Contents

SHIFTS AND ROTATES

In assembly language, you have direct control over the bits and bytes of data, which allows for highly efficient and platform-specific optimizations. Let's dive a bit deeper into these concepts:

Bit shifting involves moving the bits of a binary number left or right.

Left Shift (<<): Shifting bits to the left by a certain number of positions effectively multiplies the value by 2 for each shift.

Right Shift (>>): Shifting bits to the right by a certain number of positions effectively divides the value by 2 for each shift (for non-negative numbers).

Logical Shift: Fills the shifted-in bits with zeros. Arithmetic Shift: Preserves the sign bit when shifting right (for signed numbers).

Bit Rotation: Bit rotation involves moving bits in a circular manner, so they wrap around. Left Rotation: Bits are shifted left, and the bits that go beyond the most significant bit (MSB) are wrapped around to the least significant bit (LSB).

Right Rotation: Bits are shifted right, and the bits that go beyond the LSB are wrapped around to the MSB.

Uses:

Optimized Multiplication and Division: Bit shifting can be used to perform multiplication and division more efficiently.

For example, shifting left by n positions is equivalent to multiplying by 2^n, and shifting right is equivalent to division by 2^n.

Data Encryption: Bit manipulation plays a crucial role in encryption algorithms like the XOR cipher.

Computer Graphics: Manipulating pixels and colors often involves bit-level operations for tasks like image processing and rendering.

Hardware Manipulation: In embedded systems, controlling hardware often requires setting or clearing specific bits to interact with peripherals. These concepts are indeed powerful and can lead to highly optimized code.

Assembly language provides the flexibility to perform arithmetic operations on arbitrary-length integers, a feature not always readily available in high-level languages.

This capability can be particularly useful when dealing with large numbers or implementing custom arithmetic operations.

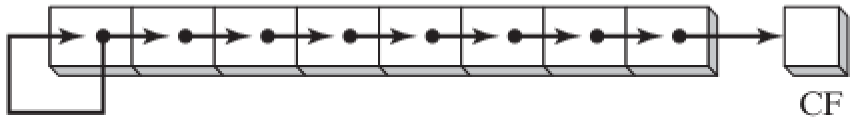
Let's discuss some key points related to shift and rotate instructions in assembly language, especially focusing on x86 processors:

==
=

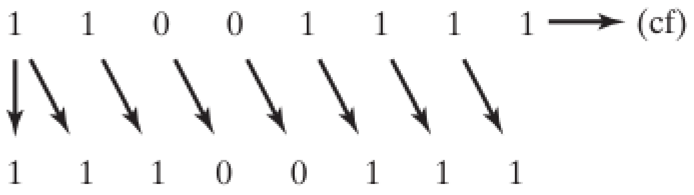
SAL/SAR (Arithmetic Shift Left/Arithmetic Shift Right):

Similar to SHL and SHR, but SAR preserves the sign bit when shifting right, making it suitable for signed integers.

The newly created bit position is filled with a copy of the original number's sign bit:



Binary 11001111, for example, has a 1 in the sign bit. When shifted arithmetically 1 bit to the right, it becomes 11100111:



ROL/ROR (Rotate Left/Rotate Right):

Rotation instructions are used to shift bits in a circular manner, wrapping around from one end to the other. ROL rotates bits left, and ROR rotates bits right.

RCL/RCR (Rotate through Carry Left/Rotate through Carry Right):

These instructions are similar to ROL and ROR but incorporate the Carry flag, making them useful for multi-precision arithmetic and shifts.

SHIFT LEFT AND SHIFT RIGHT

==============================

Shift and Rotate Instructions:

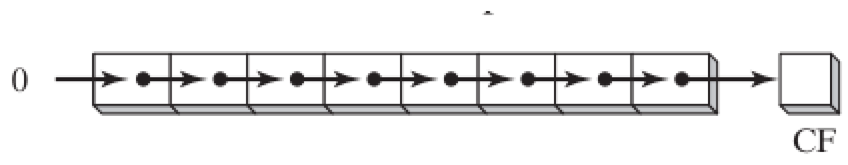
================================

Bit shifting involves moving bits within an operand either to the left or right. The x86 processor architecture offers a wide range of shift and rotate instructions, each with specific purposes and effects on flags like Overflow and Carry. Here are a few common shift and rotate instructions on x86:

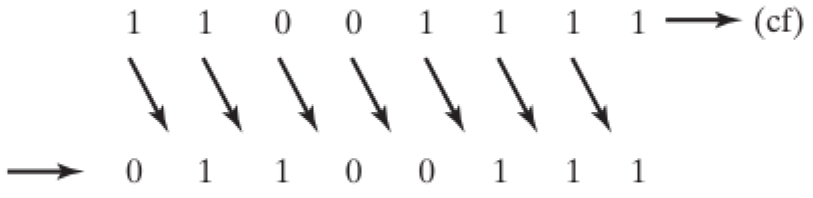
===
=

SHL(Shift Left):

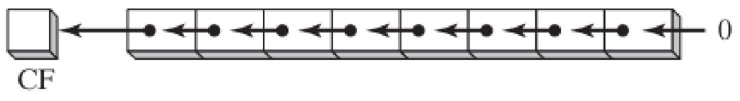
These instructions shift bits left (SHL) or right (SHR) within an operand. SHL multiplies the value by 2 for each shift left, while SHR effectively divides the value by 2 for each shift right.



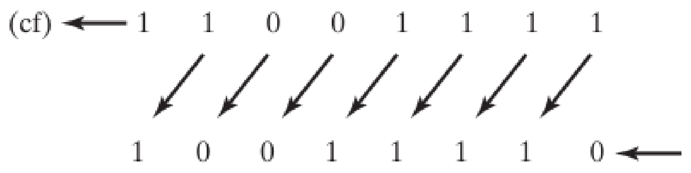
The following illustration shows a single logical right shift on the binary value 11001111, producing 01100111. The lowest bit is shifted into the Carry flag:



The **SHL (shift left) instruction** performs a logical left shift on the destination operand, filling the lowest bit with 0. The highest bit is moved to the Carry flag, and the bit that was in the Carry flag is discarded:



If you shift 11001111 left by 1 bit, it becomes 10011110:



```
SHL destination, count
```

```
SHL reg,imm8
SHL mem,imm8
SHL reg,CL
SHL mem,CL
```

The SHL instruction can **only be used to shift integers**, not floating-point numbers.

The imm8 operand must be between 0 and 7, inclusive. For example, the following instruction is invalid:

```
shl eax, 8
```

If the imm8 operand is greater than 7, the shift count will be wrapped around to the range 0-7. For example, the following instruction is equivalent to the instruction shl eax, 1:

```
shl eax, 9
```

If the CL register is used as the shift count operand, it must contain a value between 0 and 31, inclusive. For example, the following instruction is invalid:

```
shl eax, cl ; cl = 32
```

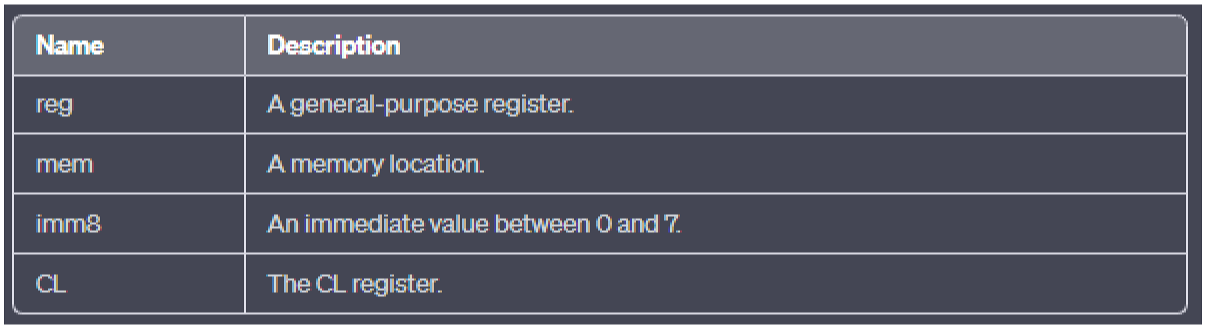
If the CL register contains a value greater than 31, the shift count will be wrapped around to the range 0-31. For example, the following instruction is equivalent to the instruction shl eax, 1:

```
shl eax, cl ; cl = 33
```

Here is a more clear explanation of the SHL instruction:

- The SHL instruction shifts the bits of the destination operand to the left by the specified number of bits.
- The highest bit of the destination operand is shifted out and copied into the Carry flag.
- The lowest bit position of the destination operand is assigned zero.

The following table shows the possible operands for the SHL instruction:

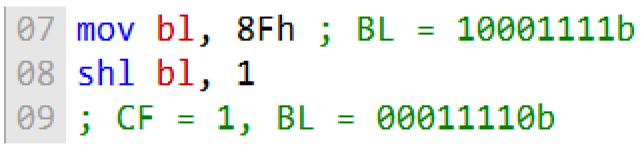


Name	Description
reg	A general-purpose register.
mem	A memory location.
imm8	An immediate value between 0 and 7.
CL	The CL register.

The notes you provided are not clear because they do not explicitly state the following:

- The **SHL instruction** can only be used to shift integers, **not** floating-point numbers.
- The **imm8** operand must be **between 0 and 7, inclusive.** For example, the following instruction is invalid:
- **shl eax, 8** If the imm8 operand is greater than 7, the shift count will be wrapped around to the range 0-7.

For example, the following instruction is equivalent to the instruction shl eax, 1:

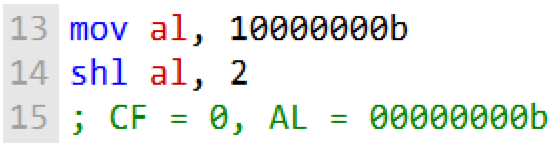


```
mov bl, 8Fh ; BL = 10001111b
shl bl, 1
; CF = 1, BL = 00011110b
```

After the SHL instruction is executed, the Carry flag will be set to 1 and the BL register will contain the value 00011110b.

When the SHL instruction is used to shift a value to the left multiple times, the Carry flag will contain the last bit to be shifted out of the most significant bit (MSB).

For example, the following code shows how to shift the value of the AL register to the left by two bits:



```
mov al, 10000000b
shl al, 2
; CF = 0, AL = 00000000b
```

After the first SHL instruction is executed, the Carry flag will be set to 1 and the AL register will contain the value 01000000b.

After the second SHL instruction is executed, the Carry flag will be set to 0 and the AL register will contain the value 00000000b.

The SHL instruction can be used to perform a variety of operations, such as multiplying a value by two, converting a binary number to a decimal number, and packing and unpacking data.

When a value is shifted rightward multiple times, the **Carry flag contains the last bit to be shifted out of the least significant bit (LSB).**

The image you sent is correct. It shows that shifting the binary number 00001010 (decimal 10) to the left by two bits is the same as multiplying it by 2^2.

This is because shifting a binary number to the left by one bit is the same as multiplying it by 2. Shifting to the left by two bits is the same as multiplying by 2^2, and so on.

```
mov  dl,10                          ; before:  00001010
shl  dl,2                           ; after:   00101000
```

After the SHL instruction is executed, the BL register will contain the value 20.

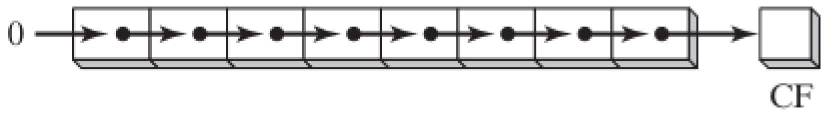
Bitwise multiplication is often used in graphics and signal processing applications. For example, it can be used to scale images, rotate images, and apply filters to images.

==========================

Shift Right Instruction

==========================

The SHR instruction performs a logical right shift on the destination operand, replacing the highest bit with a 0.



The lowest bit is copied into the Carry flag, and the bit that was previously in the Carry flag is lost.

```
mov  al,0D0h                    ; AL = 11010000b
shr  al,1                       ; AL = 01101000b, CF = 0
```

In a multiple shift operation, the last bit to be shifted out of position 0 (the LSB) ends up in the Carry flag:

```
mov  al,00000010b
shr  al,2                       ; AL = 00000000b, CF = 1
```

Bitwise division is a way of dividing two numbers by shifting the bits of the dividend (the number being divided) to the right by the number of bits in the divisor.

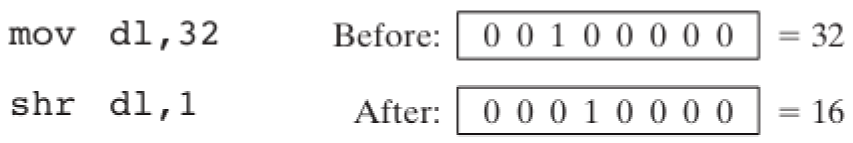
The result of this operation is the quotient of the two numbers, rounded down to the nearest integer.

For example, to divide 32 by 2^1, we would shift the bits of 32 to the right by 1 bit. This would result in the value 16, which is the quotient of 32 divided by 2^1.

Here is a step-by-step explanation of bitwise division:

1. Write the dividend and divisor in binary form.
2. Shift the bits of the dividend to the right by the number of bits in the divisor.
3. The result of the shift is the quotient of the two numbers, rounded down to the nearest integer.

For example, to divide 32 by 2^1, we would do the following:



```
mov  dl,32        Before: 0 0 1 0 0 0 0 0  = 32
shr  dl,1         After:  0 0 0 1 0 0 0 0  = 16
```

In the following example, 64 is divided by 2^3:

```
mov  al,01000000b                  ; AL = 64
shr  al,3                          ; divide by 8, AL = 00001000b
```

1. Write the dividend and divisor in binary form:

```
Dividend: 100000
Divisor: 00100
```

1. Shift the bits of the dividend to the right by 1 bit:

```
100000 >> 1 = 010000
```

1. The result of the shift is the quotient of the two numbers, rounded down to the nearest integer:

```
Quotient = 010000 = 16
```

Bitwise division can be used to perform a variety of operations, such as:

- Dividing a number by a power of two.

- Converting a hexadecimal number to a decimal number.
- Calculating the remainder of a division operation.

Division of signed numbers by shifting is accomplished using the SAR instruction because it preserves the number's sign bit.

For example, if the AX register contains the value 0x1234, then the following instruction will shift the value to the right by one bit:

```
SHR AX, 1
```

After the instruction is executed, the AX register will contain the value 0x091A.

The highest bit of the original value (0x1) has been shifted out and lost, and the lowest bit of the original value (0x4) has been copied into the Carry flag.

The SHR instruction can be used to perform a variety of operations, including:

In this statement, what's being explained is how the SHR (Shift Right) instruction works using an example with the AX register containing the value 0x1234 (a 16-bit hexadecimal number).

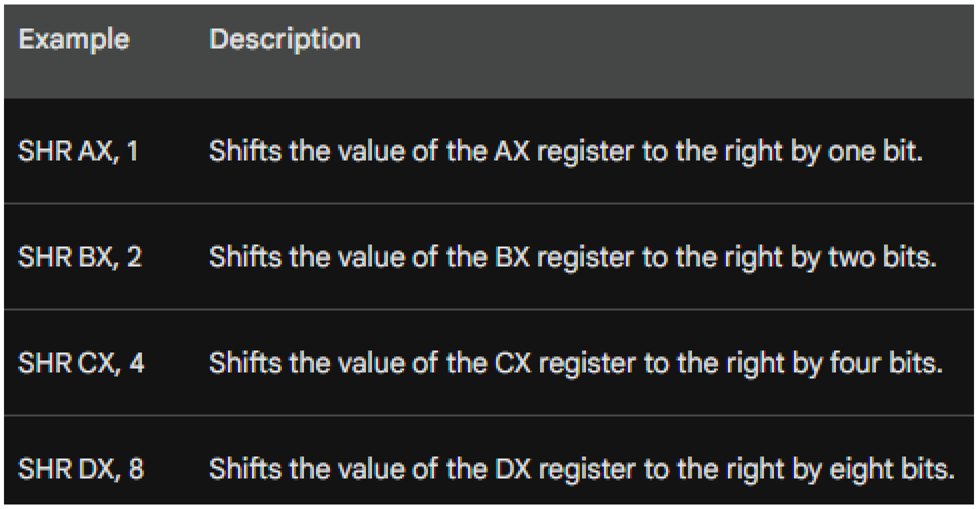
When you apply the SHR instruction to this value, you are shifting the bits of the number to the right by one bit position. Here's a step-by-step breakdown:

1. Original Value: 0x1234 (binary: 0001001000110100)
1. Right Shift by One Bit: Shifting the bits to the right by one position results in "0000100100011010."
1. The highest bit in the original value (0x1) is the leftmost bit, and it gets shifted out and is "lost" because there's no longer space for it in the 16-bit AX register.
1. The lowest bit in the original value (0x4) is the rightmost bit, and it has been copied into the Carry flag. This means that after the shift, the Carry flag will hold the value 1 (indicating that the bit that was shifted out was a 1).
1. The new value in the AX register after the shift is 0x091A (binary: 0000100100011010).

Dividing a value by two

Converting a decimal number to a binary number Unpacking data.

The following table shows some examples of how to use the SHR instruction:



Example	Description
SHR AX, 1	Shifts the value of the AX register to the right by one bit.
SHR BX, 2	Shifts the value of the BX register to the right by two bits.
SHR CX, 4	Shifts the value of the CX register to the right by four bits.
SHR DX, 8	Shifts the value of the DX register to the right by eight bits.

The SHR instruction can also be used to test the value of the Carry flag.

For example, the following instruction will test the value of the Carry flag and set the Zero flag if the Carry flag is clear:

```
SHR AX, 1
JZ TEST
```

If the Carry flag is clear, then the JZ instruction will jump to the TEST label. Otherwise, the JZ instruction will be skipped.

========================

Summary b4 we continue

========================

Certainly, you've provided a detailed explanation of shift and rotate instructions, specifically focusing on SHL (Shift Left) and SHR (Shift Right) instructions in the context of x86 assembly language programming. Here's a summary of the information you've provided:

Shift Left (SHL) Instruction:

- SHL shifts bits to the left within an operand.
- It multiplies the value by 2 for each left shift.
- The highest bit is moved to the Carry flag, and the lowest bit is set to 0.
- It can only be used for integers, not floating-point numbers.
- The imm8 operand must be between 0 and 7, inclusive.
- If imm8 is greater than 7, the shift count wraps around to 0-7.

