Copying Strings

Using a loop that copies a string, represented as an array of bytes with a null terminator value.

Indexed addressing works well for this type of operation because the same index register references both strings.

The target string must have enough available space to receive the copied characters, including the null byte at the end:

```
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO ,dwExitCode:DWORD

.data
    source  BYTE    "This is the source string",0
    target  BYTE    SIZEOF  source DUP(0)

.code
main PROC
    mov esi, 0                  ;index register
    mov ecx, SIZEOF source      ;loop counter

L1:
    mov al, source[esi]         ;get a character from source
    mov target[esi], al         ;store it in the target
    inc esi                     ;move to the next character

invoke ExitProcess, 0
main ENDP
END main
```

When you move a 32-bit constant to a 64-bit register, the upper 32 bits (bits 32–63) of the destination are cleared (equal to zero):

```
mov rax,0FFFFFFFFh      ; rax = 00000000FFFFFFFF
```

When you move a 16-bit constant or an 8-bit constant into a 64-bit register, the upper bits are also cleared:

```
mov rax,06666h          ; clears bits 16-63
mov rax,055h            ; clears bits 8-63
```

When you move memory operands into 64-bit registers, however, the results are mixed. For example, moving a 32-bit memory operand into EAX (the lower half of RAX) causes the upper 32 bits in RAX to be cleared:

```
.data
    myDword DWORD   80000000h
.code
    mov rax, 0FFFFFFFFFFFFFFFFh
    mov eax, myDword                    ;RAX = 0000000080000000
```

But when you move an 8-bit or a 16-bit memory operand into the lower bits of RAX, the highest bits in the destination register are not affected:

The following assembly code example shows how moving an 8-bit or 16-bit memory operand into the lower bits of RAX does not affect the highest bits in the destination register:

```
.data
    myByte BYTE 55h
    myWord WORD 6666h

.code
    ; Move the 16-bit value from myWord into the lower bits of RAX.
    mov ax, myWord

    ; Display the value of RAX in hexadecimal format.
    ; Output: 0066 (only the lower 16 bits are displayed)
    mov edx, 1 ; standard output
    mov eax, RAX
    syscall

    ; Move the 8-bit value from myByte into the lower bits of RAX.
    mov al, myByte

    ; Display the value of RAX in hexadecimal format.
    ; Output: 55 (only the lower 8 bits are displayed)
    mov edx, 1 ; standard output
    mov eax, RAX
    syscall
```

This code is performing the following operations:

Data Section, It defines two variables: myByte as a single-byte variable (BYTE) initialized with the value 55h (hexadecimal 55). myWord as a two-byte variable (WORD) initialized with the value 6666h (hexadecimal 6666). It begins the code section where the actual operations are performed.

Moving myWord into RAX: It moves the 16-bit value stored in myWord into the lower 16 bits of the RAX register using mov ax, myWord. This operation copies the value 6666h into the lower 16 bits of RAX.

Displaying RAX in Hexadecimal Format: It prepares for displaying the value of RAX in hexadecimal format. mov edx, 1 sets up the standard output file descriptor for syscall (assuming a Unix-like environment).

mov eax, RAX copies the value from RAX to EAX, ensuring that the upper 32 bits are zeros. syscall is used to display the value in EAX, which contains 6666h. The output is "0066" in hexadecimal format

because only the lower 16 bits are displayed.

Moving myByte into RAX: It moves the 8-bit value stored in myByte into the lower 8 bits of the RAX register using mov al, myByte. This operation copies the value 55h into the lower 8 bits of RAX.

Displaying RAX in Hexadecimal Format Again: It repeats the preparation for displaying the value of RAX in hexadecimal format. mov eax, RAX copies the value from RAX to EAX, ensuring that the upper 32 bits are zeros.

syscall is used to display the value in EAX, which contains 55h. The output is "55" in hexadecimal format because only the lower 8 bits are displayed.

In summary, this code demonstrates how to move 16-bit and 8-bit values from memory into the RAX register and then display the lower bits of RAX in hexadecimal format using system calls.

