Lab 4 : MACRO VÀ THỦ TỤC

1. **Chuẩn đầu ra :** Sau bài này, người học có thể :
   - ✓ Tạo và sử dụng macro.
   - ✓ Tạo và sử dụng thủ tục
2. **Chuẩn bị :** Đọc trước phần lý thuyết về macro và thủ tục.
3. **Phương tiện :**
   - ✓ Máy vi tính.
   - ✓ Chương trình nasm.
4. **Thời lượng : 4 tiết**
5. **Command Line Arguments and the Stack**

   5.1. Function Definitions and Function Calls in 32-bit program

   The C calling convention in 32-bit programs is as follows. In the following description, the words *caller* and *callee* are used to denote the function doing the calling and the function which gets called.

   - The caller pushes the function's parameters on the stack, one after another, in reverse order (right to left, so that the first argument specified to the function is pushed last).
   - The caller then executes a near CALL instruction to pass control to the callee.
   - The callee receives control, and typically (although this is not actually necessary, in functions which do not need to access their parameters) starts by saving the value of ESP in EBP so as to be able to use EBP as a base pointer to find its parameters on the stack. However, the caller was probably doing this too, so part of the calling convention states that EBP must be preserved by any C function. Hence the callee, if it is going to set up EBP as a frame pointer, must push the previous value first.
   - The callee may then access its parameters relative to EBP. The doubleword at [EBP] holds the previous value of EBP as it was pushed; the next doubleword, at [EBP+4], holds the return address, pushed implicitly by CALL. The parameters start after that, at [EBP+8]. The leftmost parameter of the function, since it was pushed last, is accessible at this offset from EBP; the others follow, at successively greater offsets. Thus, in a function such as printf which takes a variable number of parameters, the pushing of the parameters in reverse order means that the function knows where to find its first parameter, which tells it the number and type of the remaining ones.
   - The callee may also wish to decrease ESP further, so as to allocate space on the stack for local variables, which will then be accessible at negative offsets from EBP.
   - The callee, if it wishes to return a value to the caller, should leave the value in AL, AX or EAX depending on the size of the value. Floating-point results are typically returned in ST0.
   - Once the callee has finished processing, it restores ESP from EBP if it had allocated local stack space, then pops the previous value of EBP, and returns via RET (equivalently, RETN).

- When the caller regains control from the callee, the function parameters are still on the stack, so it typically adds an immediate constant to ESP to remove them (instead of executing a number of slow POP instructions). Thus, if a function is accidentally called with the wrong number of parameters due to a prototype mismatch, the stack will still be returned to a sensible state since the caller, which *knows* how many parameters it pushed, does the removing.

There is an alternative calling convention used by Win32 programs for Windows API calls, and also for functions called *by* the Windows API such as window procedures: they follow what Microsoft calls the __stdcall convention. This is slightly closer to the Pascal convention, in that the callee clears the stack by passing a parameter to the RET instruction. However, the parameters are still pushed in right-to-left order.

Thus, you would define a function in C style in the following way:

```
global  _myfunc

_myfunc:
        push    ebp
        mov     ebp,esp
        sub     esp,0x40            ; 64 bytes of local stack space
        mov     ebx,[ebp+8]         ; first parameter to function

        ; some more code

        leave                       ; mov esp,ebp / pop ebp
        ret
```

At the other end of the process, to call a C function from your assembly code, you would do something like this:

```
extern  _printf

; and then, further down...

push    dword [myint]   ; one of my integer variables
push    dword mystring  ; pointer into my data segment
call    _printf
add     esp,byte 8      ; `byte' saves space

; then those data items...

segment _DATA

        myint   dd   1234
        mystring    db   'This number -> %d <- should be
        1234',10,0
```

This piece of code is the assembly equivalent of the C code

```
        int myint = 1234;
-       printf("This number -> %d <- should be 1234\n", myint);
```

5.2. Accessing Data Items

To get at the contents of C variables, or to declare variables which C can access, you need only declare the names as GLOBAL or EXTERN. (Again, the names require leading underscores, as stated in section 9.1.1.) Thus, a C variable declared as int i can be accessed from assembler as

```
        extern _i
        mov eax,[_i]
```

And to declare your own integer variable which C programs can access as extern int j, you do this (making sure you are assembling in the _DATA segment, if necessary):

```
        global _j
_j      dd 0
```

To access a C array, you need to know the size of the components of the array. For example, int variables are four bytes long, so if a C program declares an array as int a[10], you can access a[3]by coding mov ax,[_a+12]. (The byte offset 12 is obtained by multiplying the desired array index, 3, by the size of the array element, 4.) The sizes of the C base types in 32-bit compilers are: 1 for char, 2 for short, 4 for int, long and float, and 8 for double. Pointers, being 32-bit addresses, are also 4 bytes long.

