PUSHed, you would use the POP instruction. Thus, the stack is a LIFO (Last In First Out) buffer. Finally, you should remember that just like any other address in RAM on the 8086, any stack address has a 20-bit absolute address which is composed of a segment and an offset. The segment of the stack if the SS register, and the offset is the SP register, and this always points to the last word PUSHed on the stack.

FUNCTION OVERVIEW

First, what is a function exactly? A function is pretty much just a label - and when you CALL that label, your program saves the offset of the intruction right after the function call (it pushes the instruction pointer (IP register) on the stack) and replaces IP with the offset of the label, effectively jumping to that label.

Once the function is done executing, there should be a "ret" instruction at the end, which tells the CPU to POP the return address from the stack (which should be the instruction pointer we PUSHed earlier) and use it as the current IP, effectively jumping back to the instruction following the function call.

SIMPLE FUNCTION

Thus, the simplest assembly function can be written like this:

```
=========
...
call simple_function          ← this is the function call
...
simple_function:              ← this is a label
ret                           ← this is the return command
...
=========
```

This would call a function that immediately returns.

RETURNING A VALUE

The next step would be to write a function that take no parameters but returns a value. Normally, the return value should be in AX. Thus, a function that does nothing but returns 0 would look like this:

```
========
...
return_zero:
mov ax, 0
ret
...
========
```

PRESERVING DATA

One important thing to consider is that functions that actually do something tend to use and modify registers, and you should preserve as much registers as possible (all but AX if possible) to not modify any data or operations that the calling code was doing with the registers. To do so, you should PUSH registers that your function uses to the stack before executing the function body, and then POP them back (in reverse order, as the stack is LIFO) just before returning.

Here is an example of a function that would modify pretty much all the registers in the body:

```
============
...
do_stuff:

push bx
push cx
push dx
push si
push di

; operations that modify all those registers

pop di
pop si
pop dx
pop cx
pop bx

ret
...
============
```

Note that you should not assume any registers will be preserved after any interrupt, so if your function uses any interrupts you should make sure that all the registers that are possibly modified are PUSHed/POPed.

PASSING PARAMETERS

Now we come to the interesting part - passing parameters on the stack. The idea is to PUSH parameters on the stack, in reverse order of their definition, and the function will then retrieve these parameters by accessing the stack. Finally, given that the stack pointer (SP register) was modified from all this PUSHing, you need to ADD the right value to SP (because the stack grows downwards by

decrementing SP) to make sure the stack is balanced after the function call. This stack balancing task should be done by the function.

Note that what I described above is called the stdcall calling convention. There is also the C calling convention, among others, where you PUSH the parameters in the reverse order once again, but it is up to the caller to balance the stack frame rather than the function. I suggest you stick to the stdcall calling convention as it is widely used in the Windows API among other areas, and it is very convenient except for when passing a variable number of parameters - in that case you should use the C calling convention. This is because when passing a variable number of parameters, only the You will also see that the calling convention is very important when doing inline assembly or linking to modules from other languages.

Here is an example of a function with 2 parameters, both WORD sized (16 bits or 2 bytes):

```
==========
...
sum_words:

push bp ; save BP
mov bp, sp ; set base pointer

push dx ; preserve DX, we will be using it

mov ax, WORD PTR ss:[bp+4] ; retrieve parameter 1
mov dx, WORD PTR ss:[bp+6] ; retrieve parameter 2
add ax, dx ; add both of them together, result being stored in AX

pop dx ; restore DX

mov sp, bp ; restore original SP
pop bp ; restore BP

ret 4 ; return and ADD 4 to SP
...
==========
```

The function above takes in 2 word sized parameters, adds them together, and returns the result in AX.

The function prototype would look like this:
WORD sum_words(WORD param1, WORD param2)

Notice that the first thing we do is save BP to the stack and set it the the stack pointer (SP). This is so that we have a stable reference point in our stack with which we will be referencing our stack parameters relative to that point. This also serves as making sure that SP preserves its value after the function body, as we restore SP based on BP before returning.

Let's now calculate where relative to BP our parameters will be on the stack - to do so here is a piece of code that would call the above function:

```
===========
...
mov ax, 10
push ax
mov ax, 1
push ax
call sum_words
mov dx, ax
...
```

==========

Which is equivalent in a high level language to:
DX = sum_words(1, 10)

Notice the parameters are pushed in reverse order, this is a convention for stdcall and is also very convenient in a few situations that you may encounter later on.

