Extended Addition and Subtraction

The **ADC (add with carry) instruction** in assembly language is used to add two operands, taking into account the Carry flag.

The Carry flag is set when the result of a previous addition or subtraction operation overflows.

The ADC instruction is typically used to perform multi-byte or multi-word addition and subtraction operations.

```
0976 ; Load the first operand into the AL register
0977 mov al, 0FFh
0978
0979 ; Add the second operand to the AL register, setting the Carry flag if necessary
0980 add al, 0FFh
0981
0982 ; Save the result in the DL register, adding the Carry flag if necessary
0983 adc dl, 0
```

After the first instruction, the AL register contains the value FEh and the Carry flag is set.

The second instruction then adds the Carry flag to the DL register, resulting in a final value of 01FEh in the DL:AL register pair.

Here is another example of using the ADC instruction:

```
0988 ; Load the first operand into the EAX register
0989 mov eax, 0xFFFFFFFFh
0990
0991 ; Add the second operand to the EAX register, setting the Carry flag if necessary
0992 add eax, 0xFFFFFFFFh
0993
0994 ; Save the result in the EDX register, adding the Carry flag if necessary
0995 adc edx, 0
```

After the first instruction, the EAX register contains the value FFFFFFFFh and the Carry flag is set.

The second instruction then adds the Carry flag to the EDX register, resulting in a final value of 00000001FFFFFFFEh in the EDX:EAX register pair.

The ADC instruction can be used to add operands of any size, including 1024-bit integers.

To do this, multiple ADC instructions would be used in sequence, carrying the Carry flag from one instruction to the next.

Here is an example of how to add two 1024-bit integers using the ADC instruction:

```
; Load the first operand into the [EAX:EBX:ECX:EDX] register quad
mov eax, [operand1]
mov ebx, [operand1 + 4]
mov ecx, [operand1 + 8]
mov edx, [operand1 + 12]

; Load the second operand into the [ESI:EDI:ESI:EDI] register quad
mov esi, [operand2]
mov edi, [operand2 + 4]

; Add the two operands, carrying the Carry flag from each instruction to the next
adc eax, esi
adc ebx, edi
adc ecx, edi
adc edx, esi

; Save the result in the [EAX:EBX:ECX:EDX] register quad
mov [result], eax
mov [result + 4], ebx
mov [result + 8], ecx
mov [result + 12], edx
```

This code will add the two 1024-bit operands stored in the operand1 and operand2 arrays and store the result in the result array. Let's break down the code step by step:

Loading the First Operand (operand1):

The code begins by loading the first operand, which is a 1024-bit value, into the [EAX:EBX:ECX:EDX] register quad. It does this in four 32-bit chunks (4 * 32 = 128 bits):

mov eax, [operand1]: Loads the first 32 bits of operand1 into the EAX register.

mov ebx, [operand1 + 4]: Loads the next 32 bits (bits 32-63) of

operand1 into the EBX register.

mov ecx, [operand1 + 8]: Loads the following 32 bits (bits 64-95) of operand1 into the ECX register.

mov edx, [operand1 + 12]: Loads the last 32 bits (bits 96-127) of operand1 into the EDX register. Loading the Second Operand (operand2):

The code then loads the second operand, which is also a 1024-bit value, into the [ESI:EDI:ESI:EDI] register quad. Like the first operand, it does this in four 32-bit chunks:

mov esi, [operand2]: Loads the first 32 bits of operand2 into the ESI register.

mov edi, [operand2 + 4]: Loads the next 32 bits (bits 32-63) of operand2 into the EDI register.

Adding the Two Operands with Carry Propagation:

The actual addition of the two operands is performed in this step. It uses the adc (add with carry) instruction, which allows for carry propagation.

adc eax, esi: Adds the first 32 bits of the first operand (EAX) and the first 32 bits of the second operand (ESI) along with any carry from the previous addition. The result is stored in EAX.

adc ebx, edi: Adds the next 32 bits of the first operand (EBX) and the next 32 bits of the second operand (EDI) along with any carry from the previous addition. The result is stored in EBX.

adc ecx, edi: Adds the following 32 bits of the first operand (ECX) and the next 32 bits of the second operand (EDI) along with any carry from the previous addition. The result is stored in ECX.

adc edx, esi: Adds the last 32 bits of the first operand (EDX) and the first 32 bits of the second operand (ESI) along with any carry from the previous addition. The result is stored in EDX.

