

Passing 8-Bit and 16-Bit Arguments on the Stack

Passing 8-bit Arguments

When passing stack arguments to procedures in 32-bit mode, it is best to push 32-bit operands.

This is because the stack pointer (ESP) must be aligned on a doubleword boundary.

If 16-bit operands are pushed onto the stack, ESP will not be aligned and a page fault may occur. Additionally, runtime performance may be degraded.

If you need to pass a 16-bit operand to a procedure in 32-bit mode, you can use the MOVZX instruction to expand the operand to 32 bits before pushing it onto the stack.

For example, the following Uppercase procedure receives a character argument and returns its uppercase equivalent in AL:

```

1645 Uppercase PROC
1646 push ebp
1647 mov ebp,esp
1648 mov al,[esp+8]
1649 ; AL = character
1650 cmp al,'a'
1651 ; less than 'a'?
1652 jb L1
1653 ; yes: do nothing
1654 cmp al,'z'
1655 ; greater than 'z'?
1656 ja L1
1657 ; yes: do nothing
1658 sub al,32
1659 ; no: convert it
1660 L1:
1661 pop ebp
1662 ret 4
1663 ; clean up the stack
1664 Uppercase ENDP

```

If we pass a character literal to Uppercase, the PUSH instruction will automatically expand the character to 32 bits:

```

1667 push 'x'
1668 call Uppercase

```

However, if we pass a character variable to Uppercase, the PUSH instruction will not allow us to push an 8-bit operand onto the stack.

To work around this, we can use the MOVZX instruction to expand the character into EAX before pushing it onto the stack:

```
1672 .data
1673     charVal BYTE 'x'
1674 .code
1675     movzx eax,charVal
1676     ; move with extension
1677     push eax
1678     call Uppercase
```

This will ensure that ESP is aligned on a doubleword boundary and that the call to Uppercase is successful.

Passing 16-bit Arguments.

The AddTwo procedure expects two 32-bit integer arguments (the two integers to be added). However, the word1 and word2 variables are 16-bit integers.

Therefore, if we push word1 and word2 onto the stack and call AddTwo, the procedure will not be able to correctly add the two integers.

To fix this, we can zero-extend each argument before pushing it onto the stack. Zero-extension means that the high-order 16 bits of the argument are set to zero.

This will effectively convert the 16-bit argument to a 32-bit argument.

The following code correctly calls AddTwo by zero-extending each argument before pushing it onto the stack:

```

1686 .data
1687     word1 WORD 1234h
1688     word2 WORD 4111h
1689 .code
1690     movzx eax,word1
1691     push eax
1692     movzx eax,word2
1693     push eax
1694     call AddTwo
1695     ; sum is in EAX

```

The **MOVZX instruction** is used to zero-extend the 16-bit word1 and word2 variables into the 32-bit EAX register.

Once the arguments have been zero-extended, they are pushed onto the stack in reverse order (word2 first, then word1).

When AddTwo is called, it will pop the two arguments off the stack and add them together. The sum of the two integers will be returned in the EAX register.

It is important to note that the caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

In the case of stack parameters, the order and size of the parameters are important.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

Passing 64-bit Arguments

To pass 64-bit integer arguments to procedures in 32-bit mode, we must push the high-order doubleword of the argument first, followed by the low-order doubleword.

This is because the stack grows downwards, so the lower order doubleword of the argument will be at the lower address on the stack.

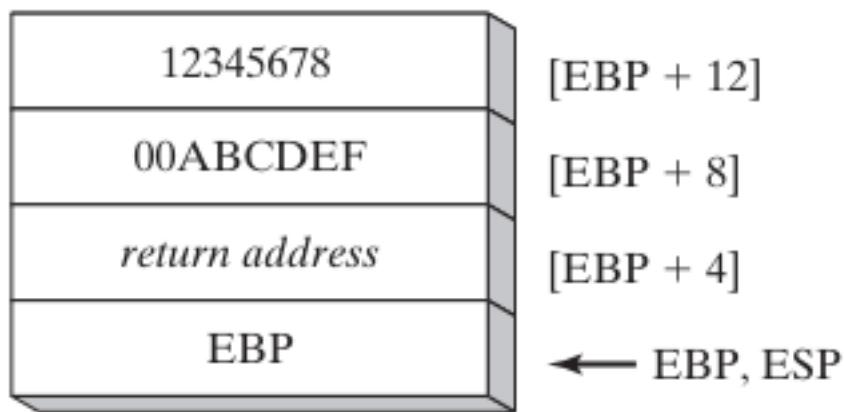
The following WriteHex64 procedure displays a 64-bit integer in hexadecimal:

```
1700 WriteHex64 PROC
1701     push ebp
1702     mov ebp,esp
1703     mov eax,[ebp+12]
1704     ; high doubleword
1705     call WriteHex
1706     mov eax,[ebp+8]
1707     ; low doubleword
1708     call WriteHex
1709     pop ebp
1710     ret 8
1711 WriteHex64 ENDP
```