Here is what happens from the stack's point of view:
-You PUSH parameter 2
-you PUSH parameter 1
-you CALL sum_words => IP is PUSHed on the stack
-you PUSH BP
-you PUSH DX
-you POP DX
-you POP BP
-you "ret" from sum_words => IP is POPed off the stack
-the "ret" instruction ADDs 4 to SP, equivalent to POPing two words from the stack

Here is the stack frame once the code reaches sum_words' body, "x" being a given reference address:
==========
ADDRESS | VALUE
x - 0 | param2
x - 2 | param1
x - 4 | IP
x - 6 | BP <<== BP points here
x - 8 | DX <<== SP points here
==========

First recall that SP points to the last PUSHed item.
Also note that all the values we are pushing in our example are WORD sized, that is they are 2 bytes wide, and so our stack will always grow or shrink by 2.

Now note that BP points to its own current location in the stack segment, 2 bytes above it is the return address (IP), 2 bytes above that is the first parameter, and 2 bytes above that is the second parameter.

Thus, ss:[BP+4] is first parameter, and ss:[bp+6] is the second parameter. Notice that we use "ss:" to specify explicitly that we are referring to the stack segment (SS register) rather than the default data segment (DS register).

Finally, note that at the end of the function, the "ret" instruction has a parameter - it specifies what value to ADD to the SP register in order to correct the stack frame. Because we previously PUSHed two WORD sized parameters on the stack (param2 and param1), we balance the stack frame by ADDing 4 (2 * 2 bytes) to the SP register.

Note that there are many ways of making all this process a lot simpler, such as using the PROC/ENDP directives, specifying the function prototype and using INVOKE or specific instructions that automate a few steps here and there, among other things. But I recommend you start out by writing function with stack parameters the "raw" way, and once you get a feel for it you can try out a few shortcuts, which still require you to understand what is going on.

SYMBOLIC CONSTANTS

One neat shortcut I do suggest you use is using the EQU directive to name your parameters. For example, the above sum_words could be re-written as:

```
==========
...
sum_words:

param1 EQU WORD PTR ss:[bp+4]
param2 EQU WORD PTR ss:[bp+6]

push bp
mov bp, sp

push dx

mov ax, param1 ; retrieve parameter 1
mov dx, param2 ; retrieve parameter 2
add ax, dx

pop dx

mov sp, bp
pop bp

ret 4
...
==========
```

Finally, note that I am assuming that you are using the "small" memory model, if not some of the things here might be a little more complicated - for example when CALLing a function from certain memory models that use more than one CODE segment, the assembler first PUSHes the current CODE segment before PUSHing the IP register, and so when referencing the variables on the stack from inside the function, they are 2 bytes further than they would normally be if there was only a single CODE segment.

Please use the SMALL memory model for this lab.

==========

PART 2: TASM PROGRAMMING

PART 2 is the assignment.
You should use lab 2 as your model for writing this lab.
This lab is a standard programming lab. It has nothing to do with graphics.

Write an INTEL assembler program using TASM in DOSBOX that does the following:

1. The program asks the user for an integer number. “Please input triangle size:”.
2. The program asks the user for a single character. “Please input triangle symbol”.
3. The program then displays a right-angle triangle, with its right-angle at the left side of the screen. The triangle will be the size indicated by the user. The triangle will be printed using the character input by the user.
    a. The program prompts for the size and symbol of the triangle from the main program.
    b. Printing the triangle occurs in the function: TRIANGLE. The size and symbol are passed as parameters to the triangle function. This function displays a single right-angle triable based on those parameters.
    c. The function returns to the main program. The function is void. The function signature, in C, would be: `void triangle(int size, char symbol);`
4. The program then terminates

Here is an example execution:

```
Please input triangle size: 4
Please input triangle symbol: X

X
XX
XXX
XXXX
```

The program ends.

NOTE: If the user inputs a number less than or equal to zero, then nothing is printed.

WHAT TO HAND IN

For PART 1 – nothing to hand in
For PART 2 – triangle.asm

HOW IT WILL BE GRADED

This lab is worth 20 points.

- Prompt triangle size, uses INT 21 . . . 1 pts
- Prompt triangle symbol, uses INT 21 . . 1 pts
- Prints triangle correctly, uses INT 21 . . 8 pts
- Prompts happen in main program . . . 1 pt
- Proper function call . . . . . 1 pt
- Proper function parameter passing . . . 3 pts
- Proper function return . . . . 1 pt
- Proper program termination . . . . 1 pt
- Runs under TASM within DOSBOX. . . 3 pts