- If the CL register is used, it must contain a value between 0 and 31.
- If CL exceeds 31, the shift count wraps around.
- It is often used for operations like multiplying a value by 2 and bitwise multiplication.

Shift Right (SHR) Instruction:

- SHR performs a logical right shift on the destination operand.
- It replaces the highest bit with 0 and copies the lowest bit to the Carry flag.
- In multiple shifts, the last bit shifted out ends up in the Carry flag.
- It can be used for operations like dividing a value by 2, converting decimal to binary, and unpacking data.
- SAR (Shift Arithmetic Right) is used for signed numbers to preserve the sign bit.

Bitwise Division:

- Bitwise division involves shifting bits of the dividend to the right by the number of bits in the divisor.
- The result is the quotient, rounded down to the nearest integer.
- It's used for operations like dividing by a power of two and converting hexadecimal to decimal.
- SAR is used for signed numbers to maintain the sign bit.

Since bitwise division (right shifting) and bitwise multiplication (left shifting) are essentially inverse operations of each other, you can focus on one of them in your notes and simply mention their relationship. Here's a more concise way to express this:

- • • **Bitwise Shifting:** This operation involves moving the bits of a binary number.
- • • **Left Shifting:** Equivalent to multiplying by 2 to the power of the shift count.
- • • **Right Shifting:** Equivalent to dividing by 2 to the power of the shift count

Bitwise operations, including left and right shifts, are primarily related to binary data manipulation. They work with binary representations, and you can perform these operations on hexadecimal numbers, but the shifts operate at the binary level.

In contrast, when you perform bitwise operations on other number formats like decimal or hexadecimal, the operations affect their binary representations. Bitwise operations are versatile but fundamentally operate on binary data, while left and right shifts are more broadly applicable, working with various data types beyond binary numbers.

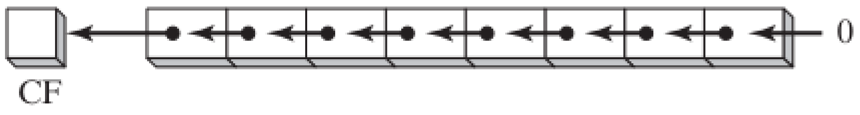
ARITHMETIC SHIFT RIGHT AND ARITHMETIC SHIFT LEFT

===

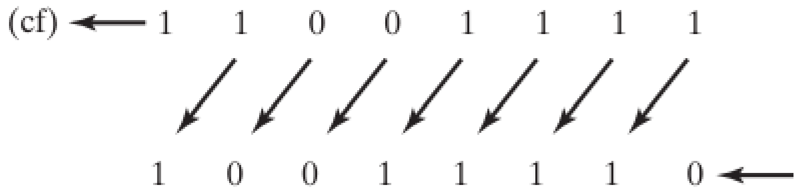
SAL (Shift Arithmetic Left) Instruction:

===

The **SAL instruction**, also known as **Shift Arithmetic Left**, is similar to the SHL (Shift Left) instruction in its behavior. It's used for moving the bits of the destination operand to the left by a specified number of bits.



Shift Left Operation: SAL shifts each bit in the destination operand to the next highest bit position. In other words, it moves the bits leftward. If you shift binary 11001111 to the left by one bit, it becomes 10011110:



Carry Flag: As the bits shift to the left, the highest bit in the destination operand is moved into the Carry flag. The Carry flag stores the value that was in the most significant bit (MSB) position.

Lowest Bit: The lowest bit in the destination operand is assigned the value 0. This means that the bit that was in the least significant bit (LSB) position is set to 0.

Discarding the Carry Bit: The bit that was in the Carry flag is effectively discarded in this operation. It is no longer a part of the destination operand.

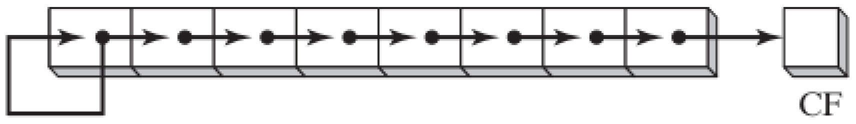
The SAL instruction is typically used to perform logical left shifts and, like SHL, is often used for operations such as multiplying a value by powers of 2. It's a valuable tool for bit manipulation in assembly language programming.

===

SAR (Shift Arithmetic Right) Instruction:

===

The SAR instruction, which stands for Shift Arithmetic Right, is used for shifting the bits of the destination operand to the right by a specified number of bits. Unlike logical right shifts (as in SHR), SAR is used for signed numbers to preserve the sign bit.



Shift Right Operation: SAR performs a right shift on the bits of the destination operand, moving them to the right.

```
SAR destination,count
```

Sign Bit Preservation: The key feature of SAR is that it preserves the sign bit. This means that when a signed number is shifted right, the most significant bit (MSB), which indicates the sign (0 for positive, 1 for negative), is preserved during the shift.

Carry Flag: The lowest bit of the destination operand is moved into the Carry flag, similar to other shift instructions. The Carry flag now stores the value that was in the least significant bit (LSB) position.

Bit Insertion: The highest bit position, which was shifted out, is determined by the original sign bit. If the original sign bit was 0 (indicating a positive number), the highest bit is set to 0. If the original sign bit was 1 (indicating a negative number), the highest bit is set to 1. This preserves the signedness of the number.

SAR is used for operations like signed division by powers of 2 and other signed arithmetic operations. It's a crucial instruction in working with signed numbers in assembly language programming, ensuring that the sign bit is maintained correctly during shifts.

=============

Examples:

=============

The following example shows how **SAR duplicates the sign bit**. AL is negative before and after it is shifted to the right:

```
mov  al,0F0h                        ; AL = 11110000b (-16)
sar  al,1                           ; AL = 11111000b (-8), CF = 0
```

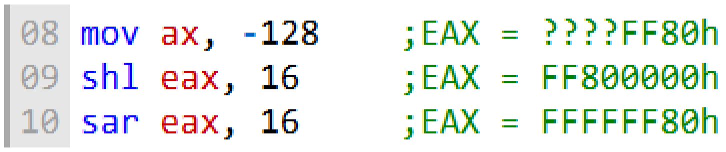
Signed Division

You can divide a signed operand by a power of 2, using the SAR instruction. In the following example, –128 is divided by 23. The quotient is –16:

```
mov  dl,-128                        ; DL = 10000000b
sar  dl,3                           ; DL = 11110000b
```

Sign-Extend AX into EAX

Suppose AX contains a signed integer and you want to extend its sign into EAX. First shift EAX 16 bits to the left, then shift it arithmetically 16 bits to the right:



```
mov ax, -128     ;EAX = ????FF80h
shl eax, 16      ;EAX = FF800000h
sar eax, 16      ;EAX = FFFFFF80h
```

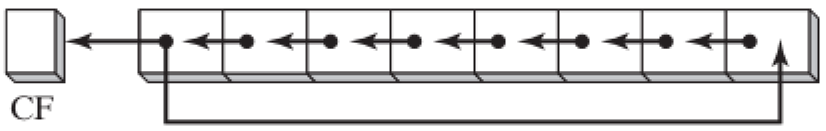
ROTATE LEFT AND ROTATE RIGHT

================

ROL Instruction

================

ROL (rotate left) instruction is used to perform bitwise rotation, where bits are shifted in a circular fashion. There are different ways to achieve this, including one where the bit leaving one end of the number is immediately copied into the other end.



ROL can also use the Carry flag as an intermediate point for shifted bits.

When you use the ROL instruction, each bit in the operand is shifted to the left. The highest bit (the one farthest to the left) is copied into **both the Carry flag and the lowest bit position** (farthest to the right). The instruction format is similar to that of the SHL instruction.

Preservation of Bits: Bit rotation is unique in that it doesn't lose any bits during the process. If a bit is rotated off one end of a number, it appears again at the other end. For example, let's consider the following code:

```
mov al, 40h ; AL = 01000000b
rol al, 1    ; AL = 10000000b, CF = 0
rol al, 1    ; AL = 00000001b, CF = 1
rol al, 1    ; AL = 00000010b, CF = 0
```

In this example, we start with the value 01000000b in AL, and after three rotations, the high bit is copied into both the Carry flag (CF) and bit position 0.

Multiple Rotations: When you use a rotation count greater than 1, the Carry flag contains the last bit rotated out of the most significant bit (MSB) position. For instance:

```
mov al, 00100000b
rol al, 3 ; CF = 1, AL = 00000001b
```

Here, when we rotate AL three times to the left, the Carry flag (CF) holds the last bit shifted out of the most significant bit position.

Exchanging Groups of Bits: ROL can be employed to exchange groups of bits within a byte. For example, by rotating a value like 26h four bits in either direction, you can effectively swap the upper (bits 4–7) and lower (bits 0–3) halves of a byte:

```
mov al, 26h
rol al, 4 ; AL = 62h
```

Furthermore, when you rotate a multibyte integer by four bits, you are effectively shifting each hexadecimal digit one position to the right or left. For instance:

```
mov ax, 6A4Bh
rol ax, 4 ; AX = A4B6h
rol ax, 4 ; AX = 4B6Ah
rol ax, 4 ; AX = B6A4h
rol ax, 4 ; AX = 6A4Bh
```

In this sequence, you can observe how each four-bit chunk of the value is rotated, eventually returning to the original value.

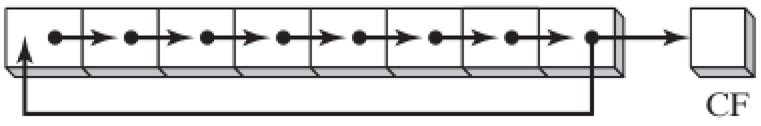
These instructions provide valuable tools for bit manipulation and data reordering in assembly language programming.

================

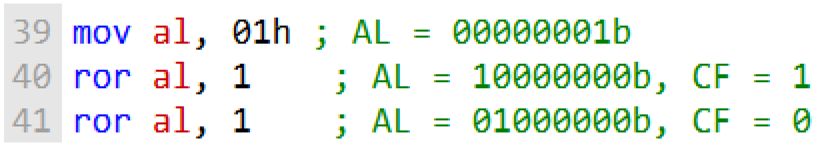
ROR Instruction

================

The **ROR (rotate right) instruction** is used to perform bitwise rotation to the right. In this operation, each bit is shifted to the right, and the lowest bit (the one farthest to the right) is copied into both the Carry flag (CF) and the highest bit position (the one farthest to the left). The instruction format for ROR is the same as that for SHL.



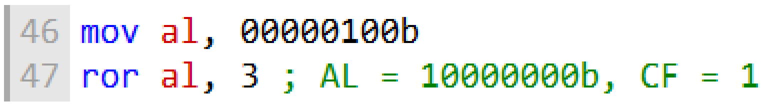
When you use the ROR instruction, each bit in the operand is shifted to the right. The lowest bit is copied into both the Carry flag (CF) and the highest bit position. Here's an example to illustrate this:



```
mov al, 01h ; AL = 00000001b
ror al, 1     ; AL = 10000000b, CF = 1
ror al, 1     ; AL = 01000000b, CF = 0
```

In this example, the initial value in AL is 00000001b, and after two rotations to the right, the lowest bit is copied into both the Carry flag (CF) and the highest bit position.

Multiple Rotations: When you perform multiple rotations with a count greater than 1, the Carry flag (CF) contains the last bit rotated out of the least significant bit (LSB) position. For example:



```
mov al, 00000100b
ror al, 3 ; AL = 10000000b, CF = 1
```

In this case, after three right rotations, the lowest bit (00000100b) is shifted to the right, and the Carry flag (CF) holds the last bit, which is 1.

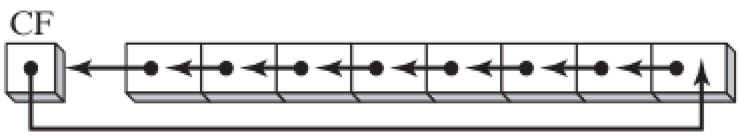
The ROR instruction is valuable for tasks that involve bitwise rotation to the right and is essential for various bit manipulation operations in assembly language programming.

===================================

RCL(Rotate Carry Left) Instruction

===================================

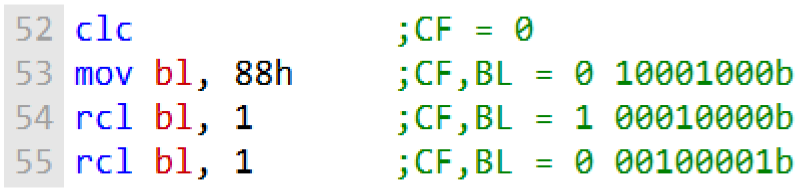
RCL (Rotate Carry Left) Instruction: The RCL instruction is used to perform bitwise rotation to the left, similar to the ROL instruction.



However, it has the additional feature of incorporating the Carry flag (CF) as an extra bit at the high end of the operand. Here's how it works:

- Each bit is shifted to the left, and the Carry flag is copied to the least significant bit (LSB).
- The most significant bit (MSB) is then copied into the Carry flag.

Example:



```
clc                 ;CF = 0
mov bl, 88h         ;CF,BL = 0 10001000b
rcl bl, 1           ;CF,BL = 1 00010000b
rcl bl, 1           ;CF,BL = 0 00100001b
```

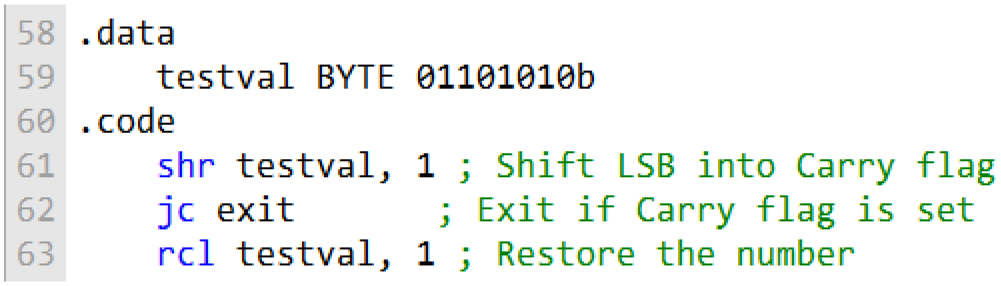
In this example, the initial value of BL is 10001000b, and after two RCL instructions, the bits are shifted to the left, with the CF acting as an extra bit and being involved in the rotation.

Recovering a Bit from the Carry Flag

RCL can be used to recover a bit that was previously shifted into the Carry flag.

In the following example, it checks the lowest bit of the "testval" variable by shifting its lowest bit into the Carry flag.

If the lowest bit is 1, a jump is taken; otherwise, RCL restores the number to its original value:



```
.data
    testval BYTE 01101010b
.code
    shr testval, 1 ; Shift LSB into Carry flag
    jc exit        ; Exit if Carry flag is set
    rcl testval, 1 ; Restore the number
```

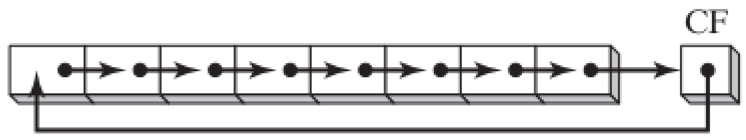
Here, the SHR instruction shifts the LSB into the Carry flag, and if it's 1, it takes a jump; otherwise, RCL restores the number.

=========================

RCR (Rotate Carry Right)

=========================

The RCR instruction is similar to RCL but rotates bits to the right. Here's how it operates:Each bit is shifted to the right, and the Carry flag is copied into the **most significant bit (MSB).**



The least significant bit (LSB) is copied into the Carry flag.

Example:

```
stc                 ;CF = 1
mov ah, 10h         ;AH, CF = 00010000 1
rcr ah, 1           ;AH, CF = 10001000 0
```

In this example, the STC instruction sets the Carry flag to 1, and then AH is rotated to the right, incorporating the Carry flag in the process.

These instructions are crucial for bit manipulation and advanced operations, allowing you to shift and rotate bits while incorporating the Carry flag for more intricate calculations in assembly language programming.

=========================

Visualizing the integer

=========================

When working with rotate and shift instructions, visualizing the integer as a 9-bit value can be helpful.

In this representation, the Carry flag is positioned to the right of the least significant bit (LSB), acting as an extra bit during the rotation.

Example: RCR (Rotate Carry Right)

In the following code example, the STC instruction sets the Carry flag to 1, and then a rotate carry right operation is performed on the AH register:

```
stc     ; CF = 1
mov ah, 10h ; AH, CF = 00010000 1
rcr ah, 1 ; AH, CF = 10001000 0
```

This code demonstrates how the Carry flag is used during the right rotation, influencing the resulting bits in AH.

=========================

Signed Overflow

=========================

The Overflow flag (OF) is set if the act of shifting or rotating a signed integer by one bit position generates a value outside the signed integer range of the destination operand.

In other words, this means that the sign of the number is reversed. Two examples illustrate this concept:

<u>Positive Integer Becoming Negative (ROL):</u>

A positive integer (+127) stored in an 8-bit register becomes negative (-2) when rotated left.

```
mov  al,+127                    ; AL = 01111111b
rol  al,1                       ; OF = 1, AL = 11111110b
```

Initially:

```
AL = 01111111b
```

After rotation:

```
OF = 1, AL = 11111110b
```

<u>Negative Integer Changing Sign (SHR):</u>

When -128 is shifted one position to the right, the Overflow flag is set.

The result in AL (+64) has the opposite sign.

```
mov  al,-128                    ; AL = 10000000b
shr  al,1                       ; OF = 1, AL = 01000000b
```

Initially:

```
AL = 10000000b
```

After shift:

```
OF = 1, AL = 01000000b
```

The Overflow flag is set when the signed integer range is exceeded during a shift or rotation by one bit position.

The **value of the Overflow flag** is undefined when the shift or rotation count is greater than 1.

It's important to note that the value of the Overflow flag is undefined when the shift or rotation count is greater than 1.

