Topic 6: Procedure and Macro in Assembly Language Program (16 Marks)

Procedures

Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

```
name PROC

    ; here goes the code
    ; of the procedure ...

RET
name ENDP
```

*name* - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures.

Probably, you already know that **RET** instruction is used to return to operating system. The same instruction is used to return from procedure (actually operating system sees your program as a special procedure).

**PROC** and **ENDP** are compiler directives, so they are not assembled into any real machine code. Compiler just remembers the address of procedure.

**CALL** instruction is used to call a procedure.

Here is an example:

```
ORG 100h
CALL m1
MOV AX, 2
RET ; return to operating system.
m1 PROC
MOV BX, 5
```
The above example calls procedure \texttt{m1}, does \texttt{MOV BX, 5}, and returns to the next instruction after \texttt{CALL: MOV AX, 2}.

There are basically three ways that data can be passed to a procedure, they’re actually pretty similar to what goes on in a C language program:

- **Passing using processor registers,**
- **Passing parameters using variable names,**
- **Passing parameters using the stack.**

**Passing using processor registers,**

This option uses the processor registers to hold the data that will be used by the procedure, much like the DOS/BIOS calls we’ve seen. Only a small number of parameters can be passed, but they could be pointers to much larger data items. The scope of the data is very limited and somewhat erratic since the register contents are quite transient.

**Passing parameters using variable names.**

This option is equivalent to using global variables. It comes with all of the dangers associated with using global variables.

**Passing parameters using the stack.**

For this option, data items are pushed on the stack prior to calling the procedure and are popped off the stack by the procedure. Care must be taken since the return address will be pushed the parameters, but proper stack manipulation effectively limits a variable’s scope. This is the most common way that the C language passes parameters.

There are several ways to pass parameters to procedure, the easiest way to pass parameters is by using registers, here is another example of a procedure that receives two parameters in \texttt{AL} and \texttt{BL} registers, multiplies these parameters and returns the result in \texttt{AX} register:
In the above example value of AL register is update every time the procedure is called, BL register stays unchanged, so this algorithm calculates 2 in power of 4, so final result in AX register is 16 (or 10h).

Here goes another example, that uses a procedure to print a Hello World! message:
; the string address should be in SI register:
print_me   PROC

next_char:
   CMP  b.[SI], 0 ; check for zero to stop
   JE   stop ;

   MOV  AL, [SI]     ; next get ASCII char.

   MOV  AH, 0Eh      ; teletype function number.
   INT  10h          ; using interrupt to print a char in AL.

   ADD  SI, 1        ; advance index of string array.

   JMP  next_char    ; go back, and type another char.

stop:
   RET                   ; return to caller.

print_me   ENDP

;==========================================================================

msg    DB  'Hello World!', 0   ; null terminated string.

END

"b." - prefix before [SI] means that we need to compare bytes, not words. When you need to compare words add "w." prefix instead. When one of the compared operands is a register it's not required because compiler knows the size of each register.

The Stack

Stack is an area of memory for keeping temporary data. Stack is used by CALL instruction to keep return address for procedure, RET instruction gets this value from the stack and returns to that offset. Quite the same thing happens when INT instruction calls an interrupt, it stores in stack flag register, code segment and offset. IRET instruction is used to return from interrupt call.

We can also use the stack to keep any other data, there are two instructions that work with the stack:

PUSH - stores 16 bit value in the stack.
**POP** - gets 16 bit value from the stack.

Syntax for **PUSH** instruction:

- PUSH REG
- PUSH SREG
- PUSH memory
- PUSH immediate

**REG**: AX, BX, CX, DX, DI, SI, BP, SP.

**SREG**: DS, ES, SS, CS.

**memory**: [BX], [BX+SI+7], 16 bit variable, etc...

**immediate**: 5, -24, 3Fh, 10001101b, etc...

Syntax for **POP** instruction:

- POP REG
- POP SREG
- POP memory

**REG**: AX, BX, CX, DX, DI, SI, BP, SP.

**SREG**: DS, ES, SS, (except CS).

**memory**: [BX], [BX+SI+7], 16 bit variable, etc...

Notes:

- **PUSH** and **POP** work with 16 bit values only!
- Note: **PUSH immediate** works only on 80186 CPU and later!

The stack uses **LIFO** (Last In First Out) algorithm, this means that if we push these values one by one into the stack:

1, 2, 3, 4, 5

the first value that we will get on pop will be 5, then 4, 3, 2, and only then 1.
It is very important to do equal number of **PUSH**s and **POP**s, otherwise the stack maybe corrupted and it will be impossible to return to operating system. As you already know we use **RET** instruction to return to operating system, so when program starts there is a return address in stack (generally it's 0000h).

**PUSH** and **POP** instruction are especially useful because we don't have too much registers to operate with, so here is a trick:

- Store original value of the register in stack (using **PUSH**).
- Use the register for any purpose.
- Restore the original value of the register from stack (using **POP**).

