Assembly Language Lab 9

Macros: Definition and Invoke

A macro is a named block of assembly language statements. Once defined, it can be called many times in the program. It is not like procedures. When it is called, preprocessor of the assembler replace this call by the macro block of code. So, it does not need to CALL or RET instructions as it is even preprocessed before assembling. Macros should be defined before their use. They are often defined in the beginning of source code file.

Macro definition:

```
macroname MACRO [parm1, parm2,…]
     statement-list
ENDM
```

Example:

```
mPutChar MACRO char
     push EAX
     mov AL, char
     call writeChar
     pop EAX
ENDM
```

The name of this macro is mPutChar, and takes one parameter named char. This macro prints the given character on the screen.

Invoking Macros

To invoke (call) a macro, just write its name followed by parameters values if any.
Example:

```
nPutChar 'a'
nPutChar AL
```

When the preprocessor finds nPutChar, it copies macro's code block and inserts this block in the place of the macro invoke statement. Then it replaces parameters by the values given in the invoke statement in the inserted block. So, the actual generated code of the previous example is:

```
…
push EAX
mov AL, 'a' ;replace char parameter by 'a' call
writeChar
pop EAX
…
push EAX
```

```
     mov AL, AL ;replace char parameter by AL call
     writeChar
     pop EAX
     …
```

Example : WriteStr macro

This example defines a macro that prints string arrays on screen

```
include irvine32.inc

mWriteStr macro text          ;Macro Definition
     push edx
     mov edx, offset text
     call writestring
     call CrLf
     pop edx
endm

.data
str1 byte 'Hello world 1', 0
str2 byte 'Hello world 2', 0
.code
main proc

     mWriteStr str1           ;Macro Invokes
     mWriteStr str2
     exit
main endp
end main
```

You can see the generated code for the previous main procedure in disassembly window in VS Debugger as shown next:

```
Main proc

     ;mWriteStr
     str1 push edx
     mov edx, offset str1
     call writestring
     call CrLf
     pop edx

     ;mWriteStr
     str2 push edx
     mov edx, offset str2
     call writestring
     call CrLf
     pop edx
...
Main endp
```

Data definition in macros

A macro can include even data variables in its definition beside its code. But, it should be preceded by LOCAL directive to tell the preprocessor to create a unique label each time the macro invoked, otherwise this label will be redefined each time the macro invoked causing assembling error.

Example : WriteText macro

This example defines a macro that prints string constants on screen

```
include irvine32.inc

mWriteText macro text

     ;string is local variable used in macro so preprocessor
     will ;create a unique name each time the macro invoked
     local str1

     .data          ;define a data segment to store passed text
          ;allocate the string array named by the str1 label
          and ;initialized by the text parameter
          str1 byte text, 0

     .code          ;define a code segement to print the string
         push edx
         mov edx, offset str1
         call writestring
         call crlf
         pop edx
endm

.data

.code
main proc

     mWriteText 'hello world 1'

     mWriteText 'hello world 2'

     exit
main endp
end main
```

The generated code for the previous main procedure is shown next:

```
main proc

;mWriteText 'hello world 1'
.data
??0000 byte 'hello world 1', 0
.code
    push edx
    mov edx, offset ??0000
    call writestring
    pop edx

;mWriteText 'hello world 2'
.data
??0001 byte 'hello world 2', 0
.code
    push edx
    mov edx, offset ??0001
    call writestring
    pop edx

...
main endp
```

Note that ??0000 and ??0001 are valid labels, which are created by the preprocessor to assure the labels uniqueness.

Exporting Assembly Code to High-level Language

As we know, we will not use assembly code to build large sophisticated project. It is too hard to do so. Therefore, we have to know how to use assembly code in well-known high-level languages to take the advantages of both. In next two sections, we will show how to export our assembly code to a dynamic-linking library (.dll) to be used in any other high level language.