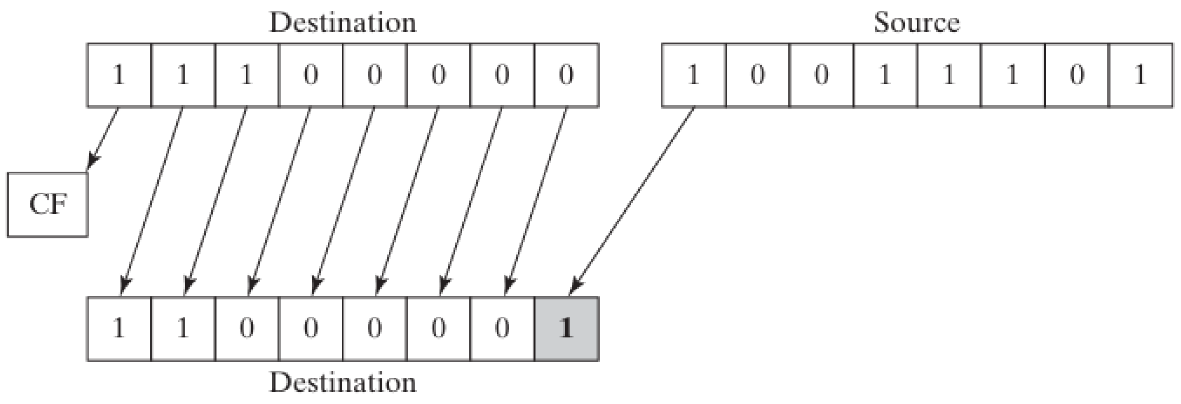
SHIFT LEFT DOUBLE AND SHIFT RIGHT DOUBLE

============================

SHLD (Shift Left Double)

============================

The **SHLD instruction** shifts a destination operand a given number of bits to the left, and fills the opened-up bit positions with the most significant bits of the source operand. The source operand is not affected.



The SHLD instruction has the following syntax:

```
SHLD dest, source, count
```

where:

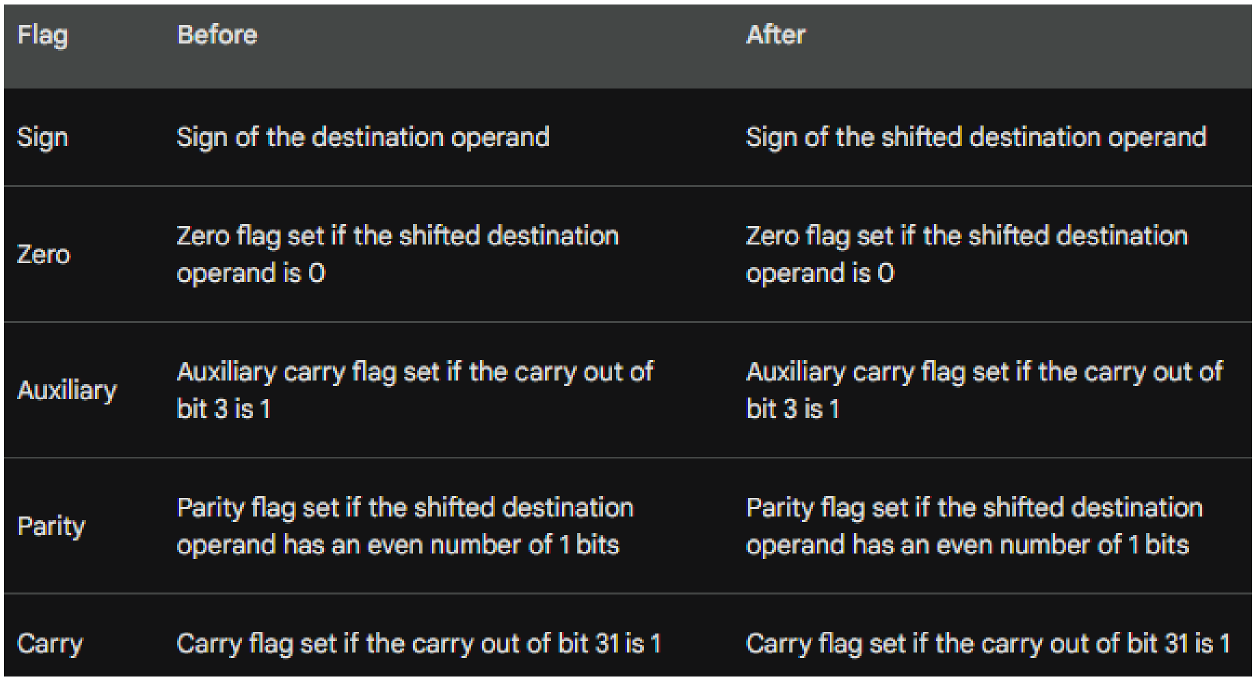
- dest is the destination operand.
- source is the source operand.
- count is the number of bits to shift.

The count operand must be a value between 0 and 31, inclusive.

If count is 0, the destination operand is not shifted.

If count is 31, the destination operand is shifted all the way to the left, and the source operand is copied into the destination operand.

The following table shows the effects of the SHLD instruction on the Sign, Zero, Auxiliary, Parity, and Carry flags:



Flag	Before	After
Sign	Sign of the destination operand	Sign of the shifted destination operand
Zero	Zero flag set if the shifted destination operand is 0	Zero flag set if the shifted destination operand is 0
Auxiliary	Auxiliary carry flag set if the carry out of bit 3 is 1	Auxiliary carry flag set if the carry out of bit 3 is 1
Parity	Parity flag set if the shifted destination operand has an even number of 1 bits	Parity flag set if the shifted destination operand has an even number of 1 bits
Carry	Carry flag set if the carry out of bit 31 is 1	Carry flag set if the carry out of bit 31 is 1

Here is an example of how to use the SHLD instruction:

```
mov eax, 0x12345678
mov ebx, 0xabcdef00
SHLD eax, ebx, 1
```

After the SHLD instruction, eax will contain the value 0x23456780.

The SHLD instruction can be used to perform a variety of tasks, such as:

- Shifting a value to the left to multiply it by a power of two.
- Shifting a value to the left to extract the most significant bits.
- Shifting a value to the left to prepare it for a bitwise operation.

============================

SHLD (Shift Right Double)

============================

The SHRD instruction shifts a destination operand a given number of bits to the right, and fills the opened-up bit positions with the least significant bits of the source operand. The source operand is not affected.

The SHRD instruction has the following syntax:

```
SHRD dest, source, count
```

where:

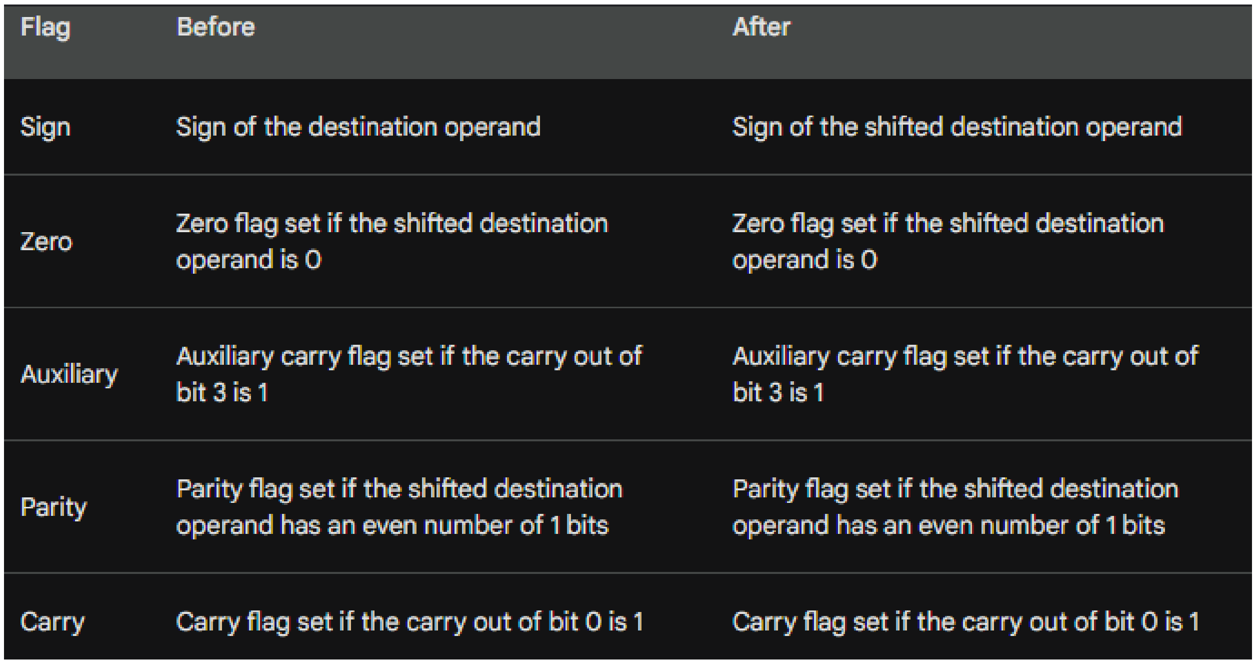
- dest is the destination operand.
- source is the source operand.
- count is the number of bits to shift.

The count operand must be a value between 0 and 31, inclusive.

If count is 0, the destination operand is not shifted.

If count is 31, the destination operand is shifted all the way to the right, and the source operand is copied into the destination operand.

The following table shows the effects of the SHRD instruction on the Sign, Zero, Auxiliary, Parity, and Carry flags:



Flag	Before	After
Sign	Sign of the destination operand	Sign of the shifted destination operand
Zero	Zero flag set if the shifted destination operand is 0	Zero flag set if the shifted destination operand is 0
Auxiliary	Auxiliary carry flag set if the carry out of bit 3 is 1	Auxiliary carry flag set if the carry out of bit 3 is 1
Parity	Parity flag set if the shifted destination operand has an even number of 1 bits	Parity flag set if the shifted destination operand has an even number of 1 bits
Carry	Carry flag set if the carry out of bit 0 is 1	Carry flag set if the carry out of bit 0 is 1

Here is an example of how to use the SHLD instruction:

```
mov eax, 0x12345678
mov ebx, 0xabcdef00
SHRD eax, ebx, 1

;After the SHRD instruction, eax will contain the value 0x092a3c40.
```

The SHRD instruction can be used to perform a variety of tasks, such as:

- Shifting a value to the right to divide it by a power of two.
- Shifting a value to the right to extract the least significant bits.
- Shifting a value to the right to prepare it for a bitwise operation.

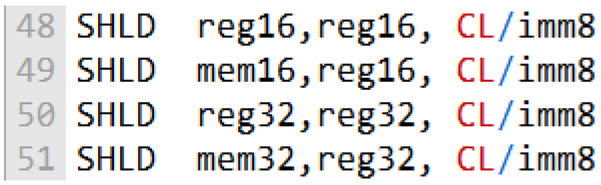
The SHRD instruction is a logical instruction that is used to shift a destination operand to the right by a specified number of bits, and then fills the vacated bit positions with the least significant bits of the source operand. The source operand is not affected by the instruction.

The SHRD instruction can be used to perform a variety of tasks, including:

- • • **Dividing a value by a power of two:** The SHRD instruction can be used to divide a value by a power of two by shifting the value to the right by the number of bits equal to the power of two. For example, to divide a value by 2, the value would be shifted to the right by 1 bit. To divide a value by 4, the value would be shifted to the right by 2 bits, and so on.
- • • **Extracting the least significant bits of a value:** The SHRD instruction can be used to extract the least significant bits of a value by shifting the value to the right by a number of bits equal to the number of least significant bits that need to be extracted. For example, to extract the least significant 4 bits of a value, the value would be shifted to the right by 4 bits.
- • • **Preparing a value for a bitwise operation:** The SHRD instruction can be used to prepare a value for a bitwise operation by shifting the value to the right by a number of bits equal to the number of bits that need to be aligned to the right. For example, to align a value to the right by 4 bits, the value would be shifted to the right by 4 bits.

The following instruction formats apply to both SHLD and SHRD. The destination operand

can be a register or memory operand, and the source operand must be a register. The count operand can be the CL register or an 8-bit immediate operand:



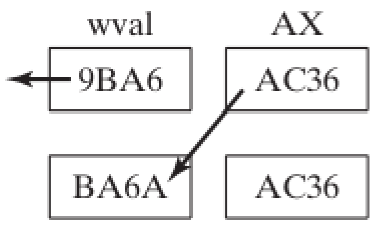
```
SHLD  reg16,reg16, CL/imm8
SHLD  mem16,reg16, CL/imm8
SHLD  reg32,reg32, CL/imm8
SHLD  mem32,reg32, CL/imm8
```

Example 1:

The following statements shift **wval** to the left 4 bits and insert the high 4 bits of AX into the low 4 bit positions of wval:

```
.data
    wval WORD 9BA6h
.code
    mov
    ax,0AC36h
    shld
    wval,ax,4        ;wval = BA6Ah
```

The data movement is shown in the following figure:

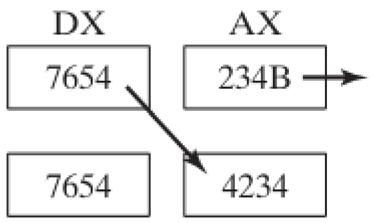


Example 2:

In the following example, AX is shifted to the right 4 bits, and the low 4 bits of DX are shifted into the high 4 positions of AX:

```
mov ax,234Bh
mov dx,7654h
shrd
ax,dx,4
```

The data movement is shown in the following figure:



Example 3:

The code below is an example of using the SHRD instruction to shift an array of doublewords to the right by 4 bits. This is a common operation in low-level programming when dealing with data manipulation, encryption, or even fast multiplication and division with very long integers. Let's break down the code and explain it step by step:

```
.data
    array DWORD 648B2165h,8C943A29h,6DFA4B86h,91F76C04h,8BAF9857h
.code
    mov bl,4
    ; shift count
    mov esi,OFFSET array
    ; offset of the array
    mov ecx,(LENGTHOF array) - 1
    ; number of array elements

L1:
    push ecx
    ; save loop counter
    mov eax,[esi + TYPE DWORD]
    mov cl,bl
    ; shift count
    shrd [esi],eax,cl
    ; shift EAX into high bits of [ESI]
    add esi,TYPE DWORD
    ; point to the next doubleword pair
    pop ecx
    ; restore loop counter
    loop L1
    shr DWORD PTR [esi],COUNT
    ; shift the last doubleword
```

In the .data section, you define an array of doublewords (DWORD). Each doubleword contains a hexadecimal value. This array represents the data that you want to shift.

In the .code section, you set up some initial values:

bl is set to 4, which represents the shift count (you want to shift the data by 4 bits).

esi is loaded with the offset of the array.

ecx is set to the number of elements in the array minus 1 (using LENGTHOF array - 1). This **loop counter** will be used to iterate through the array.

The core of the code is a loop labeled L1. Here's what happens inside the loop:

push ecx saves the loop counter on the stack.

mov eax, [esi + TYPE DWORD] loads the doubleword at the current esi offset into the eax register.

mov cl, bl loads the shift count into the cl register.

shrd [esi], eax, cl performs the shift operation, shifting the value in eax to the right by the number of bits specified in cl.

This result is stored back in the memory location pointed to by esi.

add esi, TYPE DWORD moves esi to the next doubleword in the array.

pop ecx restores the loop counter.

loop L1 decrements ecx (the loop counter) and jumps back to L1 as long as ecx is not zero. This loop processes each doubleword in the array.

Finally, after the loop, you perform a shift on the last doubleword using the shr instruction.

This code effectively shifts the entire array of doublewords to the right by 4 bits.

===============

Questions

===============

Which instruction shifts each bit in an operand to the left and copies the highest bit into both the Carry flag and the lowest bit position?

Answer: The instruction that performs this operation is SHL (Shift Left) or SAL (Shift Arithmetic Left).

Which instruction shifts each bit to the right, copies the lowest bit into the Carry flag, and copies the Carry flag into the highest bit position?

Answer: The instruction that shifts each bit to the right, copies the lowest bit into the Carry flag, and copies the Carry flag into the highest bit position is SHR (Shift Right).

Which instruction performs the following operation (CF = Carry flag)?

Answer: The operation you described is performed by the RCL (Rotate through Carry Left) instruction. It rotates the bits to the left through the Carry flag, as indicated.

What happens to the Carry flag when the SHR AX,1 instruction is executed?

Answer: When the SHR AX,1 instruction is executed, the Carry flag (CF) will receive the value of the least significant bit (LSB) of the AX register before the shift, and the LSB itself will be shifted out of the AX register.

Challenge: Write a series of instructions that shift the lowest bit of AX into the highest bit of BX without using the SHRD instruction. Next, perform the same operation using SHRD.

Answer: Below are the series of instructions to achieve this without SHRD:

```
mov cx, 1       ; Set the shift count to 1
shl bx, 1       ; Shift left the bits in BX by 1 position
rcl bx, 1       ; Rotate through Carry Left (this moves the original LSB of AX to the MSB of BX)
```

To perform the same operation using SHRD:

```
shrd bx, ax, 1  ; Shift right double (moves LSB of AX to the MSB of BX)
```

Challenge: One way to calculate the parity of a 32-bit number in EAX is to use a loop that shifts each bit into the Carry flag and accumulates a count of the number of times the Carry flag was set. Write a code that does this, and set the Parity flag accordingly.

Answer:

```
xor ecx, ecx      ; Clear the counter
mov ebx, eax      ; Make a copy of EAX
parity_loop:
shr ebx, 1        ; Shift right by 1 bit
adc ecx, 0        ; Add the Carry flag to the counter
jnz parity_loop   ; Jump back if there's still a 1 bit in EBX
test ecx, 1       ; Test the least significant bit of the counter
setp al           ; Set the Parity flag based on the counter
```

This code calculates the parity of a 32-bit number in EAX using a loop that shifts each bit into the Carry flag and accumulates a count of the number of times the Carry flag was set. It sets the Parity flag (PF) accordingly. If the count is even, PF will be set; if it's odd, PF will be cleared.

SHIFTING MULTIPLE DOUBLEWORDS

================================

Shifting Multiple Doublewords

================================

In the realm of assembly programming, you can manipulate extended-precision integers that are organized into arrays of bytes, words, or doublewords.

However, it's imperative to understand how these array elements are stored.

A prevalent method of storing these integers is referred to as "little-endian order."

In little-endian order, the low-order byte is placed at the array's starting address, and then, as you progress from this byte to the high-order byte, each is consecutively stored in the next memory location.

This ordering holds true regardless of whether you're working with bytes, words, or doublewords because x86 machines consistently use little-endian order for all these data formats.

Now, let's delve into the specific steps for shifting an array of bytes one bit to the right:

Step 1:

To accomplish this operation, you start by shifting the highest byte located at [ESI+2] to the right.