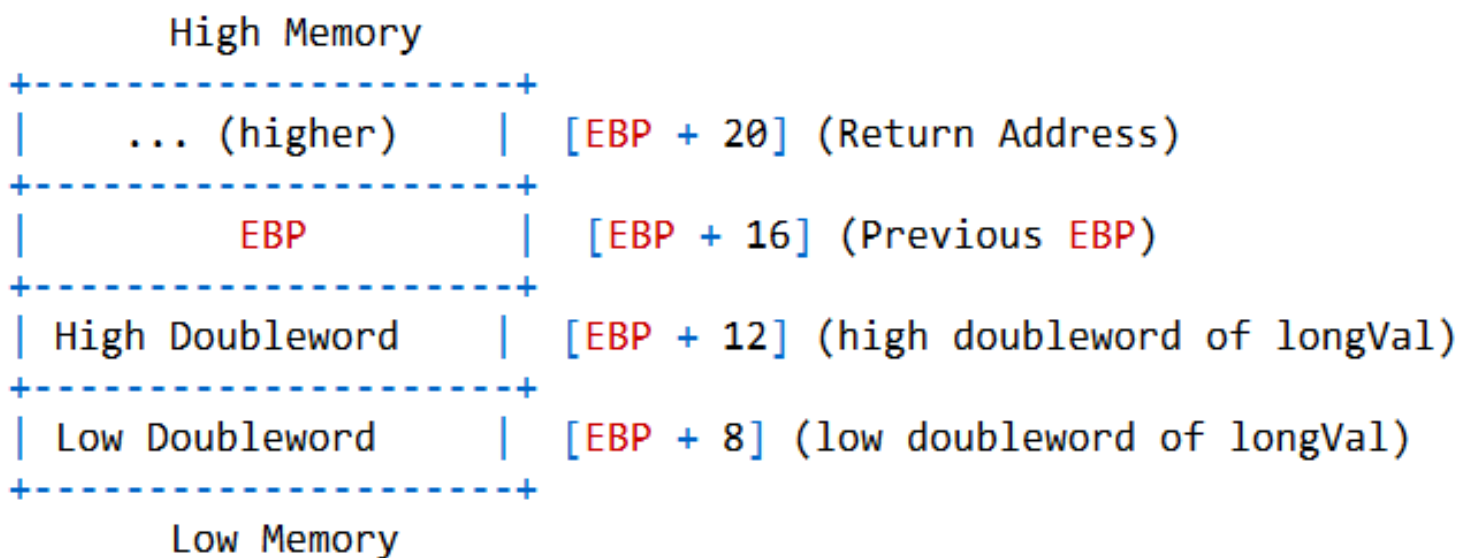
The following sample call to WriteHex64 pushes the upper half of longVal, followed by the lower half:

```
1715 .data
1716     longVal QWORD 1234567800ABCDEFh
1717 .code
1718     push DWORD PTR longVal + 4
1719     ; high doubleword
1720     push DWORD PTR longVal
1721     ; low doubleword
1722     call WriteHex64
```

Figure below shows the stack frame inside WriteHex64 just after EBP was pushed on the stack and ESP was copied to EBP:



Or



The WriteHex64 procedure can then easily retrieve the high and low doublewords of the argument from the stack and display them in hexadecimal.

It is important to note that the caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

In the case of stack parameters, the order and size of the parameters are important.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

Here is a more in-depth explanation of why we must push the high-

order doubleword of a 64-bit integer first when passing it to a procedure in 32-bit mode:

In 32-bit mode, the stack grows downwards. This means that when we push a value onto the stack, the stack pointer (ESP) is decremented.

When we pop a value off the stack, ESP is incremented.

When we pass a 64-bit integer to a procedure in 32-bit mode, we must push the high-order doubleword of the integer first, followed by the low-order doubleword.

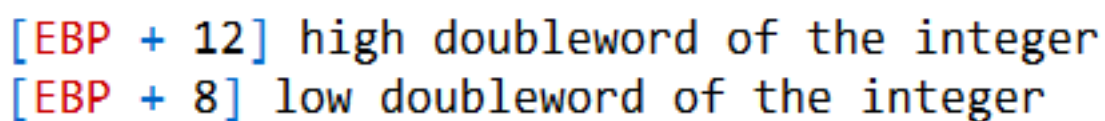
This is because we want the integer to be stored on the stack in little-endian order.