Storing the Result in the result Array:

Finally, the result of the addition is saved back into the result array in four 32-bit chunks.

mov [result], eax: Stores the first 32 bits of the result (in EAX) in the result array.

mov [result + 4], ebx: Stores the next 32 bits (in EBX) of the result in the result array.

mov [result + 8], ecx: Stores the following 32 bits (in ECX) of the result in the result array.

mov [result + 12], edx: Stores the last 32 bits (in EDX) of the result in the result array.

This code essentially performs a multi-precision addition for 1024-bit operands, taking care of carry propagation between 32-bit chunks. It's a low-level operation that handles large integers by breaking them into manageable pieces.

The ADC instruction is a powerful tool for performing multi-byte and multi-word addition and subtraction operations. It can be used to add and subtract operands of any size, including very large integers.

==========

The Extended_Add procedure below is an example of how to add two extended integers of the same size using assembly language.

It works by iterating through the two integers, adding each corresponding byte and carrying over any carry from the previous iteration. The procedure takes four arguments:

ESI and EDI: Pointers to the two integers to be added.

EBX: A pointer to a buffer in which the sum will be stored. The buffer must be one byte longer than the two integers.

ECX: The length of the longest integer in bytes. The procedure assumes that the integers are stored in little-endian order, with the least significant byte at the lowest offset.

Here is a more detailed explanation of the code:

```
;-----------------------------------------------------
;Extended_Add PROC
;-----------------------------------------------------
pushad ; Save all registers on the stack.
clc ; Clear the Carry flag.

L1: mov al,[esi] ; Get the next byte from the first integer.
; Add the next byte from the second integer, including any carry from the previous iteration.
adc al,[edi]
pushfd ; Save the Carry flag.
mov [ebx],al ; Store the partial sum.
add esi,1 ; Advance the pointers to the next bytes in the integers.
add edi,1
add ebx,1
popfd ; Restore the Carry flag.
loop L1 ; Repeat the loop until all bytes have been added.

; Clear the high byte of the sum, since it may contain a carry.
mov byte ptr [ebx],0
adc byte ptr [ebx],0 ; Add any leftover carry.

popad ; Restore all registers from the stack.
ret ; Return from the procedure.
Extended_Add ENDP
```

The loop at L1 iterates through the two integers, adding each corresponding byte and carrying over any carry from the previous iteration.

The Carry flag is saved and restored on each iteration so that it is always in the correct state when the next addition is performed.

After the loop has finished iterating, the high byte of the sum is cleared.

This is necessary because the high byte may contain a carry from the addition of the two highest bytes of the integers.

The ADC instruction is then used to add any leftover carry to the high byte of the sum.

Finally, the registers are restored from the stack and the procedure returns.

The Extended_Add procedure is a useful example of how to perform extended precision arithmetic in assembly language.

It can be used to add two integers of any size, regardless of whether they fit within the registers of the CPU.

The following sample code calls the Extended_Add procedure to add two 8-byte integers:

```
.data
    op1 BYTE 34h, 12h, 98h, 74h, 06h, 0A4h, 0B2h, 0A2h
    op2 BYTE 02h, 45h, 23h, 00h, 00h, 87h, 10h, 80h
    sum BYTE 9 dup(0)

.code
main PROC
    ; Load addresses of operands and result
    mov esi, OFFSET op1   ; First operand
    mov edi, OFFSET op2   ; Second operand
    mov ebx, OFFSET sum   ; Result operand

    ; Determine the number of bytes to process
    mov ecx, LENGTHOF op1

    ; Call the Extended_Add function
    call Extended_Add

    ; Display the sum.
    mov esi, OFFSET sum
    mov ecx, LENGTHOF sum
    call Display_Sum

    ; Call a function to output a newline.
    call Crlf

    ; Exit the program
    invoke ExitProcess, 0

main ENDP
```

The .data section of the code defines three byte arrays: op1, op2, and sum. The op1 and op2 arrays store the two integers to be added, and the sum array will store the result of the addition.

The sum array is one byte longer than the other two arrays to accommodate any carry that may be generated.

The .code section of the code contains the main procedure.

The main procedure first moves the pointers to the two operand arrays and the sum array into the ESI, EDI, and EBX registers, respectively. It then moves the length of the operands into the ECX register.

Next, the main procedure calls the Extended_Add procedure.

The Extended_Add procedure will add the two operands and store the result in the sum array.