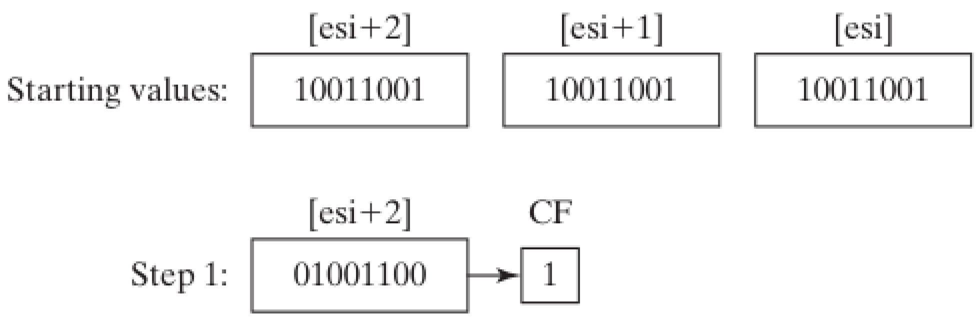
During this shift, the lowest bit of this byte is automatically copied into the Carry flag.

This behavior is a standard operation performed by instructions like SHR (Shift Right) in many assembly languages.

To visually demonstrate this, here's how you might express it in x86 assembly code, assuming that the ESI register holds the base address of the array:

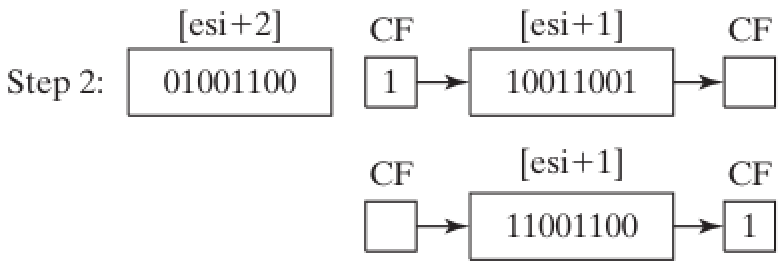
```
SHR byte ptr [ESI+2], 1
```

This instruction effectively shifts the byte at [ESI+2] one bit to the right, with the least significant bit being transferred to the Carry flag.



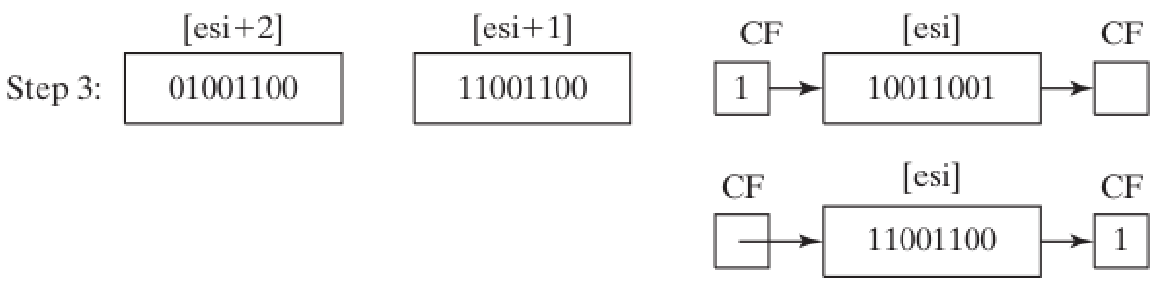
Step 2:

Rotate the value at [ESI+1] to the right, filling the highest bit with the value of the Carry flag, and shifting the lowest bit into the Carry flag:

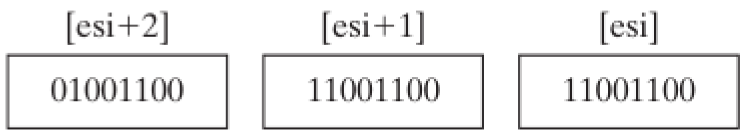


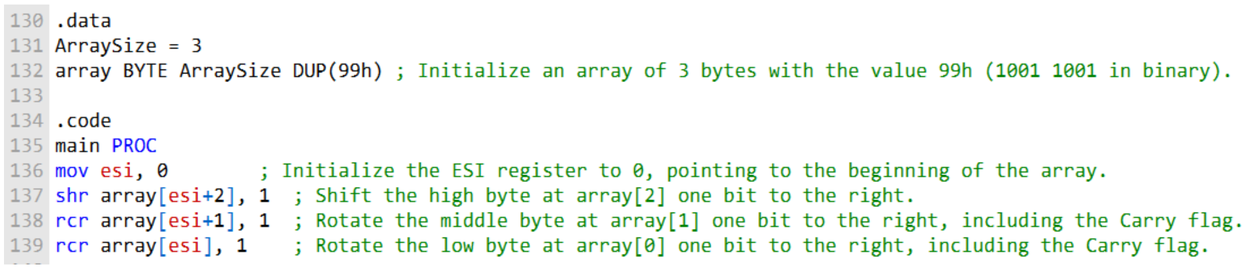
Step 3:

Rotate the value at [ESI] to the right, filling the highest bit with the value of the Carry flag, and shifting the lowest bit into the Carry flag:



After Step 3 is complete, all bits have been shifted 1 position to the right:





```
.data
ArraySize = 3
array BYTE ArraySize DUP(99h) ; Initialize an array of 3 bytes with the value 99h (1001 1001 in binary).

.code
main PROC
mov esi, 0          ; Initialize the ESI register to 0, pointing to the beginning of the array.
shr array[esi+2], 1  ; Shift the high byte at array[2] one bit to the right.
rcr array[esi+1], 1  ; Rotate the middle byte at array[1] one bit to the right, including the Carry flag.
rcr array[esi], 1    ; Rotate the low byte at array[0] one bit to the right, including the Carry flag.
```

Here's a breakdown of what's happening in this code:

The **.data section** defines the ArraySize as 3 and initializes an array called array with three bytes, each containing the value 99h. This means the binary representation of each byte is "1001 1001."

In the **.code section**, the main procedure begins.

mov esi, 0 initializes the ESI register to 0, pointing to the first byte in the array.

shr array[esi+2], 1 performs a right shift on the high byte at array[2]. This operation moves the most significant bit one position to the right, effectively shifting the entire byte to the right by one bit.

rcr array[esi+1], 1 rotates the middle byte at array[1] one bit to the right. A rotate operation combines shifting with a rotation of the Carry flag. It effectively moves the least significant bit of the high byte into the least significant bit of the middle byte, and the Carry flag into the most significant bit of the high byte.

Similarly, **rcr array[esi], 1** rotates the low byte at array[0], including the Carry flag. This rotates the bytes and carries the bit that was shifted out from the middle byte into the low byte.

As a result, this code has shifted the entire array of bytes by one bit to the right, and the Carry flag holds the value of the bit that was shifted out from the low byte.

The code can indeed be adapted to handle arrays of words or doublewords by changing the data types and adjusting the number of bytes processed. Additionally, by using a loop, you can shift arrays of arbitrary size efficiently.

BINARY SHIFTING AND MULTIPLICATION

Let's discuss the concept of binary multiplication in assembly language, specifically focusing on optimizing integer multiplication through bit shifting rather than using the MUL instruction.

Bit Shifting for Multiplication:

In assembly programming, the SHL (Shift Left) instruction is often used to perform unsigned multiplication when the multiplier is a power of 2.

Shifting an unsigned integer left by 'n' bits is equivalent to multiplying it by 2^n.

This is a highly efficient way to perform multiplication when the multiplier is a power of 2 because it can be achieved through simple bit shifting operations, which are faster than a full multiplication.

Representing Non-Power-of-2 Multipliers:

For multipliers that are not powers of 2, you can express them as a sum of powers of 2. This allows you to break down the multiplication into simpler bit-shifting operations.

For example, to multiply a value in the EAX register by 36, you can represent 36 as $2^5 + 2^2$ and apply the distributive property of multiplication as follows:

```
EAX * 36 = EAX * (2^5 + 2^2)
         = EAX * (32 + 4)
         = (EAX * 32) + (EAX * 4)
```

This decomposition enables you to perform two separate bit-shifting operations: one to multiply by 32 (shifting left by 5 bits) and another to multiply by 4 (shifting left by 2 bits). These individual results can then be added to obtain the final product.

Example: 123 * 36:

The passage mentions an example of multiplying 123 by 36, resulting in 4428. This multiplication can be achieved using the approach described above. In this case, you would perform two separate bit-shifting operations.

```
                 0 1 1 1 1 0 1 1         123
         ×       0 0 1 0 0 1 0 0         36
         ------------------------
              0 1 1 1 1 0 1 1            123 SHL 2
  +       0 1 1 1 1 0 1 1                123 SHL 5
  -------------------------------
   0 0 0 1 0 0 0 1 0 1 0 0 1 1 0 0       4428
```

```
123 * 32 (2^5) = 3936
123 * 4 (2^2) = 492
```

```
mov     eax, 123
mov     ebx, eax
shl     eax, 5      ; multiply by 2^5
shl     ebx, 2      ; multiply by 2^2
add     eax, ebx
```

This code snippet first moves the value 123 into the register EAX.

Then, it moves a copy of EAX into the register EBX.

Next, it shifts EAX left by 5 bits, which is equivalent to multiplying it by 25.

It then shifts EBX left by 2 bits, which is equivalent to multiplying it by 22.

Finally, it adds the two products together in the register EAX.

To generalize this example and create a procedure that multiplies any two 32-bit unsigned integers using shifting and addition, we can use the following algorithm:

- **• • Initialize the product register to 0.**
- **For each bit in the multiplier, starting with the least significant bit:**
- **If the bit is set, shift the multiplicand left by the corresponding number of bits and add it to the product register.**
- **If the bit is not set, do nothing.**
- **Return the product register.**

The following pseudocode shows this algorithm:

```c
#include <stdio.h>

int multiply(int multiplicand, int multiplier) {
    int product = 0;

    for (int i = 0; i < 32; i++) {
        if ((multiplier & (1 << i)) != 0) {
            product += (multiplicand << i);
        }
    }

    return product;
}

int main() {
    int multiplicand = 123;  // Replace with your values
    int multiplier = 36;    // Replace with your values

    int result = multiply(multiplicand, multiplier);

    printf("Result: %d\n", result);
    return 0;
}
```

We define a function multiply that takes two integers, multiplicand and multiplier, as parameters and returns the product as an integer.

Inside the function, we initialize product to 0, which will hold the result.

We use a for loop to iterate from 0 to 31 to examine each bit of the multiplier.

Within the loop, we use bitwise operations to check if the i-th bit of the multiplier is set (1).

If it is, we shift the multiplicand left by i bits and add the result to the product.

Finally, the function returns the computed product.

In the main function, you can replace the values of multiplicand and multiplier with the numbers you want to multiply. When you run the program, it will calculate the product and print the result.

The following assembly code implements this algorithm:

```
multiply:
  push   ebp
  mov    ebp, esp
  mov    eax, multiplicand
  mov    ebx, multiplier
  xor    ecx, ecx
  mov    ecx, 31
loop:
  shl    eax, 1
  test   ebx, 1
  jz     next
  add    eax, ebx
next:
  dec    ecx
  jnz    loop
  mov    esp, ebp
  pop    ebp
  ret
```

This assembly code appears to multiply two integers, multiplicand and multiplier, and the result is stored in the EAX register, which is a common convention for returning values in assembly language functions. Let's break down the code:

push ebp: This instruction saves the base pointer (EBP) on the stack to establish a new stack frame. This is a common practice at the beginning of a function.

mov ebp, esp: It sets the base pointer (EBP) to the current stack pointer (ESP). This establishes a new stack frame for the function.

mov eax, multiplicand: This instruction loads the value of multiplicand into the EAX register.

mov ebx, multiplier: It loads the value of multiplier into the EBX register.

xor ecx, ecx: This clears the ECX register to zero. It will be used as a loop counter.

mov ecx, 31: This sets ECX to 31, which is the loop iteration count. The loop will iterate for 32 bits (0 to 31).

loop:: This is a label for the loop.

shl eax, 1: It shifts the EAX register left by 1 bit, effectively multiplying it by 2. This is performed in each iteration to handle the next bit of the multiplier.

test ebx, 1: This instruction tests the least significant bit of EBX (the multiplier) to check if it's set (1).

jz next: If the least significant bit of the multiplier is not set (i.e., it's zero), the code jumps to the next label without adding EBX to the result in EAX.

add eax, ebx: If the least significant bit of the multiplier is set, it adds EBX to EAX, effectively performing an addition in each iteration.

next:: This label is used to continue with the next iteration of the loop.

dec ecx: It decrements the loop counter in ECX.

jnz loop: This instruction checks if ECX is not zero (meaning there are more bits to process in the multiplier). If it's not zero, the code jumps back to the loop label, continuing the multiplication process.

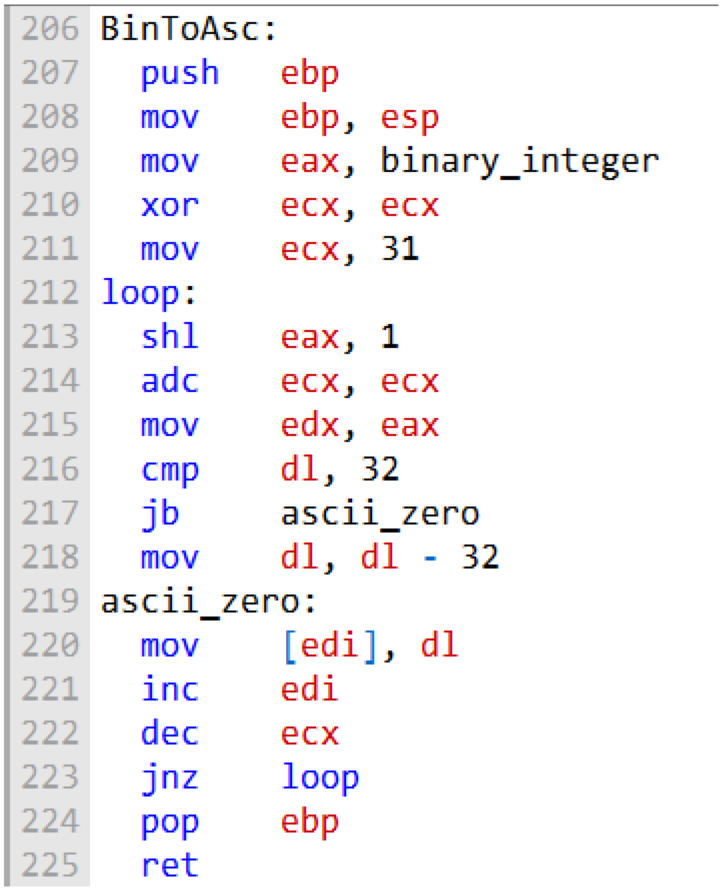
mov esp, ebp: This restores the stack pointer (ESP) to its previous value, effectively cleaning up the stack frame.

pop ebp: It restores the base pointer (EBP) to its previous value, completing the stack frame cleanup.

ret: This is the return instruction, and it returns the result in the EAX register.

Overall, the code is an assembly implementation of integer multiplication using bit-shifting and addition, and it follows a loop-based approach to handle each bit of the multiplier.

BINTOASC



```
BinToAsc:
   push    ebp
   mov     ebp, esp
   mov     eax, binary_integer
   xor     ecx, ecx
   mov     ecx, 31
loop:
   shl     eax, 1
   adc     ecx, ecx
   mov     edx, eax
   cmp     dl, 32
   jb      ascii_zero
   mov     dl, dl - 32
ascii_zero:
   mov     [edi], dl
   inc     edi
   dec     ecx
   jnz     loop
   pop     ebp
   ret
```

This procedure works by iterating over the bits in the binary integer, starting with the most significant bit.

For each bit, the procedure shifts the binary integer left by 1 bit and adds the carry flag to the counter register (ECX).

The carry flag is used to keep track of whether the previous iteration resulted in a carry-out.

If the binary integer is less than 32, then the least significant bit will be 0 and the carry flag will be 0. In this case, the procedure will move the ASCII character '0' (0x30) to the buffer at the address specified by the register EDI.

If the binary integer is greater than or equal to 32, then the least significant bit will be 1 and the carry flag will be 1.

In this case, the procedure will move the ASCII character '1' (0x31) to the buffer at the address specified by the register EDI.

After the procedure has finished iterating over the bits in the binary integer, the buffer at the address specified by the register EDI will contain the ASCII binary string representation of the binary integer.

Example 2 Usage

:

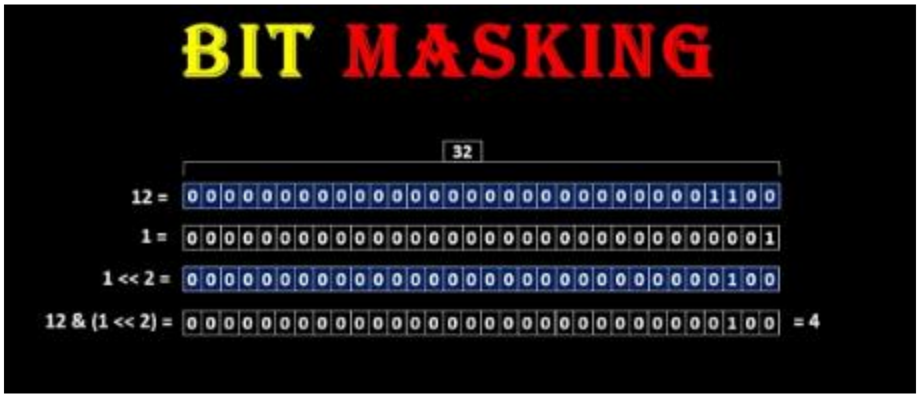
The following code snippet shows how to use the BinToAsc procedure to convert the binary integer 123 (01111011) to an ASCII binary string:

```
mov      eax, 123
call    BinToAsc

;The buffer at the address specified by the register `EDI`
;will now contain the ASCII binary string "01111011".
```

The BinToAsc procedure is a simple and efficient way to convert a binary integer to an ASCII binary string. It is useful for displaying binary data on the console or in a file.