Here is an example:

```
ORG    100h
MOV    AX, 1234h
PUSH   AX          ; store value of AX in stack.
MOV    AX, 5678h   ; modify the AX value.
POP    AX          ; restore the original value of AX.
RET
END
```
Another use of the stack is for exchanging the values, here is an example:

```
ORG 100h
MOV AX, 1212h ; store 1212h in AX.
MOV BX, 3434h ; store 3434h in BX

PUSH AX ; store value of AX in stack.
PUSH BX ; store value of BX in stack.

POP AX ; set AX to original value of BX.
POP BX ; set BX to original value of AX.

RET
END
```

The exchange happens because stack uses **LIFO** (Last In First Out) algorithm, so when we push `1212h` and then `3434h`, on pop we will first get `3434h` and only after it `1212h`.

The stack memory area is set by **SS** (Stack Segment) register, and **SP** (Stack Pointer) register. Generally operating system sets values of these registers on program start.

"**PUSH source**" instruction does the following:

- Subtract 2 from **SP** register.
- Write the value of **source** to the address **SS:SP**.

"**POP destination**" instruction does the following:

- Write the value at the address **SS:SP** to **destination**.
- Add 2 to **SP** register.
The current address pointed by **SS:SP** is called **the top of the stack**.

For **COM** files stack segment is generally the code segment, and stack pointer is set to value of **0FFFEh**. At the address **SS:0FFFEh** stored a return address for **RET** instruction that is executed in the end of the program.

**Macros**

Macros are just like procedures, but not really. Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions. If you declared a macro and never used it in your code, compiler will simply ignore it. **emu8086.inc** is a good example of how macros can be used, this file contains several macros to make coding easier for you.

```
Macro definition:

name MACRO [parameters,...]

    <instructions>

ENDM
```

Unlike procedures, macros should be defined above the code that uses it, for example:

```
MyMacro    MACRO  p1, p2, p3
    
    MOV AX, p1
    MOV BX, p2
    MOV CX, p3

ENDM

ORG 100h

MyMacro 1, 2, 3

MyMacro 4, 5, DX

RET
```
The above code is expanded into:

MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX

Some important facts about macros and procedures:

- When you want to use a procedure you should use CALL instruction, for example:
  
  CALL MyProc

- When you want to use a macro, you can just type its name. For example:

  MyMacro

- Procedure is located at some specific address in memory, and if you use the same procedure 100 times, the CPU will transfer control to this part of the memory. The control will be returned back to the program by RET instruction. The stack is used to keep the return address. The CALL instruction takes about 3 bytes, so the size of the output executable file grows very insignificantly, no matter how many time the procedure is used.

- Macro is expanded directly in program's code. So if you use the same macro 100 times, the compiler expands the macro 100 times, making the output executable file larger and larger, each time all instructions of a macro are inserted.

- You should use stack or any general purpose registers to pass parameters to procedure.

- To pass parameters to macro, you can just type them after the macro name. For example:

  MyMacro 1, 2, 3

- To mark the end of the macro ENDM directive is enough.

- To mark the end of the procedure, you should type the name of the procedure before the ENDP directive.
Macros are expanded directly in code, therefore if there are labels inside the macro definition you may get "Duplicate declaration" error when macro is used for twice or more. To avoid such problem, use **LOCAL** directive followed by names of variables, labels or procedure names. For example:

```
MyMacro2    MACRO
    LOCAL label1, label2
    CMP  AX, 2
    JE label1
    CMP  AX, 3
    JE label2
    label1:
        INC  AX
    label2:
        ADD  AX, 2
ENDM

ORG 100h
MyMacro2
MyMacro2
RET
```

If you plan to use your macros in several programs, it may be a good idea to place all macros in a separate file. Place that file in *Inc* folder and use **INCLUDE file-name** directive to use macros. See *Library of common functions - emu8086.inc* for an example of such file.
### Difference between macros and procedures

<table>
<thead>
<tr>
<th>Macros</th>
<th>Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessed during assembly when name given to macro is written as an instruction in the assembly program.</td>
<td>Accessed by CALL and RET instructions during program execution.</td>
</tr>
<tr>
<td>Machine code is generated for instructions each time a macro is called.</td>
<td>Machine code for instructions is put only once in the memory.</td>
</tr>
<tr>
<td>This due to repeated generation of machine code requires more memory.</td>
<td>This as all machine code is defined only once so less memory is required.</td>
</tr>
<tr>
<td>Parameters are passed as a part of the statement in which macro is called.</td>
<td>Parameters can be passed in register memory location or stack.</td>
</tr>
<tr>
<td>I don’t use macros.</td>
<td>I do use procedures.</td>
</tr>
</tbody>
</table>

### Assemble Directives

- **PROC** Indicates the beginning of a procedure
- **ENDP** End of procedure
- **FAR** Intersegment call
- **NEAR** Intrasegment call

```assembly
procname PROC[NEAR/ FAR]  
    ...
    ...
    ...
    RET
pro name ENDP
```

```assembly
procname PROC[NEAR/ FAR]  
    ...
    ...
    ...
    RET
pro name ENDP
```
User defined name of the procedure