The **MOVSXD instruction (move with sign-extension)** permits the source operand to be a 32-bit register or memory operand. The following instructions cause RAX to equal FFFFFFFFFFFFFFFFh:

```
mov ebx,0FFFFFFFFh
movsxd rax,ebx
```

The code is moving the 16-bit value from the variable myWord into the lower 16 bits of the RAX register. RAX is a 64-bit register, so the highest bits of the register are not affected.

In other words, the code is copying the lower 16 bits of myWord into RAX, and the upper 48 bits of RAX remain unchanged.

This is done by using the MOV instruction in assembly language. The MOV instruction copies the specified number of bits from the source operand to the destination operand.

In this case, the source operand is myWord, which is a 16-bit variable, and the destination operand is RAX, which is a 64-bit register. Therefore, only the lower 16 bits of myWord are copied into RAX.

This operation is often used to prepare data for further processing. For example, the lower 16 bits of a 32-bit value may be copied into RAX so that it can be used in a mathematical operation. Or, the lower 16 bits of a 64-bit value may be copied into RAX so that it can be stored in a 32-bit register.

The MOVSXD instruction in assembly language is used to move a signed doubleword (32 bits) to a signed quadword (64 bits) with sign extension. This means that if the sign bit of the source operand is set, then the upper 32 bits of the destination operand are also set.

In the image you sent, the MOVSXD instruction is being used to move the 32-bit value from the EBX register to the RAX register. RAX is a 64-bit register, so the upper 32 bits of the register will be set to the sign bit of EBX.

This operation is often used to prepare data for further processing. For example, a 32-bit value may be sign extended to a 64-bit value so that it can be used in a mathematical operation that expects a 64-bit operand. Or, a 32-bit value may be sign extended to a 64-bit value so that it can be stored in a 64-bit register.

In the specific example of the code you sent, the purpose of the operation is not clear. It is possible that the code is preparing the value of EBX for further processing, or it is possible that the code is simply copying the value of EBX to another location in memory.

Here is an example of how the MOVSXD instruction can be used to prepare data for further processing:

```
; EBX contains a 32-bit signed value.
; RAX is a 64-bit register.

movsxd rax, ebx ; Sign extend the 32-bit value in EBX to 64 bits.

; RAX now contains a 64-bit signed value.
```

This code can then be used to perform a mathematical operation on the value in RAX, such as:

```
; Add 10 to the value in RAX.
add rax, 10
```

The **ADD instruction** expects a 64-bit operand, so the sign extended value in RAX is used in the operation.

The **MOVSXD instruction** is a powerful tool that can be used to prepare data for further processing in a variety of ways.

The OFFSET operator generates a 64-bit address, which must be held by a 64-bit register or variable. In the following example, we use the RSI register:

```
.data
    myArray WORD 10,20,30,40
.code
    mov rsi,OFFSET myArray
```

In the data section, you define an array named myArray of type WORD (16-bit words) and initialize it with four values: 10, 20, 30, and 40. Code Section:

The code section contains the actual assembly language instructions. Using the OFFSET Operator:

The line mov rsi, OFFSET myArray is where the action happens. OFFSET myArray is an operator that calculates the memory address (64-bit in this case) of the myArray variable.

mov rsi, OFFSET myArray moves this calculated memory address into the 64-bit RSI register.

After this instruction, the RSI register holds the memory address where the myArray variable is located in memory.

In summary, this code calculates the memory address of the myArray variable using the OFFSET operator and stores it in the 64-bit RSI register.

This can be useful when you want to work with the memory address of a variable or an array in a 64-bit environment, allowing you to manipulate data at that address or access elements of the array.

Certainly, let's break down the key concepts in the provided text:

64-Bit Mode and Registers:

In 64-bit mode, the RCX register is commonly used as the loop counter with the LOOP instruction.

Programming in 64-bit mode is often more straightforward when using 64-bit integer variables and registers.

ASCII strings consist of bytes, making indirect or indexed addressing common when working with them.