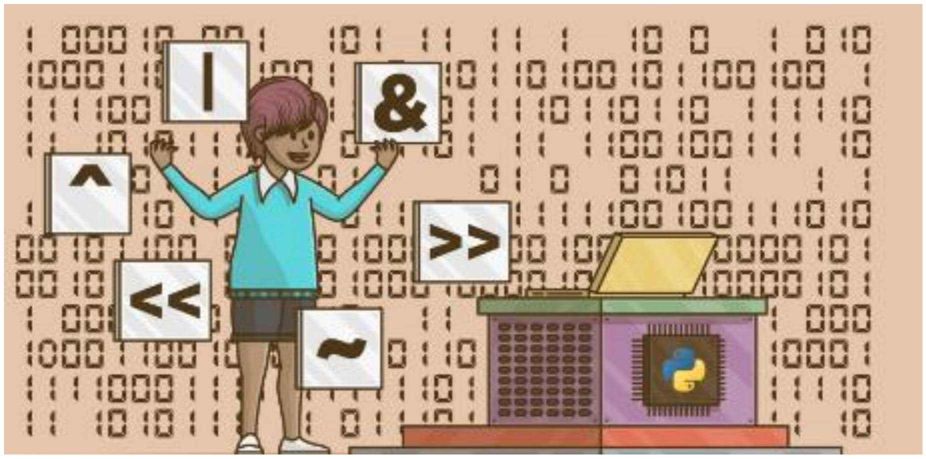
EXTRACTING FILE DATE FIELDS

Shifting and masking: The two most important operations used to extract bit strings are shifting and masking.



Shifting allows you to move the bit string to the desired position within a register, while **masking** allows you to clear any unwanted bits.

Using the AX register: The AX register is a convenient register to use for extracting bit strings, as it is 16 bits wide. This means that it can hold two 8-bit byte values.



This can be useful for extracting bit strings that are spread across two bytes, such as the month and day fields of a date stamp.

Storing the extracted bit strings: Once you have extracted the bit strings, you need to store them somewhere.

This can be done by copying them to other registers or to memory.

The following code snippet shows how to extract the day, month, and year fields of a date stamp integer stored in the DX register:

```
;Make a copy of DL and mask off bits not belonging to the day field.
mov     al, dl
and     al, 00011111b
mov     day, al

;Shift bits 5 through 8 into the low part of AL before masking off all other bits.
mov     ax, dx
shr     ax, 5
and     al, 00001111b
mov     month, al

;Copy the year field from DH to AL and shift right by 1 bit to clear AH.
mov     al, dh
shr     al, 1
mov     ah, 0
add     ax, 1980
mov     year, ax
```

This code snippet first makes a copy of the DL register to the AL register. Then, it masks off all bits except for the day field (bits 0 through 4). Finally, it copies the masked value to the day variable.

Next, the code snippet shifts bits 5 through 8 of the DX register into the low part of the AX register. Then, it masks off all bits except for the month field (bits 5 through 8). Finally, it copies the masked value to the month variable.

Finally, the code snippet copies the year field (bits 9 through 15) from the DH register to the AL register. Then, it shifts the value right by 1 bit to clear the AH register.

Finally, it adds 1980 to the value to account for the fact that the year field is relative to 1980. The code snippet then copies the final value to the year variable.

Once the day, month, and year fields have been extracted, they can be used for any purpose, such as displaying the date or calculating the number of days since the file was last modified.

----

1. **Write assembly language instructions that calculate EAX * 24 using binary multiplication.**

Here's how you can calculate **EAX * 24** in assembly language using binary multiplication:

```
mov ecx, 4       ; Initialize a counter for the number of bits to shift
mov ebx, eax     ; Make a copy of the original value in EAX
shl eax, 3       ; Multiply EAX by 2^3 (which is 8)
add eax, ebx     ; Multiply the result by 3 (24 = 8 * 3)
```

1. **Write assembly language instructions that calculate EAX * 21 using binary multiplication.**

```
Hint: 21 = 24 - 22 - 20.
```

To calculate EAX * 21, you can use binary multiplication based on the hint provided:

```
mov ebx, eax     ; Copy the original value to EBX
shl eax, 3       ; Multiply EAX by 8 (2^3)
sub ebx, eax     ; Subtract the original value by the result (EBX - EAX)
shl eax, 1       ; Multiply EAX by 2 (2^1)
add eax, ebx     ; Add the result to the previous result (EAX + EBX)
```

1. **What change would you make to the BinToAsc procedure in Section 7.2.3 in order to display the binary bits in reverse order?**

To display the binary bits in reverse order in the BinToAsc procedure, you can modify the loop that processes the bits. Instead of starting from the most significant bit (bit 31) and moving towards the least significant bit (bit 0), you can reverse the loop to start from the least significant bit and move towards the most significant bit. Here's a modified version of the BinToAsc procedure:

```
BinToAsc PROC
    pushad                  ; Preserve registers
    mov     edi, 31         ; Start from the least significant bit
    mov     ecx, 32         ; Loop through all 32 bits
    mov     esi, OFFSET outputStr ; Address of the output buffer

ConvertLoop:
    mov     al, [ebx + edi/8]   ; Load a byte from the binary data
    shl     al, cl              ; Shift the bit of interest to the lowest position
    and     al, 1               ; Mask all bits except the lowest one
    add     al, '0'             ; Convert the bit to its ASCII representation
    stosb                       ; Store the character in the output buffer
    loop    ConvertLoop
    mov     byte ptr [esi], 0   ; Null-terminate the output string
    popad                   ; Restore registers
    ret
BinToAsc ENDP
```

In this modified version, we start with the least significant bit (bit 0) and iterate through the bits in reverse order, which will display the binary bits in reverse.

1. **The time stamp field of a file directory entry uses bits 0 through 4 for the seconds, bits 5 through 10 for the minutes, and bits 11 through 15 for the hours. Write instructions that extract the minutes and copy the value to a byte variable named bMinutes.**

Here are the assembly instructions to extract the minutes from the time stamp and store the value in a byte variable named bMinutes:

```
mov     edx, [DirectoryEntryTime] ; Load the directory entry time stamp (assuming it's in edx)
and     edx, 0x07E0               ; Mask out the bits for minutes (5 through 10)
shr     edx, 5                    ; Shift the extracted minutes to the least significant bits
mov     byte ptr [bMinutes], dl   ; Store the extracted minutes in bMinutes
```

In this code, we use the and and shr instructions to isolate and shift the bits representing the minutes in the directory entry time stamp.

Finally, we store the extracted minutes in the bMinutes byte variable. Please replace [DirectoryEntryTime] with the actual address of the time stamp in your program.

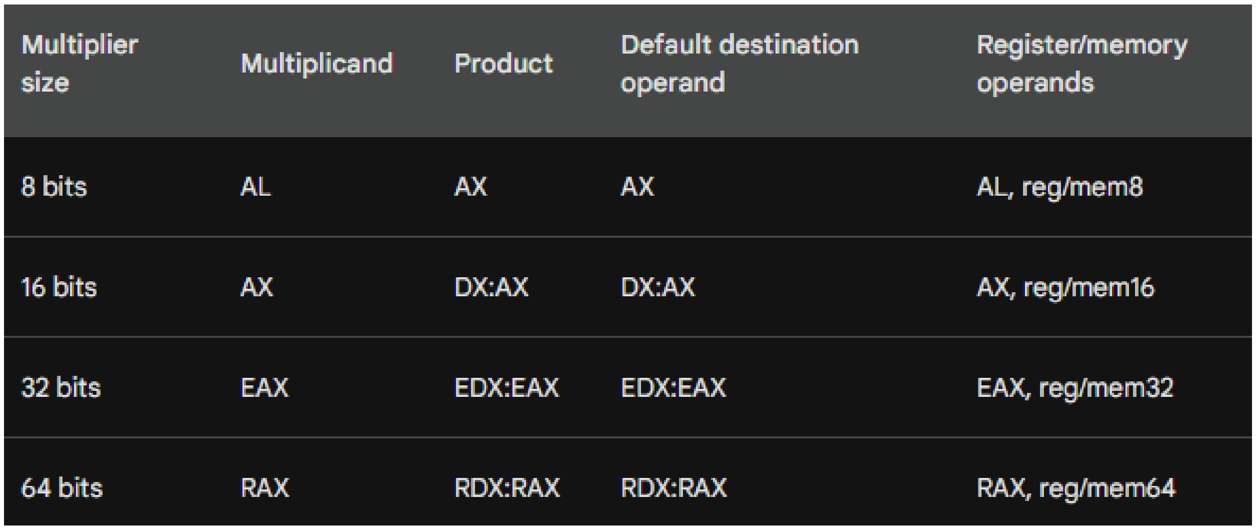
MUL OPERATOR

The MUL instruction performs unsigned integer multiplication.

It has three versions, which multiply an 8-bit operand by AL, a 16-bit operand by AX, or a 32-bit operand by EAX.

The multiplicand and multiplier must always be the same size, and the product is twice their size.

The following table shows the default multiplicand and product, depending on the size of the multiplier:



Multiplier size	Multiplicand	Product	Default destination operand	Register/memory operands
8 bits	AL	AX	AX	AL, reg/mem8
16 bits	AX	DX:AX	DX:AX	AX, reg/mem16
32 bits	EAX	EDX:EAX	EDX:EAX	EAX, reg/mem32
64 bits	RAX	RDX:RAX	RDX:RAX	RAX, reg/mem64

Because the destination operand is twice the size of the multiplicand and multiplier, overflow cannot occur.

However, the MUL instruction sets the Carry and Overflow flags if the upper half of the product is not equal to zero.

The Carry flag is ordinarily used for unsigned arithmetic, so it can be used to detect overflow in the MUL instruction.

For example, if AX is multiplied by a 16-bit operand, the product is stored in the combined DX and AX registers.

If DX is not equal to zero after the multiplication operation, then the product will not fit into the lower half of the implied destination operand, and the Carry flag will be set.

Here is an example of how to use the MUL instruction to multiply two 16-bit operands:

```
mov ax, 1000h ; load first operand into AX
mov bx, 2000h ; load second operand into BX
mul bx ; multiply AX by BX
```

After the multiplication operation, the product will be stored in the combined DX and AX registers. If DX is not equal to zero, then the product will not fit into the lower half of the AX register, and the Carry flag will be set.

As you can see, the MUL instruction supports register and memory operands for all multiplier sizes. This gives you a lot of flexibility in how you use the instruction.

For example, the following assembly code multiplies the AL register by the 8-bit operand in memory location MY_DATA:

```
mov al, 100h ; load first operand into AL
mul MY_DATA ; multiply AL by the operand in MY_DATA
```

The following assembly code multiplies the EAX register by the 32-bit operand in memory location MY_DATA:

```
mov eax, 10000000h ; load first operand into EAX
mul MY_DATA ; multiply EAX by the operand in MY_DATA
```

The MUL instruction is a powerful tool for performing unsigned integer multiplication on the x86 architecture. It is important to understand the different versions of the instruction and how to use the Carry flag to detect overflow.

A good reason for checking the Carry flag after executing MUL is to know whether the upper half of the product can safely be ignored.

The MUL instruction multiplies two operands and stores the product in two registers. If the product is too large to fit in the destination registers, the Carry flag is set.

For example, if you multiply two 16-bit operands, the product will be 32 bits.

The MUL instruction will store the lower 16 bits of the product in the AX register and the upper 16 bits of the product in the DX register.

If the upper 16 bits of the product are zero, then you can safely ignore them. However, if the upper 16 bits of the product are non-zero, then you will need to use the DX register to store the entire product.

You can check the Carry flag to determine whether the upper half of the product is zero.

If the Carry flag is clear, then the upper half of the product is zero and you can safely ignore it.

However, if the Carry flag is set, then the upper half of the product is non-zero and you will need to use the DX register to store the entire product.

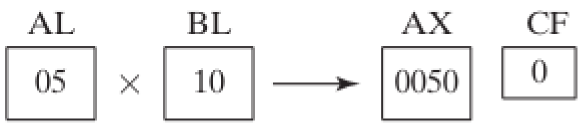
Here is an example of how to check the Carry flag after executing MUL:

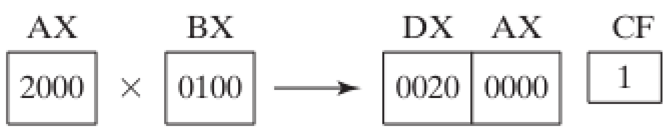
```
mov ax, 1000h   ;load first operand into AX
mov bx, 2000h   ;load second operand into BX
mul bx ; multiply AX by BX

;check the Carry flag
jc overflow     ;jump to overflow handler if the Carry flag is set

;the upper half of the product is zero, so we can ignore it
;use the AX register to store the lower half of the product
```

Example 1:

```
; 8-bit multiplication
mov al, 5h
mov bl, 10h
mul bl
; AX = 0050h, CF = 0

; 16-bit multiplication
mov ax, 2000h
mul val2
; DX:AX = 00200000h, CF = 1
```

For this code, this is the data flow:





The following statements multiply 12345h by 1000h, producing a 64-bit product in the combined EDX and EAX registers:

```
mov eax, 12345h
mov ebx, 1000h
mul ebx
; EDX:EAX = 0000000012345000h, CF = 0
```

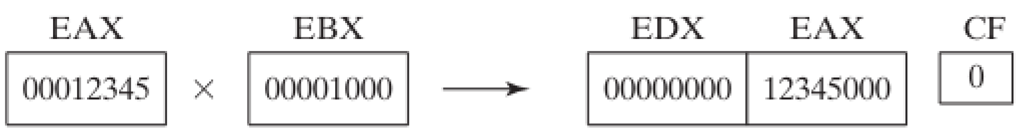
The MUL instruction multiplies two unsigned integers and stores the product in two registers: the low-order half of the product is stored in the EAX register, and the high-order half of the product is stored in the EDX register.

The Carry flag is set if the product is too large to fit in the destination registers.

In this case, the product of 12345h and 1000h is 12345000h, which is a 64-bit value. The product fits in the EDX and EAX registers, so the Carry flag is clear.

The following diagram illustrates the movement between registers:

```
Before:
EAX = 12345h
EBX = 1000h
EDX = 0

After:
EAX = 0000h
EBX = 1000h
EDX = 12345h
```



Here is a summary of the MUL instruction:

- The MUL instruction multiplies two unsigned integers and stores the product in two registers.

- The multiplicand (the first operand) is stored in the AL register (for 8-bit multiplication) or the AX register (for 16-bit multiplication).
- The multiplier (the second operand) is stored in another register or in memory.
- The product is stored in two registers: the low-order half of the product is stored in the AL register (or the AX register for 16-bit multiplication), and the high-order half of the product is stored in the AH register (or the DX register for 16-bit multiplication).
- The Carry flag is set if the product is too large to fit in the destination registers.

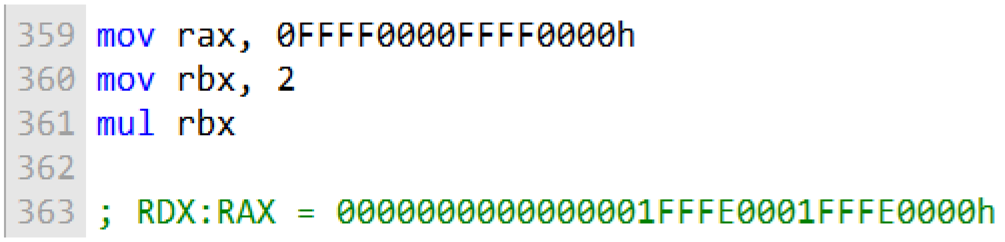
====================

MUL in 64-bit mode

=====================

In 64-bit mode, the MUL instruction can be used to multiply two 64-bit operands. The result is a 128-bit product, which is stored in the RDX:RAX register pair.

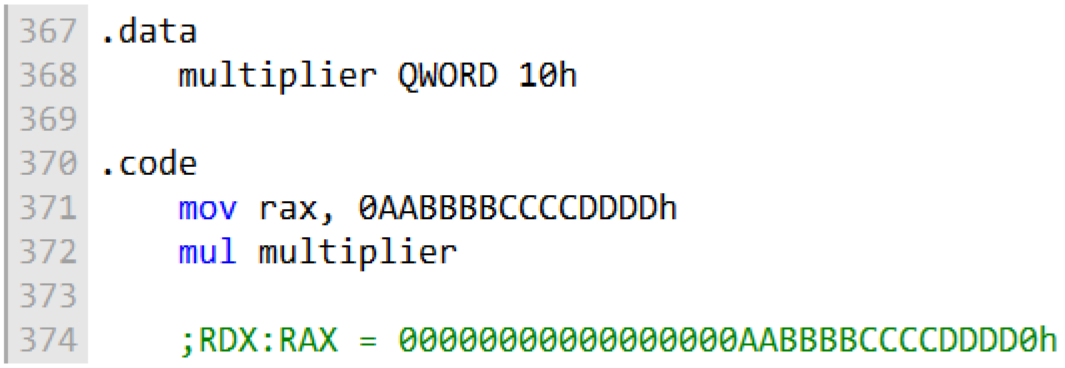
The following example shows how to use the MUL instruction to multiply RAX by 2:



```
mov rax, 0FFFF0000FFFF0000h
mov rbx, 2
mul rbx

; RDX:RAX = 0000000000000001FFFE0001FFFE0000h
```

In this example, the highest bit of RAX spills over into the RDX register because the product is too large to fit in a 64-bit register.

The following example shows how to use the MUL instruction to multiply RAX by a 64-bit memory operand:



```
.data
    multiplier QWORD 10h

.code
    mov rax, 0AABBBBCCCCDDDDh
    mul multiplier

    ;RDX:RAX = 00000000000000000AABBBBCCCCDDDD0h
```

In this example, the product is a **128-bit value**, but both halves of the product fit in RAX and RDX because the product is less than **2^{128}.**

Here is a more in-depth explanation of what happens when the MUL instruction is used in 64-bit mode:

The RAX and RDX registers are multiplied together. The low-order 64 bits of the product are stored in RAX. The high-order 64 bits of the product are stored in RDX. The Carry flag is set if the product is too large to fit in RAX and RDX.

IMUL OPERATOR

The IMUL instruction performs signed integer multiplication.

This means that it multiplies two integers and takes into account their signs. Unlike the MUL instruction, the IMUL instruction preserves the sign of the product.

How does the IMUL instruction work?

The IMUL instruction works by first sign extending the highest bit of the lower half of the product into the upper bits of the product.

This ensures that the sign of the product is the same as the sign of the multiplicand (the number being multiplied).

What are the different formats of the IMUL instruction?

The IMUL instruction has three formats:

```
One-operand format
Two-operand format
Three-operand format
```

The **one-operand format of the IMUL instruction** multiplies the operand by itself and stores the product in the same operand. This is equivalent to squaring the operand.

