**National University of Computer & Emerging Sciences, Karachi**

**EL-2003: Computer Organization & Assembly Language Lab**

***Advanced procedure call***

Spring 2022

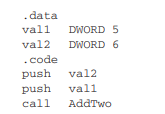
Two general types of arguments are pushed on the stack during subroutine calls:

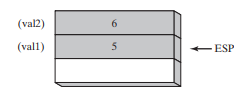
• Value arguments (values of variables and constants)

• Reference arguments (addresses of variables)

Passing by Value

When an argument is passed by value, a copy of the value is pushed on the stack. Suppose we call a subroutine named AddTwo, passing it two 32-bit integers:





*An equivalent function call written in C++ would be*

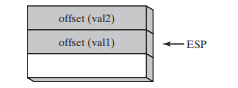
*AddTwo( val1, val2 );*

Observe that the arguments are pushed on the stack in reverse order, which is the norm for the C and C++ languages.

Passing by Reference

An argument passed by reference consists of the address (offset) of an object. The following statements call Swap, passing the two arguments by reference:





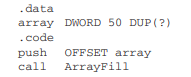
*The equivalent function call in C/C++ would pass the addresses of the val1 and val2 arguments:*

*Swap( &val1, &val2 );*

Passing Arrays

* High-level languages always pass arrays to subroutines by reference. That is, they push the address of an array on the stack. The subroutine can then get the address from the stack and use it to access the array.
* It’s easy to see why one would not want to pass an array by value, because doing so would require each array element to be pushed on the stack separately. Such an operation would be very slow and it would use up precious stack space.

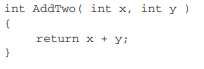
The following statements do it the right way by passing the offset of array to a subroutine named ***ArrayFill:***



Accessing Stack Parameters

AddTwo Example in C

The following **AddTwo function, written in C**, receives two integers passed by value and returns their sum:



Let’s create an **equivalent implementation in assembly language**. AddTwo pushes EBP on the stack to preserve its existing value:

AddTwo PROC

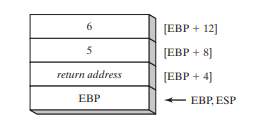
push ebp

Next, EBP is set to the same value as ESP, so EBP can be the base pointer for AddTwo’s stack frame: AddTwo PROC

push ebp

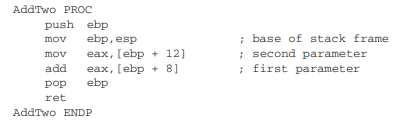
mov ebp,esp

After the two instructions execute, the following figure shows the contents of the stack frame. A function call such as AddTwo(5, 6) would cause the second parameter to be pushed on the stack, followed by the first parameter:



Base-Offset Addressing

We will use base-offset addressing to access stack parameters. EBP is the base register and the offset is a constant. 32-bit values are usually returned in EAX. The following implementation of AddTwo adds the parameters and returns their sum in EAX:

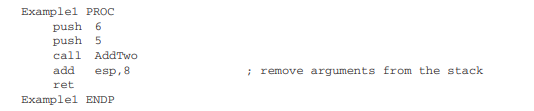


Cleaning Up the Stack

There must be a way for parameters to be removed from the stack when a subroutine returns. Otherwise, a memory leak would result, and the stack would become corrupted.

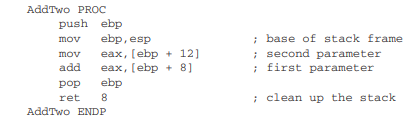
The C Calling Convention

A simple way to remove parameters from the runtime stack is to add a value to ESP equal to the combined sizes of the parameters. Then, ESP will point to the stack location that contains the subroutine’s return address. Using the current code example, we can follow the CALL with an ADD:



STDCALL Calling Convention

Another common way to remove parameters from the stack is to use a convention named STDCALL. In the following AddTwo procedure, we supply an integer parameter to the RET instruction, which in turn adds 8 to EBP after returning to the calling procedure. The integer must equal the number of bytes of stack space consumed by the subroutine parameters:



Passing 16-Bit Arguments on the Stack

* When passing stack arguments to procedures in protected mode, it’s best to push 32-bit operands.
* Though you can push 16-bit operands on the stack, doing so prevents ESP from being aligned on a doubleword boundary.
* A page fault may occur and runtime performance may be degraded. You should expand them to 32 bits before pushing them on the stack.