To access a C data structure, you need to know the offset from the base of the structure to the field you are interested in. You can either do this by converting the C structure definition into a NASM structure definition (using STRUC), or by calculating the one offset and using just that.

To do either of these, you should read your C compiler's manual to find out how it organizes data structures. NASM gives no special alignment to structure members in its own STRUC macro, so you have to specify alignment yourself if the C compiler generates it. Typically, you might find that a structure like

```
struct {
        char c;
        int i;
        } foo;
```

might be eight bytes long rather than five, since the int field would be aligned to a four-byte boundary. However, this sort of feature is sometimes a configurable option in the C compiler, either using command-line options or #pragma lines, so you have to find out how your own compiler does it.

5.3. Interfacing to 64-bit C Programs (Unix)

The first six integer arguments (from the left) are passed in RDI, RSI, RDX, RCX, R8, and R9, in that order. Additional integer arguments are passed on the stack. These registers, plus RAX, R10 and R11are destroyed by function calls, and thus are available for use by the function without saving.

Integer return values are passed in RAX and RDX, in that order.

Floating point is done using SSE registers, except for long double, which is 80 bits (TWORD) on most platforms (Android is one exception; there long double is 64 bits and treated the same as double.) Floating-point arguments are passed in XMM0 to XMM7; return is XMM0 and XMM1. long double are passed on the stack, and returned in ST0 and ST1.

All SSE and x87 registers are destroyed by function calls.

On 64-bit Unix, long is 64 bits.

Integer and SSE register arguments are counted separately, so for the case of

void foo(long a, double b, int c)

a is passed in RDI, b in XMM0, and c in ESI.

5.4. Stack frame

Getting the command line arguments from a DOS program is not an enjoyable experience, because working with the PSP and having to worry about segments is simply a pain. In Linux things are much simpler: all arguments are available on the stack when the program starts, so to get them you just pop them off.
As an example, say you run a program called program and give it three arguments:
./program foo bar 42
The stack will then look as follows:

4	Number of arguments (argc), including the program name
program	Name of the program (argv[0])
foo	Argument 1, the first real argument (argv[1])
bar	Argument 2 (argv[2])
42	Argument 3 (argv[3]) (Note: this is the string "42", not the number 42)

Now lets write the program program that takes the three arguments:

```
section .text

    global _start
```

```
_start:

    pop   eax       ; Get the number of arguments

    pop   ebx       ; Get the program name

    pop   ebx  ; Get the first actual argument ("foo")

    pop   ecx       ; "bar"

    pop   edx       ; "42"

    mov   eax,1

    mov   ebx,0

    int   80h       ; Exit
```

After all that popping, EAX contains the number of arguments, EBX points to wherever "foo" is stored in memory, ECX points to "bar" and EDX to "42". This is obviously *way* more elegant and simple than in DOS. It took us just 5 lines to get the arguments and even how many there are, while in DOS it takes 14 rather complicated lines just to get *one* argument! Note that the 3rd pop overwrites the value we put in EBX with the 2nd pop (which was the program name). Unless you have a really good reason, you can usually chuck away the program name as we did here.

6. Macro

- Writing a macro is another way of ensuring modular programming in assembly language
- A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.
- In NASM, macros are defined with **%macro** and **%endmacro** directives.
- The macro begins with the %macro directive and ends with the %endmacro directive.
- The Syntax for macro definition

```
%macro macro_name  number_of_params
<macro body>
%endmacro
```

- Where, *number_of_params* specifies the number parameters, *macro_name*specifies the name of the macro.
- The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a

program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

-

7. Thủ tục

- Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size. Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well-defined job. End of the procedure is indicated by a return statement.
- The syntax to define a procedure

```
proc_name:
   procedure body
   ...
   ret
```