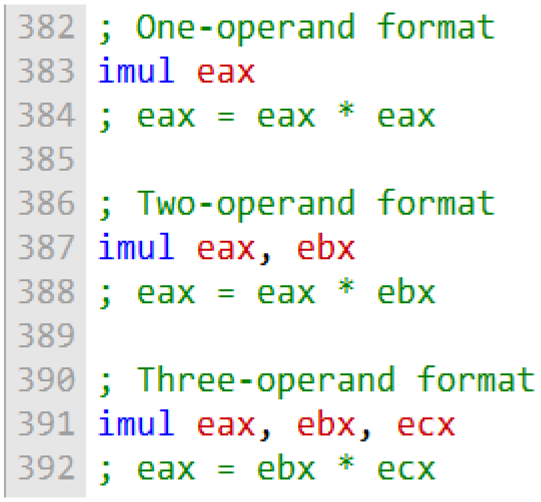
The **two-operand format of the IMUL instruction** multiplies the two operands and stores the product in the first operand. The second operand can be a register, a memory operand, or an immediate value.

The **three-operand format of the IMUL instruction** multiplies the first and third operands and stores the product in the second operand. The first and third operands can be registers or memory operands, and the second operand must be a register.

When should I use the IMUL instruction?

You should use the IMUL instruction whenever you need to perform signed integer multiplication. This is especially important when you need to preserve the sign of the product.

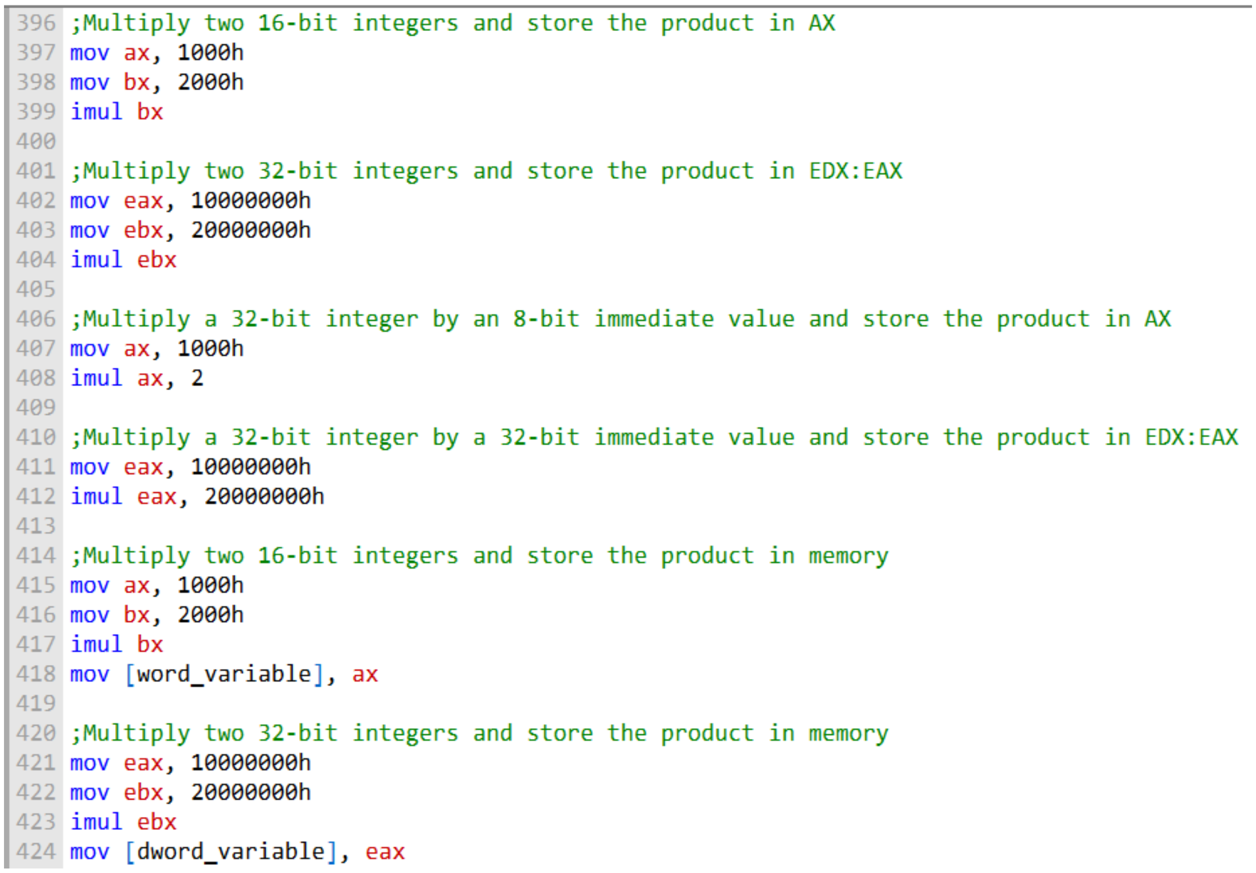
Here are some examples of how to use the IMUL instruction:



```
; One-operand format
imul eax
; eax = eax * eax

; Two-operand format
imul eax, ebx
; eax = eax * ebx

; Three-operand format
imul eax, ebx, ecx
; eax = ebx * ecx
```

Important things to keep in mind:

- The IMUL instruction can generate overflow. If the product of the two operands is too large to fit in the destination operand, the Overflow flag is set.
- The IMUL instruction can also generate a carry. If the highest bit of the product is set, the Carry flag is set.
- It is important to check the Overflow and Carry flags after performing an IMUL operation to ensure that the result is correct.



```
;Multiply two 16-bit integers and store the product in AX
mov ax, 1000h
mov bx, 2000h
imul bx

;Multiply two 32-bit integers and store the product in EDX:EAX
mov eax, 10000000h
mov ebx, 20000000h
imul ebx

;Multiply a 32-bit integer by an 8-bit immediate value and store the product in AX
mov ax, 1000h
imul ax, 2

;Multiply a 32-bit integer by a 32-bit immediate value and store the product in EDX:EAX
mov eax, 10000000h
imul eax, 20000000h

;Multiply two 16-bit integers and store the product in memory
mov ax, 1000h
mov bx, 2000h
imul bx
mov [word_variable], ax

;Multiply two 32-bit integers and store the product in memory
mov eax, 10000000h
mov ebx, 20000000h
imul ebx
mov [dword_variable], eax
```

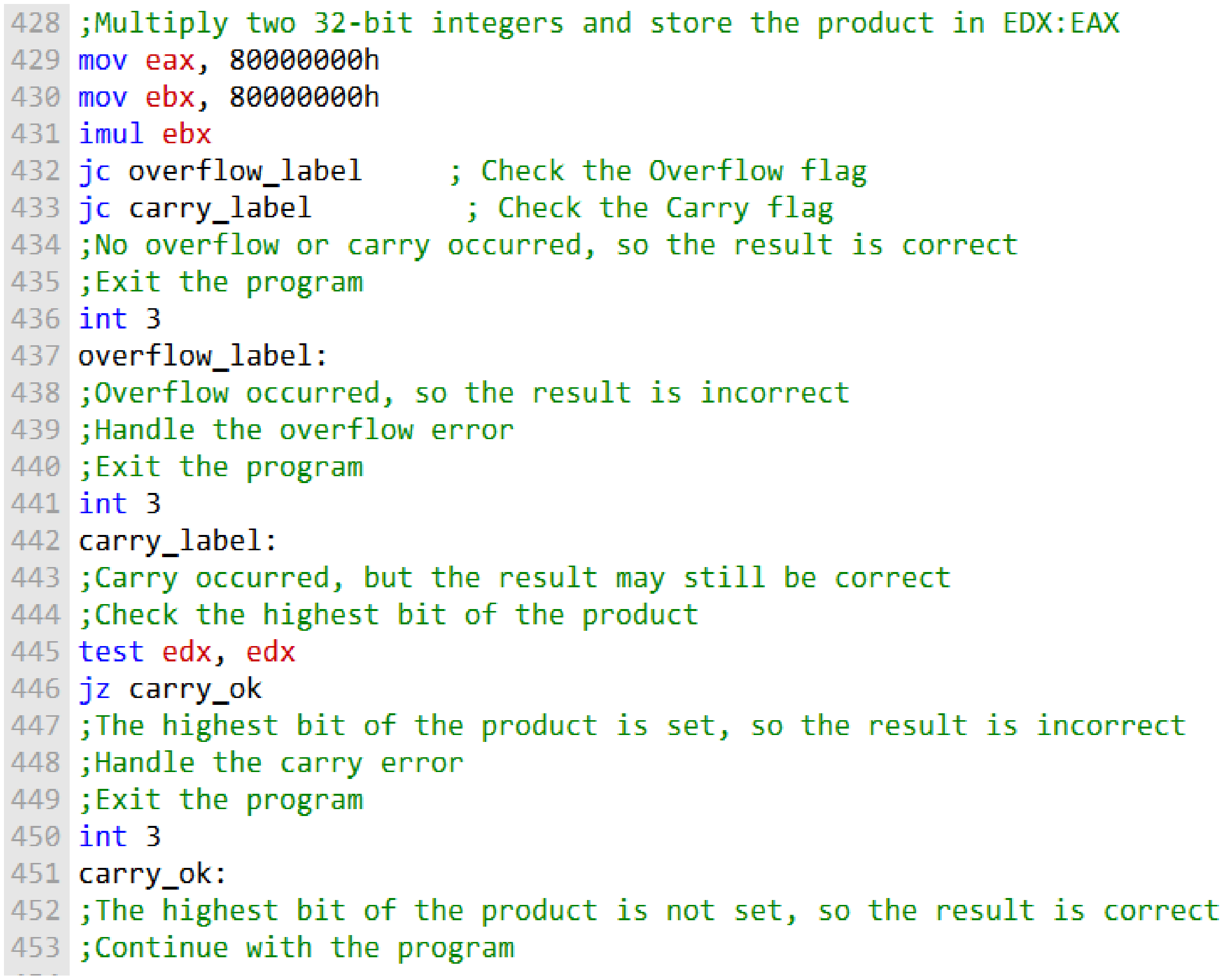
The first two lines of code move the values 1000h and 2000h into the AX and BX registers, respectively. The third line then uses the IMUL instruction to multiply the two values and store the product in the AX register.

The next two lines of code do the same thing, but with 32-bit integers. The IMUL instruction in this case stores the product in the EDX:EAX register pair.

The next two lines of code multiply a 32-bit integer by an 8-bit and 32-bit immediate value, respectively. The IMUL instruction in these cases stores the product in the AX and EDX:EAX register pair, respectively.

The last two lines of code multiply two 16-bit and 32-bit integers and store the product in memory. The IMUL instruction in these cases stores the product in the word and dword variable, respectively.

It is important to note that the IMUL instruction can generate overflow. This means that if the product of the two operands is too large to fit in the destination operand, the Overflow flag is set. It is important to check the Overflow flag after performing an IMUL operation to ensure that the result is correct.



```
;Multiply two 32-bit integers and store the product in EDX:EAX
mov eax, 80000000h
mov ebx, 80000000h
imul ebx
jc overflow_label     ; Check the Overflow flag
jc carry_label         ; Check the Carry flag
;No overflow or carry occurred, so the result is correct
;Exit the program
int 3
overflow_label:
;Overflow occurred, so the result is incorrect
;Handle the overflow error
;Exit the program
int 3
carry_label:
;Carry occurred, but the result may still be correct
;Check the highest bit of the product
test edx, edx
jz carry_ok
;The highest bit of the product is set, so the result is incorrect
;Handle the carry error
;Exit the program
int 3
carry_ok:
;The highest bit of the product is not set, so the result is correct
;Continue with the program
```

The first two lines of code move the values 1000h and 2000h into the AX and BX registers, respectively. The third line then uses the IMUL instruction to multiply the two values and store the product in the AX register.

The next two lines of code do the same thing, but with 32-bit integers. The IMUL instruction in this case stores the product in the EDX:EAX register pair.

The next two lines of code multiply a 32-bit integer by an 8-bit and 32-bit immediate value, respectively. The IMUL instruction in these cases stores the product in the AX and EDX:EAX register pair, respectively.

The last two lines of code multiply two 16-bit and 32-bit integers and store the product in memory. The IMUL instruction in these cases stores the product in the word and dword variable, respectively.

It is important to note that the IMUL instruction can generate overflow. This means that if the product of the two operands is too large to fit in the destination operand, the Overflow flag is set. It is important to check the Overflow flag after performing an IMUL operation to ensure that the result is correct.

```
;Multiply two 32-bit integers and store the product in EDX:EAX
mov     eax, 80000000h      ;Load the first integer into EAX
mov     ebx, 80000000h      ;Load the second integer into EBX
imul    ebx                 ;Multiply EAX by EBX; result in EDX:EAX
;Check for overflow:
jc      overflow_label      ;Jump if the Overflow flag is set
;Check for carry:
jc      carry_label         ;Jump if the Carry flag is set
;No overflow or carry occurred, so the result is correct
;Exit the program
int     3
overflow_label:
;Overflow occurred, so the result is incorrect
;Handle the overflow error
;Exit the program
int     3
carry_label:
;Carry occurred, but the result may still be correct
;Check the highest bit of the product
test    edx, edx            ;Test the value in EDX
jz      carry_ok            ;Jump if the highest bit of the product is not set
;The highest bit of the product is set, so the result is incorrect
;Handle the carry error
;Exit the program
int     3
carry_ok:
;The highest bit of the product is not set, so the result is correct
;Continue with the program
```

In this example, we multiply two 32-bit integers, 80000000h and 80000000h. The product of these two integers is 6400000000h, which is too large to fit in a 32-bit register. Therefore, the Overflow flag is set.

We then check the Carry flag. The Carry flag is set if the highest bit of the product is set. In this case, the highest bit of the product is not set, so the Carry flag is not set.

We then check the Overflow flag to see if overflow occurred. If overflow occurred, the result of the multiplication is incorrect. In this case, overflow occurred, so the result is incorrect.

We could handle the overflow error in a number of ways. For example, we could print an error message and exit the program. Or, we could try to scale the result down so that it fits in a 32-bit register.

If overflow did not occur, we need to check the Carry flag. If the Carry flag is set, the highest bit of the product is set. This may or may not indicate that the result is incorrect.

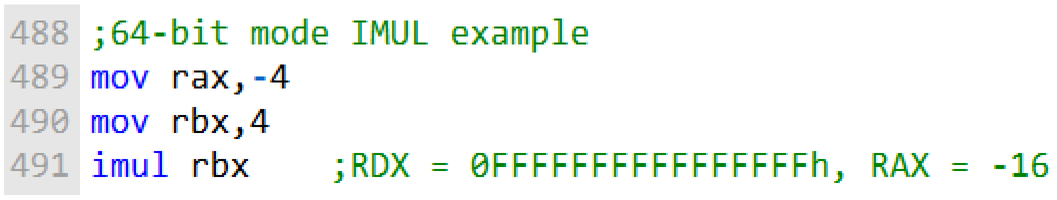
In this example, the highest bit of the product is not set, so the result is correct. We can then continue with the program.

It is important to note that this is just a simple example of how to illustrate the IMUL instruction overflow and carry flag in MASM. There are many other ways to handle overflow and carry errors.

=====================

IMUL in 64-Bit Mode

=====================

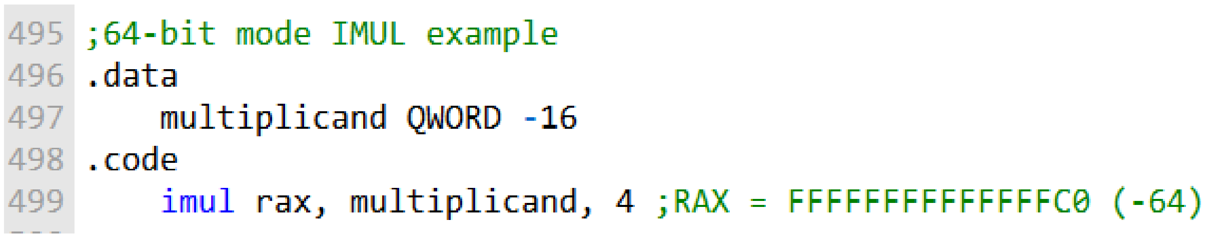


```
;64-bit mode IMUL example
mov rax,-4
mov rbx,4
imul rbx     ;RDX = 0FFFFFFFFFFFFFFFFh, RAX = -16
```

In this example, we multiply the 64-bit register RBX by the 64-bit register RAX. The product of these two integers is -64, which is a 128-bit value. The IMUL instruction in this case stores the product in the RDX:RAX register pair.

The RDX register stores the high-order 64 bits of the product, and the RAX register stores the low-order 64 bits of the product.

In this case, the high-order 64 bits of the product are all ones, so the RDX register is filled with the value 0xFFFFFFFF...FFFFh. The low-order 64 bits of the product are equal to -64, so the RAX register is filled with the value FFFFFFFFFFFFFFC0h.



```
;64-bit mode IMUL example
.data
    multiplicand QWORD -16
.code
    imul rax, multiplicand, 4 ;RAX = FFFFFFFFFFFFFFC0 (-64)
```

In this example, we multiply the 64-bit memory operand multiplicand by the immediate value 4. The product of these two operands is -64, which is a 64-bit value. The IMUL instruction in this case stores the product in the RAX register.

The multiplicand memory operand is defined in the data section of the program. It is a QWORD variable, which means that it is 64 bits wide. The immediate value 4 is a 32-bit value, but it is automatically promoted to 64 bits before the multiplication operation is performed.

The IMUL instruction in this case stores the product in the RAX register, which is a 64-bit register. Therefore, the RAX register will be filled with the value FFFFFFFFFFFFFFC0h after the IMUL instruction is executed.

Unsigned multiplication

The IMUL instruction can also be used to perform unsigned multiplication. However, there is a small disadvantage to doing so: the Carry and Overflow flags will not indicate whether the upper half of the product is equal to zero.

This is because the IMUL instruction always sign-extends the product, even if the operands are unsigned. This means that the Carry and Overflow flags will be set if the high-order bit of the product is set, even if the product is a valid unsigned integer.

The IMUL instruction is a powerful instruction that can be used to perform signed and unsigned multiplication in 64-bit mode. However, it is important to be aware of the fact that the Carry and Overflow flags will not indicate whether the upper half of the product is equal to zero when performing unsigned multiplication.