64-Bit Version of SumArray:

```
; Summing an Array (SumArray_64.asm)
ExitProcess PROTO

.data
intarray QWORD 1000000000000h,2000000000000h,3000000000000h,4000000000000h

.code
main PROC
    mov rdi,OFFSET intarray    ; RDI = address of intarray
    mov rcx,LENGTHOF intarray ; initialize loop counter
    mov rax,0                  ; sum = 0

L1: ; mark beginning of loop
    add rax,[rdi]              ; add a quadword
    add rdi,TYPE intarray      ; point to next element
    loop L1                    ; repeat until RCX = 0

    mov ecx,0                  ; ExitProcess return value
    call ExitProcess
main ENDP
END main
```

The code is an example of a program called "SumArray" that calculates the sum of an array of 64-bit integers.

It uses the QWORD directive to define an array of quadwords (64-bit integers).

Registers are updated to use 64-bit counterparts, like RDI, RCX, and RAX.

The program iterates through the array, adding each 64-bit integer to the RAX register.

The loop continues until the RCX loop counter reaches zero. The program then sets ECX to zero (for ExitProcess) and calls the ExitProcess function to exit.

Addition and Subtraction in 64-Bit Mode:

```
; Addition and Subtraction in 64-Bit Mode (AddSubtract_64.asm)

.data

.code
main PROC
    ; Adding 1 to a 32-bit number in RAX
    mov rax,0FFFFFFFFh      ; fill the lower 32 bits
    add rax,1               ; RAX = 100000000h

    ; Adding 16-bit values in AX and BX
    mov rax,0FFFFh          ; RAX = 000000000000FFFF
    mov bx,1                ; BX = 0001
    add ax,bx               ; RAX = 0000000000000000

    ; Subtracting 1 from zero in EAX
    mov rax,0               ; RAX = 0000000000000000
    mov ebx,1               ; EBX = 00000001
    sub eax,ebx             ; RAX = 00000000FFFFFFFF

    ; Subtracting 1 from zero in AX
    mov rax,0               ; RAX = 0000000000000000
    mov bx,1                ; BX = 0001
    sub ax,bx               ; RAX = 000000000000FFFF

    ; Exit the program
    mov ecx,0               ; ExitProcess return value
    call ExitProcess
main ENDP
END main
```

The ADD, SUB, INC, and DEC instructions in 64-bit mode affect CPU status flags similarly to 32-bit mode.

In an example, adding 1 to a 32-bit number in RAX causes bit 32 to receive a 1, resulting in RAX becoming 100000000h.

When using partial register operands, be aware that the remainder of the register is not modified. For example, adding 16-bit values in AX and BX does not affect the upper bits in RAX.

The same principle applies to subtraction. Subtracting 1 from zero in

EAX or AX modifies the lower bits of RAX while leaving the upper bits unchanged.

These concepts outline the behavior of instructions and registers in 64-bit mode, particularly with regard to handling integers and loops.

Use of 64-bit Registers with Indirect Operands:

In 64-bit mode, when working with indirect operands, you must use a 64-bit general-purpose register to hold the memory address. Additionally, it's essential to use the PTR operator to specify the size of the target operand. Here are some examples:

```
dec BYTE PTR [rdi]          ; Decrement an 8-bit target
inc WORD PTR [rbx]          ; Increment a 16-bit target
inc QWORD PTR [rsi]         ; Increment a 64-bit target
```

dec BYTE PTR [rdi]:

- **dec:** This instruction decrements (subtracts 1 from) the operand.
- **BYTE PTR:** It specifies that the target operand is a byte (8 bits) in size.
- **[rdi]:** This indicates that the memory location pointed to by the 64-bit register rdi will be decremented.
- **Explanation:** This instruction decreases the value at the memory location pointed to by rdi by 1 byte. It's useful for operations involving 8-bit data, like decrementing a byte in memory.

inc WORD PTR [rbx]:

- **inc:** This instruction increments (adds 1 to) the operand.
- **WORD PTR:** It specifies that the target operand is a word (16 bits) in size.
- **[rbx]:** This indicates that the memory location pointed to by the 64-bit register rbx will be incremented.
- **Explanation:** This instruction increases the value at the memory location pointed to by rbx by 1 word (16 bits). It's used for operations involving 16-bit data, like incrementing a 16-bit value in memory.

inc QWORD PTR [rsi]:

- **inc:** This instruction increments (adds 1 to) the operand. QWORD PTR: It specifies that the target operand is a quadword (64 bits) in size.
- **[rsi]:** This indicates that the memory location pointed to by the 64-bit register rsi will be incremented.
- **Explanation:** This instruction increases the value at the memory location pointed to by rsi by 1 quadword (64 bits). It's suitable for operations involving 64-bit data, such as incrementing a 64-bit integer in memory.

In summary, these instructions allow you to manipulate data of different sizes in memory, whether it's an 8-bit, 16-bit, or 64-bit target. The specific size is indicated by the BYTE PTR, WORD PTR, or QWORD PTR specifiers, and the operation (increment or decrement) is performed accordingly on the target operand at the specified memory location.

Scale Factors in Indexed Operands:

In 64-bit mode, you can also use scale factors in indexed operands, similar to 32-bit mode. If you're working with an array of 64-bit integers, you can use a scale factor of 8 to correctly calculate offsets. Here's an example:

```
.data
    array QWORD 1,2,3,4

.code
    mov esi,3                   ; Subscript
    mov eax, array[rsi*8]       ; EAX = 4
```

In this example, we're accessing a 64-bit integer in the array using a scale factor of 8 to account for the size of each element.

64-Bit Pointers:

In 64-bit mode, pointer variables hold 64-bit offsets. Here's an example where the variable ptrB holds the offset of the arrayB:

```
.data
    arrayB BYTE 10h,20h,30h,40h
    ptrB QWORD arrayB
```

Optionally, you can declare ptrB with the OFFSET operator to make the relationship between the pointer and the array clearer:

```
.data
    arrayB BYTE 10h,20h,30h,40h
    ptrB QWORD OFFSET arrayB
```

In this chapter, we've covered various essential concepts related to data transfers, addressing, arithmetic, and loops in assembly language programming. Here's a concise summary of the key points:

MOV Instruction:
• The MOV instruction is used for transferring data from a source operand to a destination operand.

Zero and Sign Extension:
• MOVZX extends a smaller operand to a larger one by zero-filling.
• MOVSX extends a smaller operand to a larger one by sign extension.

XCHG Instruction:
• XCHG swaps the contents of two operands, with at least one of them being a register.

Operand Types:
• Direct operand represents the address of a variable.
• Direct-offset operand adds a displacement to a variable's name, generating a new offset.
• Indirect operand uses a register containing the address of data.
• Indexed operand combines a constant with an indirect operand for effective memory access.

Arithmetic Instructions:

• INC adds 1 to an operand.
• DEC subtracts 1 from an operand.
• ADD adds a source operand to a destination operand.
• SUB subtracts a source operand from a destination operand.
• NEG reverses the sign of an operand.

Status Flags:
• Various CPU status flags are affected by arithmetic operations, including Sign, Carry, Parity, Auxiliary Carry, Zero, and Overflow flags.

Operators:
• OFFSET operator returns the distance of a variable from the beginning of its segment.
• PTR operator overrides a variable's declared size.
• TYPE operator returns the size of a variable or array element.
• LENGTHOF operator returns the number of elements in an array.
• SIZEOF operator returns the number of bytes used by an array initializer.
• TYPEDEF operator creates user-defined types.

Loops:
• JMP instruction unconditionally branches to another location.
• LOOP instruction is used in counting-type loops, with ECX as the counter in 32-bit mode and RCX in 64-bit mode.

64-Bit Mode:
• In 64-bit mode, a 64-bit general-purpose register must be used when working with indirect operands.
• Scale factors can be used in indexed operands, useful for arrays of 64-bit integers.

==========

QUESTIONS:

==========

Question 9: What value will RAX contain after the following instruction executes? mov rax,44445555h

Answer: RAX will contain the hexadecimal value 44445555h.