After the Extended_Add procedure has finished executing, the main procedure moves the pointer to the sum array into the ESI register and the length of the sum array into the ECX register.

It then calls the Display_Sum procedure to display the sum to the console.

The Display_Sum procedure is a simple procedure that iterates through the sum array and prints each byte to the console.

After the Display_Sum procedure has finished executing, the main procedure calls the Crlf procedure to print a newline character to the console.

The output of the program is the following:

```
1087  0122C32B0674BB5736
```

This is the correct sum of the two operands, even though the addition produced a carry. The Extended_Add procedure handles the carry correctly and stores the correct result in the sum array.

The Display_Sum procedure below is related to the Extended_Add procedure you provided in the previous question.

The Display_Sum procedure is used to display the sum of two integers that have been added using the Extended_Add procedure.

It works by iterating through the sum array in reverse order, starting with the high-order byte and working its way down to the low-order byte.

For each byte in the sum array, the Display_Sum procedure calls the WriteHexB procedure to display the byte in hexadecimal format.

Here is a more detailed explanation of the Display_Sum procedure:

```
1092 Display_Sum PROC
1093     pushad           ; Save registers
1094     ; Point to the last array element
1095     add esi, ecx
1096     sub esi, TYPE BYTE
1097     mov ebx, TYPE BYTE
1098
1099 L1:
1100     mov al, [esi]    ; Get a byte from the array
1101     call WriteHexB   ; Display it in hexadecimal
1102     sub esi, TYPE BYTE  ; Move to the previous byte
1103     loop L1
1104
1105     popad            ; Restore registers
1106     ret
1107 Display_Sum ENDP
```

The pushad instruction saves all of the registers on the stack. The add esi,ecx instruction moves the value of the ECX register into the ESI register.

The sub esi,TYPE BYTE instruction subtracts the size of a byte from the ESI register. This moves the pointer to the last element of the

sum array.

The mov ebx,TYPE BYTE instruction moves the size of a byte into the EBX register. This will be used to loop through the sum array.

The L1: label marks the beginning of the loop.

The mov al,[esi] instruction moves the byte at the current position in the sum array into the AL register. The call WriteHexB instruction calls the WriteHexB procedure to display the byte in hexadecimal format.

The sub esi,TYPE BYTE instruction subtracts the size of a byte from the ESI register. This moves the pointer to the previous byte in the sum array.

The loop L1 instruction loops back to the beginning of the loop if the EBX register is not zero.

The popad instruction restores all of the registers from the stack. The ret instruction returns from the procedure.

The Display_Sum procedure is a good example of how to iterate through an array in reverse order. It is also a good example of how to call another procedure from within a procedure.

==========

Subtract with Borrow

==========

The SBB (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag from a destination operand.

The possible operands are the same as for the ADC instruction, which means it can be used to subtract operands of any size, including 32-bit, 64-bit, and even 128-bit operands.

The following example code carries out 64-bit subtraction with 32-bit operands:

```
mov edx, 7
; upper half
mov eax, 1
; lower half
sub eax, 2
; subtract 2
sbb edx, 0
; subtract upper half
```

This code will subtract the value 2 from the 64-bit integer stored in the EDX:EAX register pair. The subtraction is done in two steps:

The value 2 is subtracted from the lower 32 bits of the integer, which are stored in the EAX register. This subtraction may set the Carry flag if a borrow is required.

The SBB instruction subtracts both 0 and the value of the Carry flag from the upper 32 bits of the integer, which are stored in the EDX register.

Here is a more detailed explanation of the code:

```
mov edx, 7
; upper half
```

This instruction moves the value 7 into the EDX register. This is the upper 32 bits of the 64-bit integer that we will be subtracting from.

```
mov eax, 1
; lower half
```

This instruction moves the value 1 into the EAX register. This is the lower 32 bits of the 64-bit integer that we will be subtracting from.

```
sub eax, 2
; subtract 2
```

This instruction subtracts the value 2 from the lower 32 bits of the integer, which are stored in the EAX register. This may set the Carry flag if a borrow is required.