- NASM doesn't have procedure definitions like you may have used in TASM. That's because procedures don't really exist in assembly: everything is a label. So if you want to write a "procedure" in NASM, you don't use proc and endp, but instead just put a label (eg. fileWrite:) at the beginning of the "procedure's" code. If you want to, you can put comments at the start and end of the code just to make it look a bit more like a procedure. Here's an example in both Linux and DOS:

Linux

```
; proc fileWrite - write a string to a
file
fileWrite:
   mov eax,4                ; write
system call
   mov ebx,[filedesc]       ; File
descriptor
   mov ecx,stuffToWrite
   mov edx,[stuffLen]
   int 80h
   ret
; endp fileWrite
```

DOS

```
proc fileWrite
   mov ah,40h               ; write DOS
service
   mov bx,[filehandle]      ; File
handle
   mov cl,[stuffLen]
   mov dx,offset stuffToWrite
   int 21h
   ret
endp fileWrite
```

-

8. Nội dung thực hành

8.1. Example 1 :

```
; ----------------------------------------------------------------
; A 64-bit Linux application that writes the first 90 Fibonacci
numbers. To
; assemble and run:
; nasm -felf64 fib.asm && gcc fib.o && ./a.out
; ----------------------------------------------------------------
        global  main
        extern  printf
```

```
        section .text
main:
        push    rbx             ; we have to save this since we use it
        mov     ecx, 90         ; ecx will countdown to 0
        xor     rax, rax        ; rax will hold the current number
        xor     rbx, rbx        ; rbx will hold the next number
        inc     rbx             ; rbx is originally 1
print:
        ; We need to call printf, but we are using rax, rbx, and
rcx.  printf
        ; may destroy rax and rcx so we will save these before the
call and
        ; restore them afterwards.

        push    rax                       ; caller-save register
        push    rcx                       ; caller-save register
        mov     rdi, format               ; set 1st parameter (format)
        mov     rsi, rax                  ; set 2nd parameter
(current_number)
        xor     rax, rax                  ; because printf is varargs
        ; Stack is already aligned because we pushed three 8 byte
registers
        call    printf                ; printf(format, current_number)
        pop     rcx                   ; restore caller-save register
        pop     rax                   ; restore caller-save register
        mov     rdx, rax              ; save the current number
        mov     rax, rbx              ; next number is now current
        add     rbx, rdx              ; get the new next number
        dec     ecx                   ; count down
        jnz     print               ; if not done counting, do some more
        pop     rbx                   ; restore rbx before returning
        ret
format:
        db  "%20ld", 10, 0
```

$ nasm -felf64 fib.asm && gcc fib.o && ./a.out

```
                   0
                   1
                   1
                   2
                   .
                   .
                   .
 679891637638612258
```

```
1100087778366101931
1779979416004714189
```

8.2. Example 1 : write a very simple procedure named *sum* that adds the variables stored in the ECX and EDX register and returns the sum in the EAX register

- **Nhập nội dung tập tin sau**

```
section .text
   global _start          ;must be declared for using gcc
_start:                   ;tell linker entry point
   mov  ecx,'4'
   sub     ecx, '0'
   mov  edx, '5'
   sub     edx, '0'
   call    sum            ;call sum procedure
   mov  [res], eax
   mov  ecx, msg
   mov  edx, len
   mov  ebx,1          ;file descriptor (stdout)
   mov  eax,4          ;system call number (sys_write)
   int  0x80           ;call kernel
   mov  ecx, res
   mov  edx, 1
   mov  ebx, 1         ;file descriptor (stdout)
   mov  eax, 4         ;system call number (sys_write)
   int  0x80           ;call kernel
   mov  eax,1          ;system call number (sys_exit)
   int  0x80           ;call kernel
sum:
   mov     eax, ecx
```

```
    add     eax, edx
    add     eax, '0'
    ret
section .data
msg db "The sum is:", 0xA,0xD
len equ $- msg
segment .bss
res resb 1
```