Question 10: What value will RAX contain after the following instructions execute?

```
.data
    dwordVal DWORD 84326732h
.code
    mov rax,0FFFFFFFF00000000h
    mov rax,dwordVal
```

Answer: RAX will contain the value FFFFFFFF84326732h.

Question 11: What value will EAX contain after the following instructions execute?

```
.data
    dVal DWORD 12345678h
.code
    mov ax,3
    mov WORD PTR dVal+2,ax
    mov eax,dVal
```

Answer: EAX will contain the value 00030000h.

Question 12: What will EAX contain after the following instructions execute?

```
.data
   dVal DWORD ?
.code
    mov dVal,12345678h
    mov ax,WORD PTR dVal+2
    add ax,3
    mov WORD PTR dVal,ax
    mov eax,dVal
```

Answer: EAX will contain the value 00030000h.

Question 12: Is it possible to set the Overflow flag if you add a positive integer to a negative integer?

Answer: Yes, it is possible to set the Overflow flag when adding a positive integer to a negative integer if the result is out of range for the destination operand.

Question 13: Will the Overflow flag be set if you add a negative integer to a negative integer and produce a positive result?

Answer: No, the Overflow flag will not be set when adding two negative integers that result in a positive value.

Question 14: Is it possible for the NEG instruction to set the Overflow flag?

Answer: Yes, the NEG instruction can set the Overflow flag if the operation results in an overflow.

Question 15: Is it possible for both the Sign and Zero flags to be set at the same time?

Answer: No, it is not possible for both the Sign and Zero flags to be set simultaneously. If the result is zero, the Zero flag is set, indicating that the sign is not negative, so the Sign flag is not set. If the result is non-zero, the Sign flag is set to indicate whether it's positive or negative.

Exercise 1: Converting from Big Endian to Little Endian

```
.data
bigEndian BYTE 12h,34h,56h,78h
littleEndian DWORD 0

.code
mov eax, 0
mov al, [bigEndian]
mov ah, [bigEndian + 1]
mov [littleEndian], ax

mov eax, 0
mov al, [bigEndian + 2]
mov ah, [bigEndian + 3]
mov [littleEndian + 2], ax
```

Indented:

```
.data
    bigEndian BYTE 12h,34h,56h,78h
    littleEndian DWORD 0

.code
    mov eax, 0
    mov al, [bigEndian]
    mov ah, [bigEndian + 1]
    mov [littleEndian], ax

    mov eax, 0
    mov al, [bigEndian + 2]
    mov ah, [bigEndian + 3]
    mov [littleEndian + 2], ax
```

This program copies the value from bigEndian to littleEndian while

reversing the order of the bytes.

Exercise 2: Exchanging Pairs of Array Values

```
.data
    array DWORD 1, 2, 3, 4, 5, 6, 7, 8
    arraySize DWORD 8

.code
    mov esi, 0
    mov ecx, [arraySize]

    exchangeLoop:
        mov eax, [array + esi]      ; Load the current element
        add esi, 4                  ; Move to the next element
        xchg eax, [array + esi]     ; Exchange the values
        loop exchangeLoop            ; Repeat for all pairs
```

This program exchanges every pair of values in the array with an even number of elements.

Exercise 3: Summing the Gaps between Array Values

```
.data
    array DWORD 0, 2, 5, 9, 10
    arraySize DWORD 5
    sumOfGaps DWORD 0

.code
    mov esi, 0
    mov ecx, [arraySize] - 1     ; One less than the array size

    sumLoop:
        mov eax, [array + esi]   ; Load the current element
        add esi, 4               ; Move to the next element
        sub eax, [array + esi]   ; Calculate the gap
        add [sumOfGaps], eax     ; Add gap to the sum
        loop sumLoop             ; Repeat for all gaps
```

This program calculates the sum of all the gaps between successive array elements in the array variable.

You can assemble and run these programs using an x86 assembler like NASM or MASM in either 32-bit or 64-bit mode as per your preference.