MUL Examples in Depth

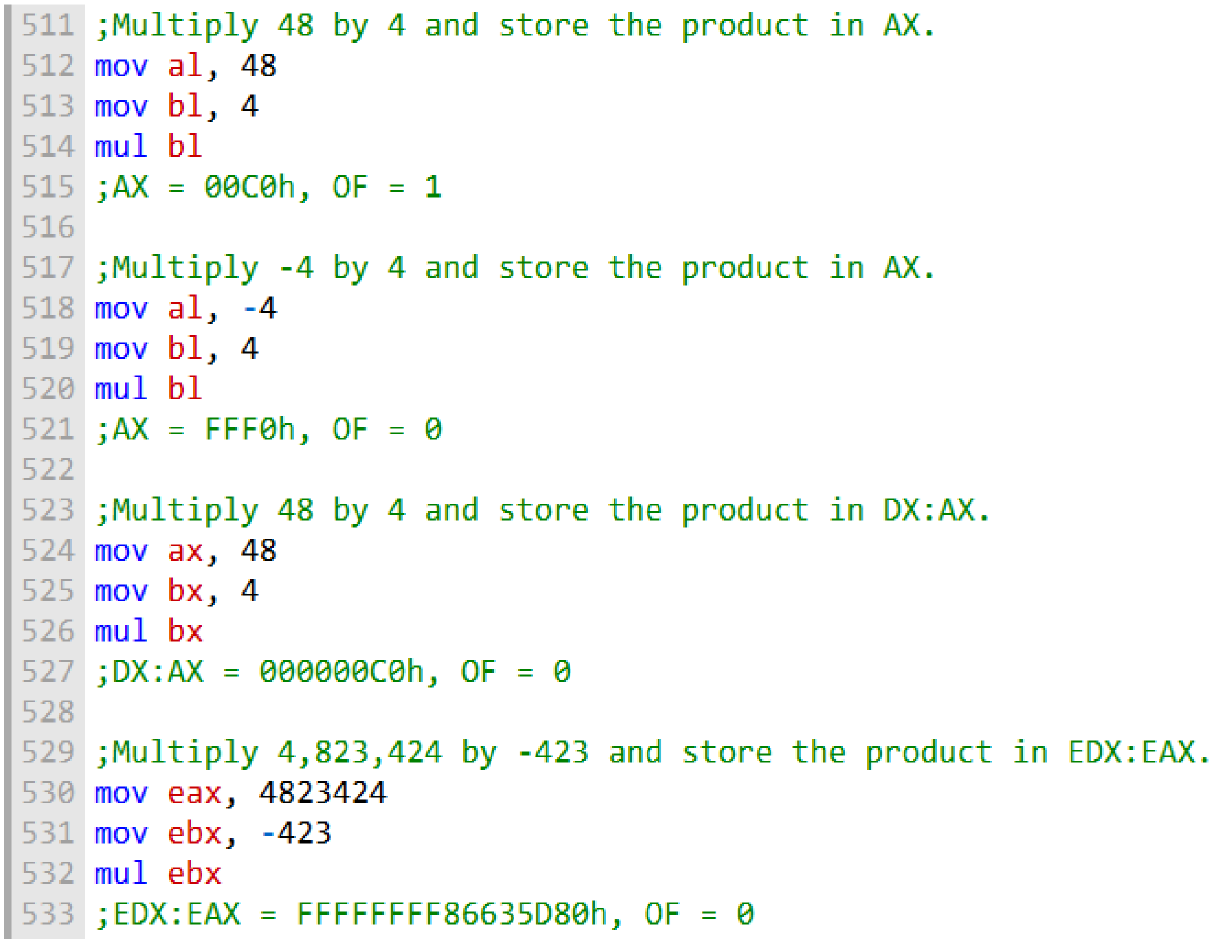
MUL Overflow

The MUL instruction can generate overflow if the product of the two operands is too large to fit in the destination operand. For example, the following code will generate overflow:

```
mov ax, -32000
imul ax, 2
```

The product of -32000 and 2 is -64000, which is too large to fit in a 16-bit register. Therefore, the Overflow flag will be set after the IMUL instruction is executed.

MUL Signed and Unsigned Examples



```
;Multiply 48 by 4 and store the product in AX.
mov al, 48
mov bl, 4
mul bl
;AX = 00C0h, OF = 1

;Multiply -4 by 4 and store the product in AX.
mov al, -4
mov bl, 4
mul bl
;AX = FFF0h, OF = 0

;Multiply 48 by 4 and store the product in DX:AX.
mov ax, 48
mov bx, 4
mul bx
;DX:AX = 000000C0h, OF = 0

;Multiply 4,823,424 by -423 and store the product in EDX:EAX.
mov eax, 4823424
mov ebx, -423
mul ebx
;EDX:EAX = FFFFFFFF86635D80h, OF = 0
```

Two-Operand IMUL Instructions

The two-operand IMUL instruction uses a destination operand that is the same size as the multiplier. Therefore, it is possible for signed overflow to occur. Always check the Overflow flag after executing these types of IMUL instructions. The following code examples demonstrate two-operand IMUL instructions:

```
;Multiply -16 by 2 and store the product in AX.
mov ax, -16
imul ax, 2
; AX = -32

;Multiply -32 by 2 and store the product in AX.
imul ax, 2
;AX = -64

;Multiply -64 by word1 and store the product in BX.
;word1 is a 16-bit variable.
mov bx, word1
imul bx, word1, -16
;BX = word1 * -16

;Multiply -64 by dword1 and store the product in BX.
;dword1 is a 32-bit variable.
mov bx, dword1
imul bx, dword1, -16
;BX = dword1 * -16

;Multiply dword1 by -2000000000 and store the product in BX.
;This instruction will generate overflow because the product is too large to fit in a 32-bit register.
imul bx, dword1, -2000000000
;signed overflow!
```

Three-Operand IMUL Instructions

The three-operand IMUL instruction uses a destination operand that is the same size as the first operand. The first operand is multiplied by the second operand and the product is stored in the third operand. The following code examples demonstrate three-operand IMUL instructions:

```
;Multiply word1 by -16 and store the product in BX.
mov bx, word1
imul bx, word1, -16
;BX = word1 * -16

;Multiply dword1 by -16 and store the product in BX.
mov bx, dword1
imul bx, dword1, -16
;BX = dword1 * -16

;Multiply dword1 by -2000000000 and store the product in BX.
;This instruction will generate overflow because the product is too large to fit in a 32-bit register.
mov bx, dword1
imul bx, dword1, -2000000000
;signed overflow!
```

The MUL and IMUL instructions are powerful instructions that can be used to perform signed and unsigned multiplication.

However, it is important to be aware of the fact that these instructions can generate overflow.

Always check the Overflow flag after executing these types of instructions to ensure that the result is correct.

MEASURING EXECUTION TIMES

The code example you provided shows how to use the GetMseconds procedure in the Irvine32 library to measure the execution time of a program.

The **GetMseconds procedure** returns the number of system milliseconds that have elapsed since midnight.

To measure the execution time of a program, you would first call the GetMseconds procedure to record the start time.

Then, you would call the program whose execution time you wish to measure. Finally, you would call the GetMseconds procedure again to record the end time.

The difference between the end time and the start time is the execution time of the program.

The following code example shows how to use the GetMseconds procedure to measure the execution time of a simple program:

```
.data
    startTime DWORD ?
    procTime DWORD ?

.code
    call GetMseconds
    ; get start time
    mov startTime, eax

    ; call the program whose execution time you wish to measure
    ; ...

    call GetMseconds
    ; get end time
    sub eax, startTime
    ; calculate the elapsed time
    mov procTime, eax
    ; save the elapsed time
```

The **variable procTime** will now contain the execution time of the program, in milliseconds.

You can use this technique to measure the execution time of any program, regardless of its complexity.

However, it is important to note that the overhead of calling the GetMseconds procedure twice is insignificant when compared to the execution time of most programs.

Relative Performance

You can also use the GetMseconds procedure to measure the relative performance of two different code implementations.

To do this, you would measure the execution time of each implementation and then divide the execution time of the first implementation by the execution time of the second implementation.

The result will be a number that indicates the relative performance of the two implementations.

For example, the following code example shows how to measure the relative performance of two different sorting algorithms:

```
.data
    startTime1 DWORD ?
    procTime1 DWORD ?
    startTime2 DWORD ?
    procTime2 DWORD ?
.code
    ;measure the execution time of the first sorting algorithm
    call GetMseconds    ;get start time
    mov startTime1, eax
    ; call the first sorting algorithm ...
    call GetMseconds
    ; get end time
    sub eax, startTime1
    ; calculate the elapsed time
    mov procTime1, eax
    ; save the elapsed time
    ; measure the execution time of the second sorting algorithm
    call GetMseconds
    ; get start time
    mov startTime2, eax
    ; call the second sorting algorithm
    ; ...
    call GetMseconds
    ; get end time
    sub eax, startTime2
    ; calculate the elapsed time
    mov procTime2, eax
    ; save the elapsed time
    ; calculate the relative performance of the two sorting algorithms
    div procTime1, procTime2
    ; the result is now in EAX
```

The EAX register will now contain the relative performance of the two sorting algorithms. A value of 1.0 indicates that the two sorting algorithms have the same performance.

A value greater than 1.0 indicates that the first sorting algorithm is faster than the second sorting algorithm. A value less than 1.0 indicates that the first sorting algorithm is slower than the second sorting algorithm.

You can use this technique to measure the relative performance of any two code implementations, regardless of their complexity.

Comparing MUL and IMUL to Bit Shifting in Depth

In older x86 processors, there was a significant difference in performance between multiplication by bit shifting and multiplication using the MUL and IMUL instructions.

However, in recent processors, Intel has managed to greatly optimize the MUL and IMUL instructions, so that they now have the same performance as bit shifting for multiplication by powers of two.

The following code shows two procedures for multiplying a number by 36 using bit shifting and the MUL instruction:

```
;Multiplies EAX by 36 using SHL, LOOP_COUNT times.
mult_by_shifting PROC
mov ecx, LOOP_COUNT
L1: push eax
; save original EAX
mov ebx, eax
shl eax, 5
shl ebx, 2
add eax, ebx
pop eax
; restore EAX
loop L1
ret
mult_by_shifting ENDP

; Multiplies EAX by 36 using MUL, LOOP_COUNT times.
mult_by_MUL PROC
mov ecx, LOOP_COUNT
L1:
push eax
; save original EAX
mov ebx, 36
mul ebx
pop eax
; restore EAX
loop L1
ret
mult_by_MUL ENDP
```

The following code calls the mult_by_shifting procedure and displays the timing results:

```
.data
    LOOP_COUNT = 0FFFFFFFFh
.data
    intval DWORD 5
.code
    call
    GetMseconds
    ; get start time
    mov
    startTime,eax
    mov
    eax,intval
    ; multiply now
    call
    mult_by_shifting
    call
    GetMseconds
    ; get stop time
    sub
    eax,startTime
    call WriteDec
    ; display elapsed time
```

The code above, is a simple example of how to measure the execution time of a program using the GetMseconds procedure in the Irvine32 library. The program multiplies the integer 5 by 36 using the mult_by_shifting procedure, and then displays the execution time.

The two .data segments in the program are used to define two variables: LOOP_COUNT and intval. LOOP_COUNT is a constant that specifies the number of times to repeat the multiplication operation. intval is the integer that is multiplied by 36.

The reason for having two .data segments is not entirely clear. It is possible that the original author of the code was simply trying to organize the data in a logical way.

However, it is also possible that the author was trying to take advantage of some optimization in the Irvine32 library.

Regardless of the reason, it is not necessary to have two .data segments in this program. The two variables could be defined in the same .data segment without any problems.

Here is a revised version of the program with the two .data segments combined into one:

```
.data
    LOOP_COUNT = 0FFFFFFFFh
    intval DWORD 5
.code
    call
    GetMseconds
    ; get start time
    mov
    startTime,eax
    mov
    eax,intval
    ; multiply now
    call
    mult_by_shifting
    call
    GetMseconds
    ; get stop time
    sub
    eax,startTime
    call WriteDec
    ; display elapsed time
```

This revised version of the program works just as well as the original version, and it is more concise and easier to read.

- You can have as many segments for .data, .code, .bss/text.
- Use segments wisely, grouping related data and code.
- Avoid excessive segments for clarity and performance.

On a legacy 4-GHz Pentium 4 processor, the mult_by_shifting procedure executed in 6.078 seconds, while the mult_by_MUL procedure executed in 20.718 seconds.

This means that using the MUL instruction was 241 percent slower. However, when running the same program on a more recent processor, the timings of both function calls were exactly the same.

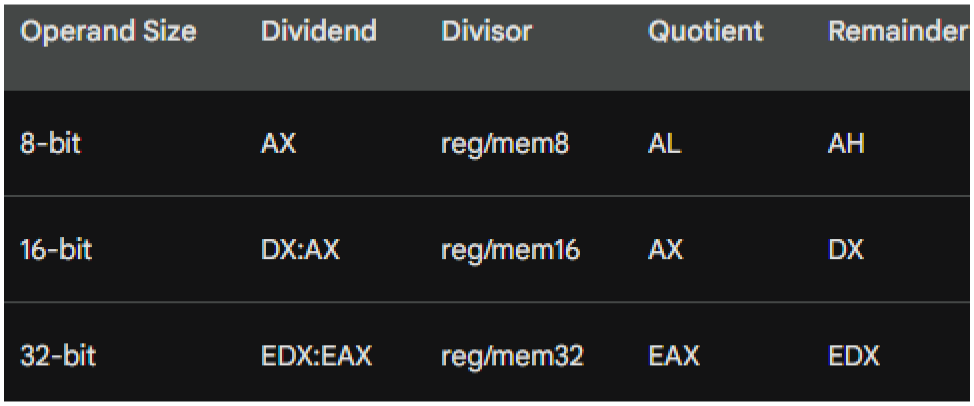
This example shows that Intel has managed to greatly optimize the MUL and IMUL instructions in recent processors.

Therefore, there is no longer any need to use bit shifting for multiplication by powers of two.

In fact, using the MUL and IMUL instructions is generally preferred, as they are more readable and easier to maintain.

DIV INSTRUCTION

The following table shows the relationship between the dividend, divisor, quotient, and remainder for the DIV instruction:



Operand Size	Dividend	Divisor	Quotient	Remainder
8-bit	AX	reg/mem8	AL	AH
16-bit	DX:AX	reg/mem16	AX	DX
32-bit	EDX:EAX	reg/mem32	EAX	EDX

In 64-bit mode, the DIV instruction uses RDX:RAX as the dividend and permits the divisor to be a 64-bit register or memory operand. The quotient is stored in RAX, and the remainder is stored in RDX.

The table above shows the relationship between the dividend, divisor, quotient, and remainder for the DIV instruction.

- **Dividend** is the number being divided.
- **Divisor** is the number by which the dividend is being divided.
- **Quotient** is the result of dividing the dividend by the divisor.
- **Remainder** is the number that is left over after the dividend is divided by the divisor.

The table shows that the operand size of the dividend and divisor determines the operand size of the quotient and remainder.

For example, if the dividend and divisor are 8-bit integers, then the quotient and remainder will also be 8-bit integers.

The table also shows that the dividend and divisor can be stored in registers or memory.

For example, the dividend can be stored in the AX register, and the divisor can be stored in the BL register.

Here is an example of how to use the DIV instruction to perform 8-bit unsigned division:

DIV Examples

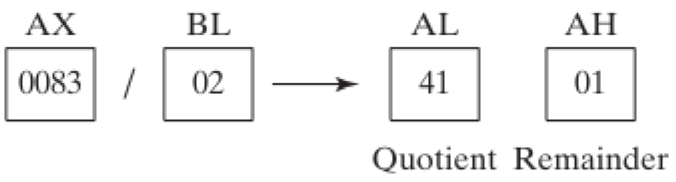
The following instructions perform 8-bit unsigned division (83h/2), producing a quotient of 41h and a remainder of 1:

```
;dividend
mov ax, 0083h
;divisor
mov bl, 2
;divide
div bl
;AL = 41h, AH = 01h
```

In this example, the dividend is stored in the AX register, and the divisor is stored in the BL register.

The DIV instruction divides the dividend by the divisor and stores the quotient in the AL register and the remainder in the AH register.

The following diagram illustrates the movement between registers:



DIV Example 2:

The following instructions perform 32-bit unsigned division using a memory operand as the divisor:

```
.data
    dividend QWORD 0000000800300020h
    divisor DWORD 00000100h
.code
    mov edx, DWORD PTR dividend + 4      ;high doubleword
    mov eax, DWORD PTR dividend          ;low doubleword
    div divisor
    ;EAX = 08003000h, EDX = 00000020h
```

Explanation:

The **.data directive** defines two variables: dividend and divisor. The dividend variable is a 64-bit integer (QWORD) that contains the dividend.

The **divisor variable is a 32-bit integer (DWORD)** that contains the divisor. The .code directive marks the beginning of the code section.

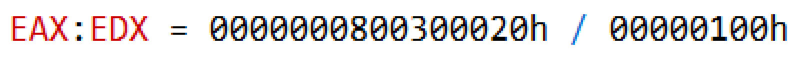
The **mov edx, DWORD PTR dividend + 4** instruction loads the high doubleword of the dividend into the EDX register.

The **mov eax, DWORD PTR** dividend instruction loads the low doubleword of the dividend into the EAX register.

The **div divisor instruction** divides the dividend in the EAX:EDX registers by the divisor in the divisor variable and stores the quotient in the EAX register and the remainder in the EDX register.

After the DIV instruction executes, the EAX register will contain the quotient (08003000h) and the EDX register will contain the remainder (00000020h).

In other words, the above code performs the following operation:



```
EAX:EDX = 0000000800300020h / 00000100h
```

The result is stored in the EAX:EDX registers, with the quotient in EAX and the remainder in EDX.

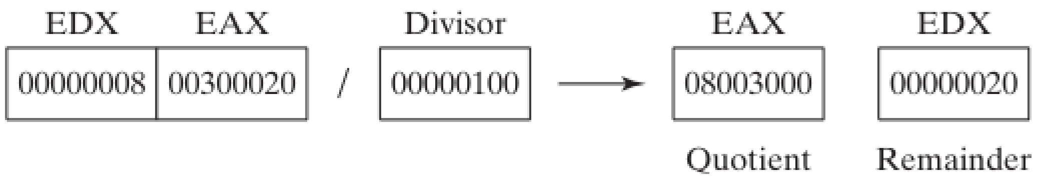
The **EAX:EDX registers** are a pair of 32-bit registers that can be used to store a 64-bit value. The EAX register stores the lower 32 bits of the value, and the EDX register stores the higher 32 bits of the value.