1) Exporting Assembly Code to DLL Library

To export your code to a .dll library, there are minor changes to be made on the skeleton program we use in generating normal .exe applications. This modified skeleton is shown next:

```
.386
.model flat, stdcall

.data
;Write your static data, if any, here

.code
;Write your procedures to be exported here

DllMain PROC hInstance:DWORD,
          fdwReason:DWORD, lpReserved:DWORD

   mov eax, 1 ; Return true to caller. ret

DllMain ENDP

END DllMain
```

As shown above, it is not so different from the regular skeleton except that the entry point procedure, DllMain. The DllMain procedure must be defined exactly as specified to be able to link this code correctly. This constraint is required as Windows assumes the existing of this function in any DLL library to be able load it and so we defined it as Windows needs.

This DllMain function does not do anything except necessary initialization for loading the library; it does not call any other procedures like normal applications. This comes from the concept of the library. Library is just a collection of functions to be called from other applications and so the DllMain function has nothing to do.

On the other hand, the assembler should know what procedures to export. Consequently, you should write a separate definition file (.def) which contains procedures names to be exported.

Example: First Assembly DLL library

This example demonstrates how to create a DLL library in assembly language. The following is the assembly source code (main.asm):

```
include irvine32.inc

.data
;no static data
.code

Sum PROC int1:DWORD,
     int2:DWORD mov eax, int1
     add eax, int2
     ret
Sum ENDP
```

```
SumArr PROC arr:PTR DWORD, sz:DWORD

     push esi
     push ecx

     mov esi, arr
     mov  ecx,  sz
     mov eax, 0
sum_loop:
     add eax, DWORD PTR [esi]
     add esi, 4
     loop sum_loop

     pop ecx
     pop esi
     Ret

SumArr ENDP

ToUpper PROC str1:PTR BYTE, sz:DWORD

     push esi
     push ecx

     mov esi, str1
     mov ecx, sz
l1:
     cmp byte ptr [esi], 'a'
     jb skip
     cmp byte ptr [esi], 'z'
     ja skip
     and byte ptr [esi], 11011111b

skip:
     inc esi
     loop l1

     pop ecx
     pop esi
     ret
ToUpper ENDP


; DllMain is required for any DLL
DllMain PROC hInstance:DWORD, fdwReason:DWORD, lpReserved:DWORD

     mov eax, 1 ; Return true to caller. ret

DllMain ENDP

END DllMain
```

The definition file (main.def) contains a list of names of exported procedures combined with an ordinal number which specify the order of the procedure in the library. Note that, if you did not fill this file, no function will be exported to the DLL library. Also, you must write names of procedures exactly the same as defined in the .asm file (and matching the letter case too). The following is the content of main.def file to export the three functions defined in the source file (main.asm):

```
EXPORTS
   Sum @1
   SumArr @2
   ToUpper @3
```

To build this DLL library, open the .NET project attached with this lab in the DllProject_template folder. Copy and paste the source code shown here in main.asm file and fill main.def file as specified and build the project. A DLL library will be generated in Debug folder. You can use this DLL in any other high-level language and use functions defined in it. We will demonstrate in the next section how to do so with C#.

2) Importing DLL library into C#

Now, we shall explain how to use the generated DLL library in C# projects. Create a new console C# application in VS. NET, and then write the following code:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;

namespace DllTest
{
    class Program
    {
        [DllImport("Project.dll")]
        private static extern int Sum(int y, int b);

        [DllImport("Project.dll")]
        private static extern int SumArr([In] int[] arr, int sz);

        [DllImport("Project.dll")]
        private static extern void ToUpper([In, Out] char[] arr, int sz);

        static void Main(string[] args)
        {
            int[] x = { 1, 2, 3 };
            char[] c = "How are u?".ToCharArray();

            //test SumArr procedure
            Console.Write(SumArr(x, x.Length));

            //test ToUpper procedure
            Console.Write(c, 0, c.Length);
            ToUpper(c, c.Length);
            Console.Write(c, 0, c.Length);

        }
    }
}
```

To compile this code, you should put the `Project.dll` file in the bin\debug folder of your C# project (beside the .exe file) or you should write the full path of the DLL file within DllImport.

Note that you need to choose accurate parameter type when using DllImport otherwise function will not work probably and may cause errors. For more information, you can read about "Interop Marshaling" in MSDN library.

Assignment #6 (Bonus):

1. Write a macro with format ReadInt32 dest that reads an integer from user and returns the input integer in the dest parameter.

2. Write a macro with format MULT dest, src that multiplies 32-bit source and destination operands and put result in destination.