-

8.3. Example 2 (Macro)

- Following example shows defining and using macros

```
; A macro with two parameters
; Implements the write system call
   %macro write_string 2
      mov   eax, 4
      mov   ebx, 1
      mov   ecx, %1
      mov   edx, %2
      int   80h
   %endmacro

section   .text
   global _start            ;must be declared for using gcc

_start:                     ;tell linker entry point
   write_string msg1, len1
   write_string msg2, len2
   write_string msg3, len3

   mov eax,1                ;system call number (sys_exit)
   int 0x80                 ;call kernel

section   .data
msg1 db   'Hello, programmers!',0xA,0xD
len1 equ $ - msg1

msg2 db 'Welcome to the world of,', 0xA,0xD
```

```
len2 equ $- msg2

msg3 db 'Linux assembly programming! '
len3 equ $- msg3
```

-

8.4. Mixing C and assembly language

```
; ----------------------------------------------------------------
; A 64-bit function that returns the maximum value of its three
64-bit integer
; arguments.  The function has signature:
;
;   int64_t maxofthree(int64_t x, int64_t y, int64_t z)
;
; Note that the parameters have already been passed in rdi,
rsi, and rdx.  We
; just have to return the value in rax.
; ----------------------------------------------------------------
        global  maxofthree
        section .text
maxofthree:
        mov     rax, rdi        ; result (rax) initially holds x
        cmp     rax, rsi        ; is x less than y?
        cmovl   rax, rsi        ; if so, set result to y
        cmp     rax, rdx        ; is max(x,y) less than z?
        cmovl   rax, rdx        ; if so, set result to z
        ret                     ; the max will be in rax
```

- Here is a C program that calls the assembly language function

```
/*
 * A small program that illustrates how to call the
maxofthree function we wrote in
 * assembly language. */
#include <stdio.h>
#include <inttypes.h>
int64_t maxofthree(int64_t, int64_t, int64_t);
int main() {
    printf("%ld\n", maxofthree(1, -4, -7));
    printf("%ld\n", maxofthree(2, -6, 1));
    printf("%ld\n", maxofthree(2, 3, 1));
    printf("%ld\n", maxofthree(-2, 4, 3));
    printf("%ld\n", maxofthree(2, -6, 5));
    printf("%ld\n", maxofthree(2, 4, 6));
```

```
        return 0;
    }
```

- *$ nasm -felf64 maxofthree.asm && gcc callmaxofthree.c maxofthree.o && ./a.out*
- *1*
- *2*
- *3*
- *4*
- *5*
- *6*
-

8.5. Local Variables and Stack Frames

When a function is called the caller will first put the parameters in the correct registers then issue the callinstruction. Additional parameters beyond those covered by the registers will be pushed on the stack prior to the call. The call instruction puts the return address on the top of stack. So if you have the function

```
int64_t example(int64_t x, int64_t y) {
    int64_t a, b, c;
    b = 7;
    return x * b + y;
}
```

Then on entry to the function, x will be in edi, y will be in esi, and the return address will be on the top of the stack. Where can we put the local variables? An easy choice is on the stack itself, though if you have enough regsters, use those.

If you are running on a machine that respect the standard ABI, you can leave rsp where it is and access the "extra parameters" and the local variables directly from rsp for example:

```
          +----------+

  rsp-24  |    a     |

          +----------+

  rsp-16  |    b     |

          +----------+

  rsp-8   |    c     |

          +----------+

  rsp     | retaddr  |
```

```
          +----------+
rsp+8     | caller's |
          | stack    |
          | frame    |
          | ...      |
          +----------+
```

So our function looks like this:

```
        global  example
        section .text
example:
        mov     qword [rsp-16], 7
        mov     rax, rdi
        imul    rax, [rsp+8]
        add     rax, rsi
        ret
```

If our function were to make another call, you would have to adjust rsp to get out of the way at that time.

On Windows you can’t use this scheme because if an interrupt were to occur, everything above the stack pointer gets plastered. This doesn’t happen on most other operating systems because there is a "red zone" of 128 bytes past the stack pointer which is safe from these things. In this case, you can make room on the stack immediately:

```
example:
        sub rsp, 24
```

so our stack looks like this:

```
          +----------+
rsp       |    a     |
          +----------+
rsp+8     |    b     |
          +----------+
rsp+16    |    c     |
```

```
              +----------+
       rsp+24 | retaddr  |
              +----------+
       rsp+32 | caller's |
              | stack    |
              | frame    |
              | ...      |
              +----------+
```

Here's the function now. Note that we have to remember to replace the stack pointer before returning!

```
        global  example
        section .text
example:
        sub     rsp, 24
        mov     qword [rsp+8], 7
        mov     rax, rdi
        imul    rax, [rsp+8]
        add     rax, rsi
        add     rsp, 24
        ret
```