Examples:

<table>
<thead>
<tr>
<th>ADD64 PROC NEAR</th>
<th>The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>ADD64 ENDP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONVERT PROC FAR</th>
<th>The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>CONVERT ENDP</td>
<td></td>
</tr>
</tbody>
</table>

- **MACRO** Indicate the beginning of a macro
- **ENDM** End of a macro

General form:

```plaintext
macroname MACRO[Arg1, Arg2 ...]  Program statements in the macro
…                                                                                       |
…                                                                                       |
…                                                                                       |
macroname ENDM                                                                               
```

User defined name of the macro

Call Instruction

The CALL Instruction:

- Stores the address of the next instruction to be executed after the CALL instruction to stack. This address is called as the return address.

- Then it changes the content of the instruction pointer register and in some cases the content of the code segment register to contain the starting address of the procedure.
Types of CALL instructions:

- DIRECT WITHIN-SEGMENT NEAR CALL: produce the starting address of the procedure by adding a 16-bit signed displacement to the contents of the instruction pointer.

- INDIRECT WITHIN-SEGMENT NEAR CALL: the instruction pointer is replaced with the 16-bit value stored in the register or memory location.

- THE DIRECT INTERSEGMENT FAR CALL: used when the called procedure is in different segment. The new value of the instruction pointer is written as bytes 2 and 3 of the instruction code. The low byte of the new IP value is written before the high byte.

- THE INDIRECT INTERSEGMENT FAR CALL: replaces the instruction pointer and the contents of the segment register with the two 16-bit values from the memory.

The 8086 RET instruction:

- When 8086 does near call it saves the instruction pointer value after the CALL instruction on to the stack.

- RET at the end of the procedure copies this value from stack back to the instruction pointer (IP).
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>RET</td>
<td>Near return to calling procedure</td>
</tr>
<tr>
<td>CB</td>
<td>RET</td>
<td>Far return to calling procedure</td>
</tr>
</tbody>
</table>

**Description**

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The RET instruction can be used to execute three different types of returns:

- **Near return**--A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intersegment return.

- **Far return**--A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.

- **Inter-privilege-level far return**--A far return to a different privilege level than that of the currently executing program or procedure.

**Example**

Transfer control to the return address located on the stack.

```assembly
ret
```

Transfer control to the return address located on the stack. Release the next 16-bytes of parameters.

```assembly
ret $-32767
```

**Long Return (lret)**

```assembly
lret
lret imm16
```

**Operation**

return to caller

**Description**

The lret instruction transfers control to a return address located on the stack. This address is usually placed on the stack by a lcall instruction. Issue the lret instruction within the called procedure to resume execution flow at the instruction following the call.

**Writing and debugging programs containing procedures**
• Carefully work out the overall structure of the program and break it down into modules which can easily be written as procedures.

• Simulate each procedure with few instructions which simply pass test values to the mainline program. This is called as dummy or stubs.

• Check that number of PUSH and POP operations are same.

• Use breakpoints before CALL, RET and start of the program or any key points in the program.

Reentrant and Recursive procedures

• Reentrant procedures: The procedure which can be interrupted, used and “reentered” without losing or writing over anything.

• Recursive procedure: It is the procedure which call itself.

Writing and Calling Far procedures

• It is the procedure that is located in a segment which has different name from the segment containing the CALL instruction.
Accessing Procedure

Accessing a procedure in another segment

- Put mainline program in one segment and all the procedures in different segment.
- Using FAR calls the procedures can accessed as discuss above.

Accessing procedure and data in separate assembly module

- Divide the program in the series of module.
- The object code files of each module can be linked together.
- In the module where variables or procedures are declared, you must use PUBLIC directive to let the linker know that it can be accessed from other modules.
- In a module which calls procedure or accesses a variable in another module, you must use the EXTERN directive.

Writing and using Assembler Macros

Defining and calling a Macro without parameters

```
PUSH-ALL MACRO
  PUSHF
  PUSH AX
  PUSH BX
  PUSH CX
  PUSH DX
  PUSH BP
  PUSH SI
  PUSH DI
  PUSH DS
  PUSH ES
  PUSH SS
ENDM
```
Passing parameters to Macros

- The words NUMBER, SOURCE and DESTINATION are called as the dummy variables. When we call the macro, values from the calling statements will be put in the instruction in place of the dummies.

```
MOVE_ASCII Макро NUMBER, SOURCE, DESTINATION
    MOV CX, NUMBER ; Номер символов, который нужно перенести в CX
    LEA SI, SOURCE ; Вынести SI на ASCII source
    LEA DI, DESTINATION ; Вынести DI на ASCII destination
    CLD ; Увеличить указатели после перемещения
    REP MOVSB ; Копировать ASCII строку в новый адрес
ENDM
```