16-Bit Argument Example

Suppose we want to pass two 16-bit integers to the AddTwo procedure shown earlier. The procedure expects 32-bit values, so the following call would cause an error:

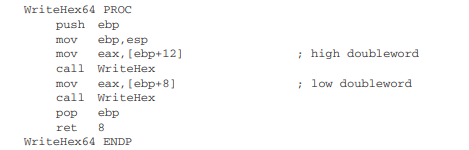


Instead, we can zero-extend each argument before pushing it on the stack. The following code correctly calls AddTwo:

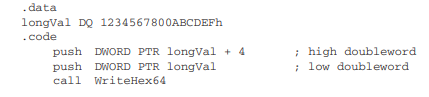


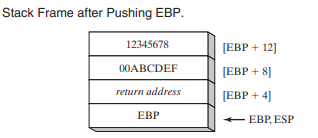
Passing Multiword Arguments

The following WriteHex64 procedure receives a 64-bit integer on the stack and displays it in hexadecimal:



The call to WriteHex64 pushes the upper half of longVal, followed by the lower half:





Local Variables

Example The following C++ function declares local variables X and Y:

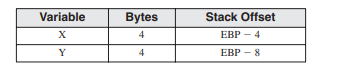
void MySub() {

int X = 10;

int Y = 20;

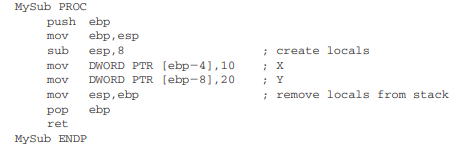
}

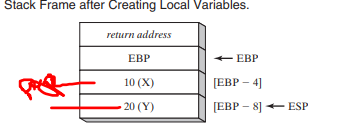
*We can use the compiled C++ program as a guide, showing how local variables are allocated by the C++ compiler. Each stack entry defaults to 32 bits, so each variable’s storage size is rounded upward to a multiple of 4. A total of 8 bytes is reserved for the two local variables:*



***Local Var:DWORD***

The following assembly of the MySub function shows how a C++ program creates local variables, assigns values, and removes the variables from the stack. It uses the C calling convention:





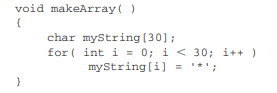
LEA Instruction

The LEA instruction returns the effective address of an indirect operand, memory variable etc.

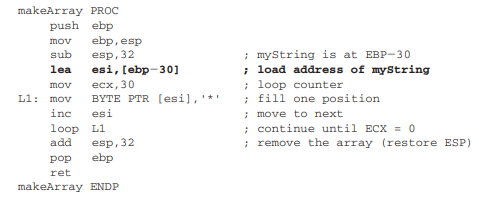
The difference between **mov** and **lea** instruction is that **mov** instruction moves the content of source to destination while **lea** instruction moves the address of source to destination. Its syntax is:

**Lea destination, source** ; returns the address of source

To show how LEA can be used, let’s look at the following C++ program, which declares a local array of char and references myString when assigning values:



* The equivalent code in assembly language allocates space for myString on the stack and assigns the address to ESI, an indirect operand.
* the array is only 30 bytes, ESP is decremented by 32 to keep it aligned on a doubleword boundary



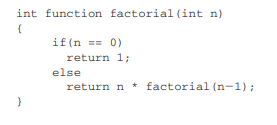
It is not possible to use OFFSET to get the address of a stack parameter because OFFSET only works with addresses known at compile time. The following statement would not assemble:

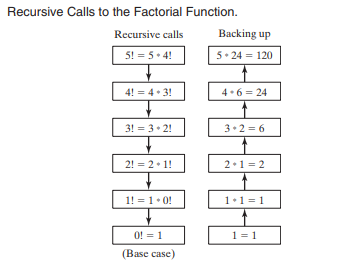
mov esi,OFFSET [ebp-30] ; error

Recursion

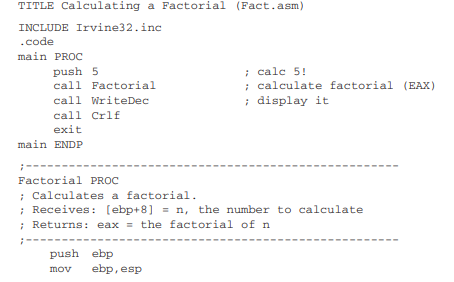
A recursive subroutine is one that calls itself, either directly or indirectly.