8.6. Example library macro and procedure

intarith.asm simple integer arithmetic

The nasm source code is intarith.asm
The result of the assembly is intarith.lst
The equivalent "C" program is intarith.c
Running the program produces output intarith.out

This program demonstrates basic integer arithmetic add, subtract, multiply and divide.
The equivalent "C" code is shown as comments in the assembly language.

```
; intarith.asm show some simple C code and corresponding nasm code
; the nasm code is one sample, not unique
;
; compile:   nasm -f elf -l intarith.lst  intarith.asm
```

```
; link:            gcc -o intarith  intarith.o
; run:       intarith
;
; the output from running intarith.asm and intarith.c is:
; c=5  , a=3, b=4, c=5
; c=a+b, a=3, b=4, c=7
; c=a-b, a=3, b=4, c=-1
; c=a*b, a=3, b=4, c=12
; c=c/a, a=3, b=4, c=4
;
;The file  intarith.c  is:
;  /* intarith.c */
;  #include
;  int main()
;  {
;    int a=3, b=4, c;
;
;    c=5;
;    printf("%s, a=%d, b=%d, c=%d\n","c=5  ", a, b, c);
;    c=a+b;
;    printf("%s, a=%d, b=%d, c=%d\n","c=a+b", a, b, c);
;    c=a-b;
;    printf("%s, a=%d, b=%d, c=%d\n","c=a-b", a, b, c);
;    c=a*b;
;    printf("%s, a=%d, b=%d, c=%d\n","c=a*b", a, b, c);
;    c=c/a;
;    printf("%s, a=%d, b=%d, c=%d\n","c=c/a", a, b, c);
;    return 0;
; }

    extern printf        ; the C function to be called

%macro pabc 1             ; a "simple" print macro
      section .data
.str   db    %1,0         ; %1 is first actual in macro call
      section .text
                          ; push onto stack backwards
      push   dword [c]    ; int c
      push   dword [b]    ; int b
      push   dword [a]    ; int a
      push   dword .str   ; users string
      push    dword fmt       ; address of format string
      call    printf          ; Call C function
      add     esp,20          ; pop stack 5*4 bytes
%endmacro

section .data             ; preset constants, writeable
```

```
a:     dd     3             ; 32-bit variable a initialized to 3
b:     dd     4             ; 32-bit variable b initializes to 4
fmt:    db "%s, a=%d, b=%d, c=%d",10,0 ; format string for printf

section .bss                ; unitialized space
c:     resd   1             ; reserve a 32-bit word

section .text               ; instructions, code segment
global  main        ; for gcc standard linking
main:                       ; label

lit5:                       ; c=5;
      mov    eax,5          ; 5 is a literal constant
      mov    [c],eax              ; store into c
      pabc   "c=5  "              ; invoke the print macro

addb:                       ; c=a+b;
      mov    eax,[a]              ; load a
      add    eax,[b]              ; add b
      mov    [c],eax              ; store into c
      pabc   "c=a+b"              ; invoke the print macro

subb:                       ; c=a-b;
      mov    eax,[a]              ; load a
      sub    eax,[b]              ; subtract b
      mov    [c],eax              ; store into c
      pabc   "c=a-b"              ; invoke the print macro

mulb:                       ; c=a*b;
      mov    eax,[a]        ; load a (must be eax for multiply)
      imul   dword [b]      ; signed integer multiply by b
      mov    [c],eax        ; store bottom half of product into c
      pabc   "c=a*b"              ; invoke the print macro

diva:                       ; c=c/a;
      mov    eax,[c]        ; load c
      mov    edx,0          ; load upper half of dividend with zero
      idiv   dword [a]      ; divide double register edx eax by a
      mov    [c],eax              ; store quotient into c
      pabc   "c=c/a"              ; invoke the print macro
     mov      eax,0             ; exit code, 0=normal
      ret                   ; main return to operating system
```

8.7. Example mix macro and procedure

fltarith.asm simple floating point arithmetic

The nasm source code is [fltarith.asm]

The result of the assembly is [fltarith.lst]

The equivalent "C" program is [fltarith.c]

Running the program produces output [fltarith.out]

This program demonstrates basic floating point add, subtract, multiply and divide.