In **little-endian order**, the low-order byte of the integer is stored at the lowest address on the stack.

If we were to push the low-order doubleword of the integer first, followed by the high-order doubleword, the integer would be stored on the stack in big-endian order.

In **big-endian order**, the high-order byte of the integer is stored at the lowest address on the stack.

The following diagram shows how a 64-bit integer is stored on the stack in little-endian order:



[EBP + 12] high doubleword of the integer
[EBP + 8] low doubleword of the integer

The WriteHex64 procedure can then easily retrieve the high and low doublewords of the integer from the stack and display them in hexadecimal.

Why is it important to ensure that the arguments passed to a procedure are consistent with the parameters expected by the procedure?

The caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

This is because the procedure is expecting certain values to be passed to it in a certain order.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

For example, if the WriteHex64 procedure expects one 64-bit integer argument, and the caller passes two 64-bit integer arguments, the procedure will not be able to correctly display the two integers.

Or, if the caller passes a 32-bit integer argument instead of a 64-bit integer argument, the procedure will also not be able to correctly display the integer.

It is important to note that the compiler will not check to make sure that the caller is passing the correct number of arguments to a procedure, or that the arguments are in the correct order or have the wrong size. This is the responsibility of the programmer.

Non-Doubleword Local Variables

In 32-bit mode, the stack grows downwards. This means that when we push a value onto the stack, the stack pointer (ESP) is decremented.

When we pop a value off the stack, ESP is incremented.

When we declare a local variable in a procedure, MASM will allocate space for it on the stack. The size of the space allocated will depend on the size of the variable.

For example, if we declare a byte variable, MASM will allocate one byte of space on the stack. If we declare a doubleword variable, MASM will allocate four bytes of space on the stack.

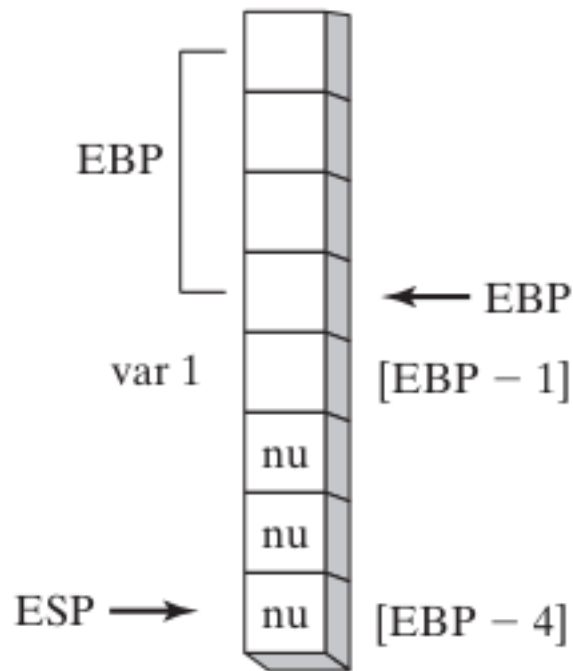
If we declare a local variable of a size that is not a multiple of four bytes (such as a byte or a word), MASM will round the size of the variable up to the next multiple of four bytes.

This is because the stack is aligned on a doubleword boundary. This means that all addresses on the stack must be divisible by four.

For example, if we declare a byte variable named `var1` in the `Example1` procedure, MASM will allocate four bytes of space for it on the stack, even though the variable is only one byte in size. The remaining three bytes will be unused.