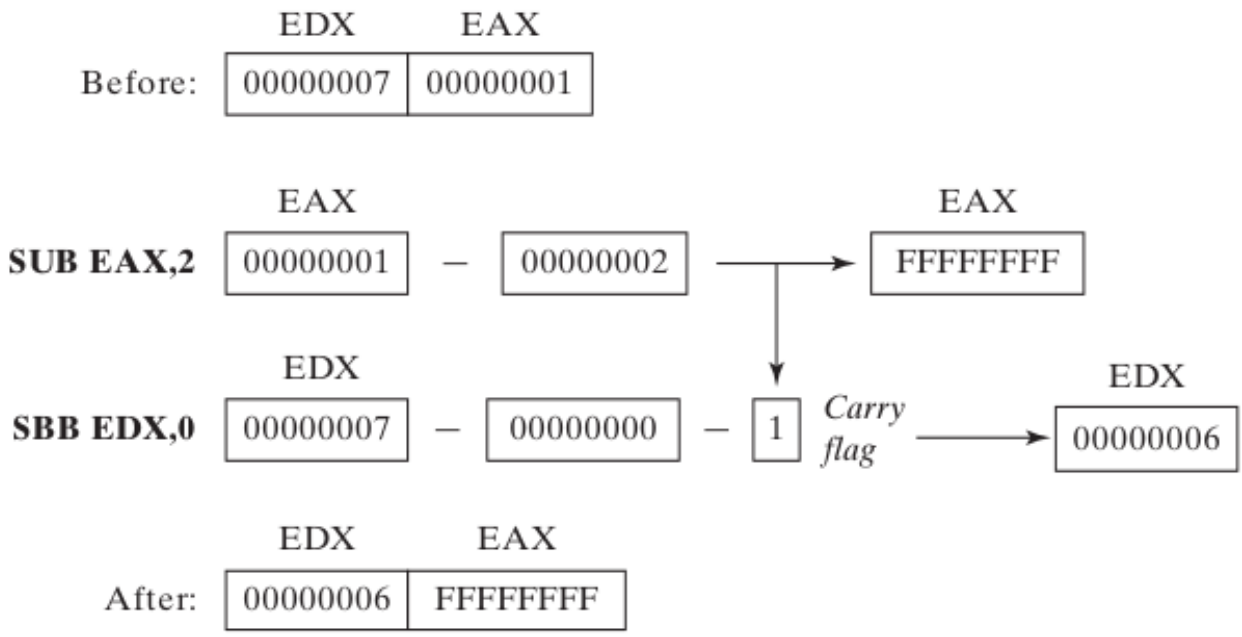
```
10 sbb edx, 0
11 ; subtract upper half
```

This instruction subtracts both 0 and the value of the Carry flag from the upper 32 bits of the integer, which are stored in the EDX register.

After the code has executed, the EDX:EAX register pair will contain the result of the subtraction, which is the value 0000000700000001h.

The SBB instruction is a powerful tool for performing multi-byte and multi-word subtraction operations. It can be used to subtract operands of any size, including very large integers.

FIGURE 7–2 Subtracting from a 64-bit integer using SBB.



Describe the ADC instruction:

The ADC (Add with Carry) instruction is used for addition in assembly language. It adds two operands, along with the value of the Carry flag, and stores the result in the destination operand. If there is a carry from the addition, it sets the Carry flag; otherwise, it clears

it. It is particularly useful for multi-precision arithmetic, where you need to handle carry from previous operations.

Describe the SBB instruction:

The SBB (Subtract with Borrow) instruction is used for subtraction in assembly language. It subtracts the source operand from the destination operand, along with the Borrow flag (Carry flag treated as borrow), and stores the result in the destination operand. If a borrow is generated from the subtraction, it sets the Carry flag; otherwise, it clears it. SBB is often used for multi-precision arithmetic to handle borrows from previous operations.

Values of EDX:EAX after the given instructions execute:

mov edx, 10h loads 16 into EDX.
mov eax, 0A0000000h loads A0000000h into EAX. add eax, 20000000h adds 20000000h to EAX without carry. adc edx, 0 adds 0 to EDX along with any carry. Result: EDX = 0 (no carry), EAX = C0000000h.

Values of EDX:EAX after the given instructions execute:

mov edx, 100h loads 256 into EDX.
mov eax, 80000000h loads 80000000h into EAX.
sub eax, 90000000h subtracts 90000000h from EAX without borrow. sbb edx, 0 subtracts 0 from EDX along with any borrow.
Result: EDX = FFFFFFFF (due to borrow), EAX = FFFFFFFF.

Contents of DX after the given instructions execute:

mov dx, 5 loads 5 into DX.
stc sets the Carry flag to 1.
mov ax, 10h loads 16 into AX.
adc dx, ax adds AX to DX along with the carry. Result: DX = 1 (due to carry).