The equivalent "C" code is shown as comments in the assembly language.

```
; fltarith.asm   show some simple C code and corresponding nasm
code the nasm code is one sample, not unique
;
; compile  nasm -f elf -l fltarith.lst  fltarith.asm
; link     gcc -o fltarith  fltarith.o
; run      fltarith
;
; the output from running fltarith and fltarithc is:
; c=5.0, a=3.000000e+00, b=4.000000e+00, c=5.000000e+00
; c=a+b, a=3.000000e+00, b=4.000000e+00, c=7.000000e+00
; c=a-b, a=3.000000e+00, b=4.000000e+00, c=-1.000000e+00
; c=a*b, a=3.000000e+00, b=4.000000e+00, c=1.200000e+01
; c=c/a, a=3.000000e+00, b=4.000000e+00, c=4.000000e+00
;
;The file  fltarith.c  is:
;  #include
;  int main()
;  {
;    double a=3.0, b=4.0, c;
;
;    c=5.0;
;    printf("%s, a=%e, b=%e, c=%e\n","c=5.0", a, b, c);
;    c=a+b;
;    printf("%s, a=%e, b=%e, c=%e\n","c=a+b", a, b, c);
;    c=a-b;
;    printf("%s, a=%e, b=%e, c=%e\n","c=a-b", a, b, c);
;    c=a*b;
;    printf("%s, a=%e, b=%e, c=%e\n","c=a*b", a, b, c);
;    c=c/a;
;    printf("%s, a=%e, b=%e, c=%e\n","c=c/a", a, b, c);
;    return 0;
; }

extern printf            ; the C function to be called

%macro pabc 1            ; a "simple" print macro
        section      .data
```

```
.str   db     %1,0   ; %1 is macro call first actual parameter
section .text
                        ; push onto stack backwards
       push   dword [c+4]  ; double c (bottom)
       push   dword [c]    ; double c
       push   dword [b+4]  ; double b (bottom)
       push   dword [b]    ; double b
       push   dword [a+4]  ; double a (bottom)
       push   dword [a]    ; double a
       push   dword .str   ; users string
      push    dword fmt        ; address of format string
      call    printf           ; Call C function
      add     esp,32           ; pop stack 8*4 bytes
%endmacro

section      .data               ; preset constants, writeable
a:     dq     3.333333333  ; 64-bit variable a initialized to 3.0
b:     dq     4.444444444  ; 64-bit variable b initializes to 4.0
five:  dq     5.0          ; constant 5.0
fmt:    db "%s, a=%e, b=%e, c=%e",10,0 ; format string for printf

section .bss               ; unitialized space
c:     resq   1            ; reserve a 64-bit word

section .text              ; instructions, code segment
       global main         ; for gcc standard linking
main:                      ; label

lit5:                      ; c=5.0;
       fld    qword [five] ; 5.0 constant
       fstp   qword [c]    ; store into c
       pabc   "c=5.0"      ; invoke the print macro

addb:                      ; c=a+b;
       fld    qword [a]    ; load a (pushed on flt pt stack, st0)
       fadd   qword [b]    ; floating add b (to st0)
       fstp   qword [c]    ; store into c (pop flt pt stack)
       pabc   "c=a+b"      ; invoke the print macro

subb:                      ; c=a-b;
       fld    qword [a]    ; load a (pushed on flt pt stack, st0)
       fsub   qword [b]    ; floating subtract b (to st0)
       fstp   qword [c]    ; store into c (pop flt pt stack)
       pabc   "c=a-b"          ; invoke the print macro

mulb:                      ; c=a*b;
       fld    qword [a]    ; load a (pushed on flt pt stack, st0)
```

```
        fmul   qword [b]    ; floating multiply by b (to st0)
        fstp   qword [c]; store product into c (pop flt pt stack)
        pabc   "c=a*b"      ; invoke the print macro

diva:                       ; c=c/a;
        fld    qword [c]    ; load c (pushed on flt pt stack, st0)
        fdiv   qword [a]    ; floating divide by a (to st0)
        fstp   qword [c]; store quotient into c (pop flt pt stack)
        pabc   "c=c/a"      ; invoke the print macro

       mov      eax,0           ; exit code, 0=normal
        ret                 ; main returns to operating system
```