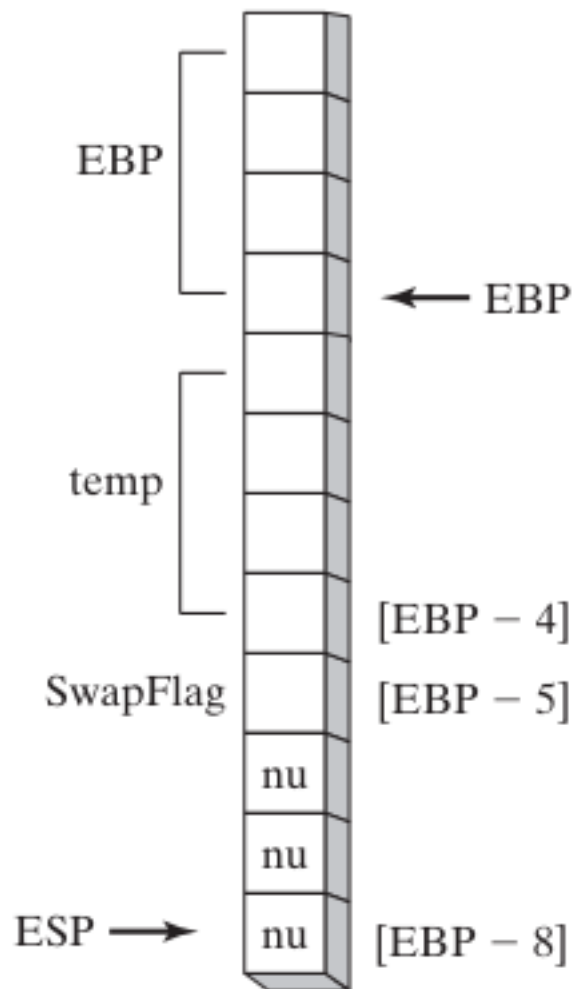
The following diagram shows how the stack looks after the `Example1` procedure has been compiled and assembled:

Creating space for local variables (*Example1* Procedure).



The `nu` blocks represent unused bytes. The following diagram shows how the stack looks after the `Example2` procedure has been compiled and assembled:

Creating space in Example 2 for local variables.



The temp variable is a doubleword variable, so it is aligned on a doubleword boundary.

The SwapFlag variable is a byte variable, but it is still allocated four bytes of space on the stack because the stack is aligned on a doubleword boundary.

Stack size for nested procedure calls

The stack size required for nested procedure calls is the sum of the stack sizes required for each individual procedure call.

This is because the stack is used to store the local variables and return addresses for all active procedure calls.

For example, in the following code:

```
1750 Sub1 PROC
1751 local array1[50]:dword
1752 ; 200 bytes
1753 callSub2
1754 .
1755 .
1756 ret
1757 Sub1 ENDP
1758 Sub2 PROC
1759 local array2[80]:word
1760 ; 160 bytes
1761 callSub3
1762 .
1763 .
1764 ret
1765 Sub2 ENDP
1766 Sub3 PROC
1767 local array3[300]:dword
1768 ; 1200 bytes
1769 .
1770 .
1771 ret
1772 Sub3 ENDP
```

The stack size required for Sub1 is 200 bytes, the stack size required for Sub2 is 160 bytes, and the stack size required for Sub3 is 1200 bytes. Therefore, the total stack size required for this code is 1560 bytes.

This stack size is the minimum amount of stack space that must be available in order for this code to execute correctly. If there is not enough stack space available, the program will crash.

Recursive procedure calls

If a procedure is **called recursively**, the stack space it uses will be approximately the size of its local variables and parameters multiplied by the estimated depth of the recursion.

For example, if a procedure has 100 bytes of local variables and parameters, and it is called recursively to a depth of 10, then the procedure will use approximately 1000 bytes of stack space.

Stack overflow

If the stack space required for a program exceeds the amount of stack space available, the program will crash. This is called a **stack overflow**.

To avoid stack overflows, it is important to be aware of the **stack space requirements of your program**. You can use the **STACK directive** to reserve additional stack space if necessary.

The stack size required for nested procedure calls is the sum of the stack sizes required for each individual procedure call.

If a procedure is called recursively, the stack space it uses will be approximately the size of its local variables and parameters multiplied by the estimated depth of the recursion.

To avoid stack overflows, it is important to be aware of the stack space requirements of your program and to reserve additional stack space, if necessary.

Here is a summary of the key points from the chapter:

- There are two types of procedure parameters: register parameters (faster, used by Irvine libraries) and stack parameters (more flexible).
- A stack frame is the region of stack used by a procedure for its parameters, local variables, saved registers, and return address.
- Parameters can be passed by value (copied) or by reference (address passed). Arrays should be passed by reference.
- Stack parameters are accessed using EBP offset addressing like [EBP-8]. LEA is good for getting stack parameter addresses.
- ENTER/LEAVE instructions manage the stack frame set up/teardown.

- Recursive procedures call themselves directly or indirectly. Recursion works well with repeating data structures.
- Local variables have restricted scope, lifetime tied to the procedure, don't cause naming clashes, and enable recursion.
- INVOKE directive calls procedures with multiple arguments. ADDR passes pointers.
- PROC declares procedures, PROTO prototypes existing procedures.
- Large programs should be split into multiple source code modules for manageability.
- Java bytecode is the machine language in compiled Java programs. The JVM executes it. Bytecodes use a stack-oriented model.