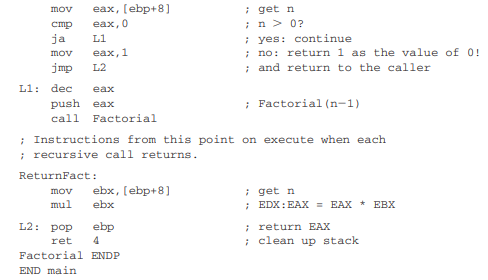
Calculating a Factorial





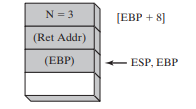
Example program in Assembly language



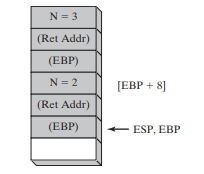


Lets examine n=3

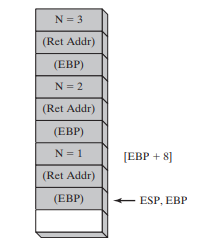
mov eax,[ebp+8] ; get n



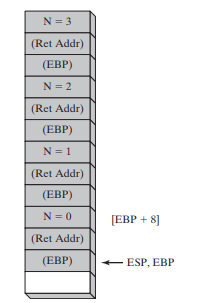
The runtime stack now holds a second stack frame, with N equal to 2:



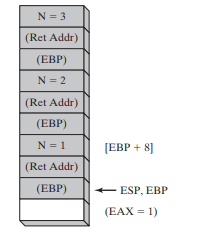
Now, entering Factorial a third time, three stack frames are active:



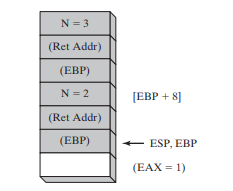
The Factorial procedure compares N to 0, and on finding that N is greater than zero, calls Factorial one more time with N = 0. The runtime stack now contains its fourth stack frame as it enters the Factorial procedure for the last time



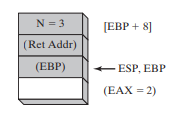
When Factorial is called with N = 0, things get interesting. The following statements cause a branch to label L2. The value 1 is assigned to EAX because 0! = 1, and EAX must be assigned Factorial’s return value:



As the RET statement executes, another frame is removed from the stack:



With EAX now equal to 2, the RET statement removes another frame from the stack:



INVOKE Directive

* The INVOKE directive pushes arguments on the stack and calls a procedure.
* INVOKE is a convenient replacement for the CALL instruction because it lets you pass multiple arguments using a single line of code.

SYNTAX

INVOKE procedureName [, argumentList]

ArgumentList is an optional comma-delimited list of arguments passed to the procedure.

Using the CALL instruction, for example, we could call a procedure named DumpArray after executing several PUSH instructions:



The equivalent statement using INVOKE is reduced to a single line in which the arguments are listed in reverse order.

*INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array*

INVOKE permits almost any number of arguments, and individual arguments can appear on separate source code lines. The following INVOKE statement includes helpful comments:

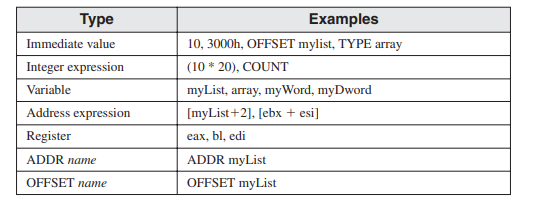
INVOKE DumpArray, ; displays an array

OFFSET array, ; points to the array

LENGTHOF array, ; the array length

TYPE array ; array component size

Argument type used in invoke:



ADDR Operator

The ADDR operator can be used to pass a pointer argument when calling a procedure using INVOKE. The following INVOKE statement,

**for example,** passes the address of myArray to the FillArray procedure: INVOKE FillArray, ADDR myArray

Example

The following INVOKE directive calls Swap, passing it the addresses of the first two elements in an array of doublewords:

.data Array DWORD 20 DUP(?)

.code

...

INVOKE Swap,

ADDR Array,

ADDR [Array+4]

Here is the corresponding code generated by the assembler,

Push OFFSET Array+4

push OFFSET Array

call Swap

PROTO Directive

The PROTO directive creates a prototype for an existing procedure. A prototype declares a procedure’s name and parameter list. It allows you to call a procedure before defining it and to verify that the number and types of arguments match the procedure definition.

MySub PROTO ; procedure prototype

.

INVOKE MySub ; procedure call

.

MySub PROC ; procedure implementation

.

.

MySub ENDP