When you say that the result of a division operation is stored in the EAX:EDX registers, it means that the quotient of the division is stored in the EAX register and the remainder of the division is stored in the EDX register.

For example, if you divide the number 100 by the number 10, the quotient is 10 and the remainder is 0. The EAX register would contain the value 10 and the EDX register would contain the value 0.

Another way to think about it is that the EAX:EDX registers can be used to store a 64-bit integer. When you perform a division operation, the result is a 64-bit integer, which is then stored in the EAX:EDX registers.

The following diagram illustrates the movement between registers:



This image is related to the text you provided. The image shows a diagram of a sequence of numbers, with the following arrows and labels:

This diagram illustrates the **32-bit unsigned division operation** that is described in the text.

The dividend is 00300020h, the divisor is 00000100h, the quotient is 08003000h, and the remainder is 00000020h.

The EAX register contains the low doubleword of the dividend and the EDX register contains the high doubleword of the dividend.

The DIV instruction divides the dividend in the **EAX:EDX registers** by the divisor in the divisor variable. The quotient is stored in the EAX register and the remainder is stored in the EDX register.

EAX:EDX = 0000000800300020h / 00000100h

This equation represents the division operation that is being performed. The dividend is 0000000800300020h and the divisor is 00000100h. The result of the division is stored in the EAX:EDX registers.

DIV Example 3:

The following 64-bit division produces the quotient (0108000000003330h) in RAX and the remainder (0000000000000020h) in RDX:

```
.data
    dividend_hi QWORD 0000000000000108h
    dividend_lo QWORD 0000000033300020h
    divisor QWORD 0000000000010000h
.code
    mov rdx, dividend_hi
    mov rax, dividend_lo
    div divisor ;RAX = 0108000000003330
    ;RDX = 0000000000000020
```

Explanation:

The .data directive defines three variables: dividend_hi, dividend_lo, and divisor.

The dividend_hi and dividend_lo variables contain the high and low doublewords of the dividend, respectively.

The divisor variable contains the divisor. The .code directive marks the beginning of the code section.

The mov rdx, dividend_hi instruction loads the high doubleword of the dividend into the RDX register.

The mov rax, dividend_lo instruction loads the low doubleword of the dividend into the RAX register.

The div divisor instruction divides the dividend in the RAX:RDX registers by the divisor in the divisor variable and stores the quotient in the RAX register and the remainder in the RDX register.

After the DIV instruction executes, the RAX register will contain the quotient (0108000000003330h) and the RDX register will contain the remainder (0000000000000020h).

Why is each hexadecimal digit in the dividend shifted 4 positions to the right?

This is because the dividend is being divided by 64. In other words, the dividend is being shifted 6 bits to the right.

Each hexadecimal digit represents 4 bits, so each hexadecimal digit in the dividend will be shifted 4 positions to the right.

For example, the high doubleword of the dividend (0000000000000108h) is shifted 4 positions to the right to produce the following result:

0000000000000108h >> 4 = 0000000000000010h

The low doubleword of the dividend (0000000033300020h) is also shifted 4 positions to the right to produce the following result:

0000000033300020h >> 4 = 0000000000003330h

The quotient of the division operation is then stored in the RAX register and the remainder is stored in the RDX register.

SIGNED DIV INSTRUCTION

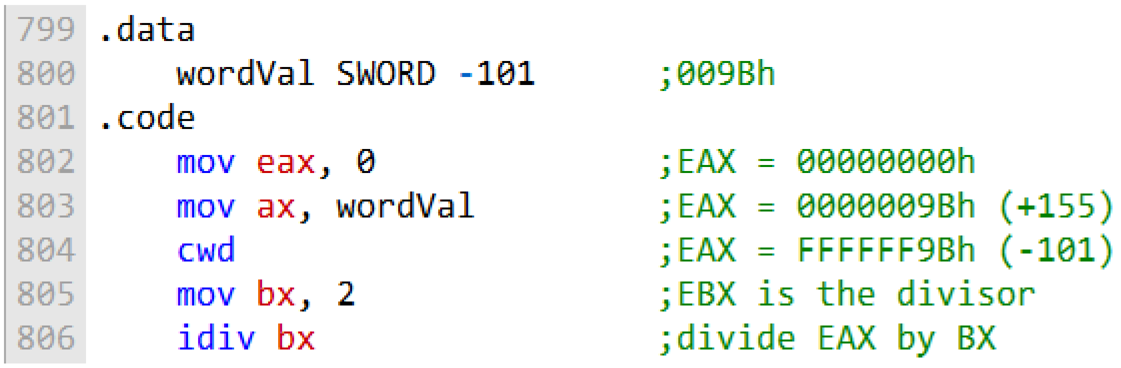
Signed integer division in MASM is similar to unsigned integer division, with one key difference: the dividend must be sign-extended before the division takes place. This is because the IDIV instruction (signed integer division) treats the dividend as a signed integer, and the result of the division is also a signed integer.

To sign-extend a number means to copy the sign bit of the number to all of the higher bits of the number.

This can be done using the:

- **CWD instruction (convert word to doubleword)**
- **CBW instruction (convert byte to word).**

The following MASM code shows how to sign-extend a 16-bit integer and then perform signed integer division:



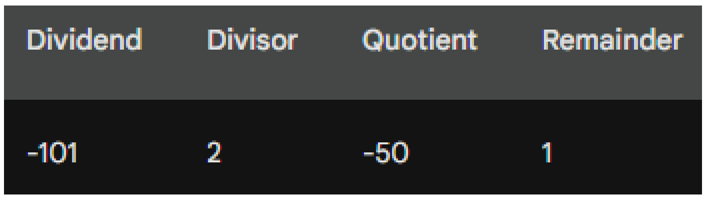
```
.data
    wordVal SWORD -101          ;009Bh
.code
    mov eax, 0                  ;EAX = 00000000h
    mov ax, wordVal             ;EAX = 0000009Bh (+155)
    cwd                         ;EAX = FFFFFF9Bh (-101)
    mov bx, 2                   ;EBX is the divisor
    idiv bx                     ;divide EAX by BX
```

In this code, the CWD instruction is used to sign-extend the AX register into the EAX register.

This ensures that the EAX register contains the correct signed value of -101 before the division operation is performed.

The IDIV instruction then divides the EAX register by the EBX register and stores the result in the EAX register.

The following table shows the results of the division operation:



Dividend	Divisor	Quotient	Remainder
-101	2	-50	1

The quotient of the division operation is -50 and the remainder is 1.

It is important to note that the **IDIV instruction** can also be used to perform unsigned integer division.

However, in this case, the dividend does not need to be sign-extended.

==

Sign Extension Instructions (CBW, CWD, CDQ)

==

The CBW, CWD, and CDQ instructions are sign extension instructions that are used to extend the sign bit of a smaller integer to a larger integer.

CBW

The CBW instruction (convert byte to word) extends the sign bit of the AL register into the AH register.

This means that if the AL register contains a negative byte value, the AH register will be set to FFh. Otherwise, the AH register will be set to 00h.

The following MASM code shows how to use the CBW instruction:

```
.data
    byteVal SBYTE -101
    ;9Bh
.code
    mov al, byteVal      ;AL = 9Bh
    cbw                  ;AX = FF9Bh
```

In this code, the CBW instruction is used to extend the sign bit of the AL register into the AH register. After the CBW instruction is executed, the AX register will contain the value FF9Bh, which is the signed representation of the number -101.

CWD

The CWD instruction (convert word to doubleword) extends the sign bit of the AX register into the DX register.

This means that if the AX register contains a negative word value, the DX register will be set to FFh. Otherwise, the DX register will be set to 00h.

The following MASM code shows how to use the CWD instruction:

```
.data
    wordVal SWORD -101
; FF9Bh
.code
    mov ax, wordVal ; AX = FF9Bh
    cwd ; DX:AX = FFFFFF9Bh
```

In this code, the CWD instruction is used to extend the sign bit of the AX register into the DX register. After the CWD instruction is executed, the DX:AX registers will contain the value FFFFFF9Bh, which is the signed representation of the number -101.

CDQ

The CDQ instruction (convert doubleword to quadword) extends the sign bit of the EAX register into the EDX register.

This means that if the EAX register contains a negative doubleword value, the EDX register will be set to FFh. Otherwise, the EDX register will be set to 00h.

The following MASM code shows how to use the CDQ instruction:

```
.data
    dwordVal SDWORD -101
    ;FFFFFF9Bh
.code
    mov eax, dwordVal   ;EAX = FFFFFF9Bh
    cdq                 ;EDX:EAX = FFFFFFFFFFFFFF9Bh
```

In this code, the CDQ instruction is used to extend the sign bit of the EAX register into the EDX register. After the CDQ instruction is executed, the EDX:EAX registers will contain the value FFFFFFFFFFFFFF9Bh, which is the signed representation of the number -101.

When to use sign extension instructions

Sign extension instructions are typically used in the following situations:

When performing signed integer arithmetic operations. When converting a signed integer from a smaller type to a larger type.

When passing a signed integer to a function that expects a signed integer parameter.

For example, if you are writing a function that calculates the average of two signed integers, you would need to use a sign extension instruction to ensure that the two integers are converted to the same type before the division operation is performed.

Conclusion

Sign extension instructions are a powerful tool that can be used to ensure that signed integers are handled correctly. By understanding how to use these instructions, you can write more efficient and reliable code.

IDIV INSTRUCTION

The **IDIV (signed divide) instruction** performs signed integer division, using the same operands as DIV.

However, before executing 8-bit division, the dividend (AX) must be completely sign-extended. The remainder always has the same sign as the dividend.

Syntax:

```
IDIV reg/mem8
IDIV reg/mem16
IDIV reg/mem32
```

Operands:

reg/mem8: An 8-bit register or memory location containing the divisor.

reg/mem16: A 16-bit register or memory location containing the divisor.

reg/mem32: A 32-bit register or memory location containing the divisor.

Operation:

The IDIV instruction divides the signed integer dividend in the AX register by the signed integer divisor in the operand.

The quotient is stored in the AL register and the remainder is stored in the AH register.

Example 1:

The following instructions divide -48 by 5. After IDIV executes, the quotient in AL is -9 and the remainder in AH is -3:

```
.data
    byteVal SBYTE -48
    ;D0 hexadecimal
.code
    mov al,byteVal
    ;lower half of dividend
    cbw
    ;extend AL into AH
    mov bl,+5
    ;divisor
    idiv bl
    ;AL = -9, AH = -3
```

Explanation:

The CBW instruction sign-extends the AL register into the AX register.

This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the AX register by the BL register and stores the quotient in the AL register and the remainder in the AH register.

Example 2:

The following instructions divide -5000 by 256:

```
.data
    wordVal SWORD -5000
.code
    mov ax,wordVal
    ;dividend, low
    cwd
    ;extend AX into DX
    mov bx,+256
    ;divisor
    idiv bx
    ;quotient AX = -19, rem DX = -136
```

Explanation:

The CWD instruction sign-extends the AX register into the DX register. This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the DX:AX registers by the BX register and stores the quotient in the AX register and the remainder in the DX register.

Example 3:

The following instructions divide 50,000 by -256:

```
.data
    dwordVal SDWORD + 50000
.code
    mov eax,dwordVal
    ;dividend, low
    cdq
    ;extend EAX into EDX
    mov ebx,-256
    ;divisor
    idiv ebx
    ;quotient EAX = -195, rem EDX = +80
```

Explanation:

The CDQ instruction sign-extends the EAX register into the EDX register.

This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the EDX:EAX registers by the EBX register and stores the quotient in the EAX register and the remainder in the EDX register.

Important:

The IDIV instruction undefines all arithmetic status flag values.

The IDIV instruction can also be used to perform unsigned integer division.

However, in this case, the dividend does not need to be sign-extended.

=================

Divide Overflow

=================

A divide overflow condition occurs when the result of a division operation is too large to fit into the destination operand. This causes a processor exception and halts the current program.

The following instructions generate a divide overflow because the quotient (100h) is too large for the 8-bit AL destination register:

```
mov ax,1000h
mov bl,10h
div bl
; AL cannot hold 100h
```

Avoiding Divide Overflow:

Use a larger destination operand. For example, instead of using the AL register, you could use the AX register or the EAX register.

Use a smaller divisor. For example, instead of dividing by 10h, you could divide by 2h. Use a combination of the above two approaches. For example, you could use the AX register as the destination operand and divide by 2h. Test the divisor before dividing to avoid division by zero.

The following code uses a 32-bit divisor and 64-bit dividend to reduce the probability of a divide overflow condition:

```
mov eax,1000h
cdq
mov ebx,10h
div ebx
; EAX = 00000100h
```

Explanation:

The CDQ instruction sign-extends the EAX register into the EDX register. This creates a 64-bit dividend in the EDX:EAX registers. The DIV instruction then divides the EDX:EAX registers by the EBX register and stores the quotient in the EAX register.

The following code uses a 32-bit divisor and 64-bit dividend to reduce the probability of a divide overflow condition and tests the divisor before dividing to avoid division by zero:

```
mov eax, dividend
mov bl, divisor
cmp bl, 0
je NoDivideZero

; Not zero: continue
div bl

; ...

NoDivideZero:
; Display "Attempt to divide by zero"
```

Explanation:

The **MOV instructions load the dividend and divisor** into the EAX and BL registers, respectively. The CMP instruction compares the BL register to zero. If the BL register is equal to zero, the JE instruction jumps to the NoDivideZero label.

If the **BL register is not equal to zero**, the DIV instruction divides the EAX register by the BL register and stores the quotient in the EAX register.

The **NoDivideZero label** is where the code will jump if the divisor is zero. At this point, the code could display an error message or take other appropriate action.

==

Implementing Arithmetic Expressions(ASM)

=======================================

To implement arithmetic expressions in assembly language, we need to break them down into their constituent operations. For example, the following C++ statement:

```
var4 = (var1 + var2) * var3;
```

can be broken down into the following assembly language instructions:

```
mov eax, var1
add eax, var2
mul var3
mov var4, eax
```

The first instruction loads the value of var1 into the EAX register. The second instruction adds the value of var2 to the EAX register.

The third instruction multiplies the value of var3 by the value in the EAX register and stores the result in the EAX register.

The fourth instruction stores the value in the EAX register into the var4 variable.

Handling Overflow

When performing arithmetic operations in assembly language, it is important to be aware of the possibility of overflow.

Overflow occurs when the result of an operation is too large to fit into the destination operand.

For example, the following assembly language instruction:

```
mul var3
```

will multiply the value in the EAX register by the value of var3 and store the result in the EAX register. If the product of the multiplication is too large to fit into the EAX register, overflow will occur.

To handle overflow, we can use the JC (jump on carry) instruction. The JC instruction will jump to a specified label if the carry flag is set.

The carry flag is set if there was an overflow when performing the previous arithmetic operation.

The following assembly language code shows how to handle overflow when multiplying two unsigned 32-bit integers:

```
mov eax, var1
add eax, var2
mul var3
jc tooBig       ;jump if overflow
mov var4, eax
jmp next
tooBig:
    ;display error message
```

If the MUL instruction generates a product larger than 32 bits, the JC instruction will jump to the tooBig label. The tooBig label can then display an error message or take other appropriate action.

Handling Signed Integers

When performing arithmetic operations on signed integers, it is important to be aware of the possibility of sign extension.

Sign extension is the process of copying the sign bit of an integer to all of the higher bits of the integer.

For example, the following assembly language instruction:

```
idiv var3
```

will divide the value in the EDX:EAX registers by the value of var3 and store the quotient in the EAX register and the remainder in the EDX register.

If the dividend is a signed integer, it is important to sign-extend the dividend into EDX before performing the division.

The following assembly language code shows how to divide two signed 32-bit integers:

```
mov eax, var2
neg eax
cdq                ;sign-extend dividend
idiv var3
mov ebx, edx       ;EBX = right side
```

The CDQ instruction sign-extends the EAX register into the EDX register. This ensures that the EDX:EAX registers contain the correct signed value of the dividend before the division operation is performed.

Questions:

Explain why overflow cannot occur when the MUL and one-operand IMUL instructions execute.

Overflow cannot occur because these instructions ensure that the destination operand is twice the size of the multiplicand and multiplier. This means there is always enough space to hold the result without overflowing.

How is the one-operand IMUL instruction different from MUL in the way it generates a multiplication product?

The one-operand IMUL instruction, unlike MUL, can perform signed integer multiplication. It generates a product that can be positive or negative, depending on the signs of the multiplicand and multiplier.

What has to happen for the one-operand IMUL to set the Carry and Overflow flags?

The Carry and Overflow flags are set when the product of one-operand IMUL is too large to fit into the destination operand size, signifying an overflow condition. This occurs when the result is outside the representable range for the given operand size.

When EBX is the operand in a DIV instruction, which register holds the quotient?

When EBX is the operand in a DIV instruction, the EAX register holds the quotient.

When BX is the operand in a DIV instruction, which register holds the quotient?

When BX is the operand in a DIV instruction, the AX register holds the quotient.

When BL is the operand in a MUL instruction, which registers hold the product?

When BL is the operand in a MUL instruction, the AX and DX registers hold the product. AX contains the low 16 bits, and DX contains the high 16 bits of the 32-bit product.

Show an example of sign extension before calling the IDIV instruction with a 16-bit operand.

Sign extension is necessary when working with signed integers. Here's an example of sign extension before calling IDIV with a 16-bit operand:

```
movsx  eax, word ptr [your_16_bit_variable] ; Sign extend 16-bit value to 32 bits
idiv   ebx ; Perform signed division with the extended value
```

This code first sign-extends the 16-bit value to a 32-bit value in the EAX register before performing a signed division with the IDIV instruction.

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.

```
; Load the first operand into the AL register
mov al, 0FFh

; Add the second operand to the AL register, setting the Carry flag if necessary
add al, 0FFh

; Save the result in the DL register, adding the Carry flag if necessary
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:

```
; Load the first operand into the EAX register
mov eax, 0xFFFFFFFFh

; Add the second operand to the EAX register, setting the Carry flag if necessary
add eax, 0xFFFFFFFFh

; Save the result in the EDX register, adding the Carry flag if necessary
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.

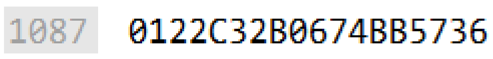
After the Extended_Add procedure has finished executing, the main procedure moves the pointer to the sum array into the ESI register and the length of the sum array into the ECX register.

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:



```
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.

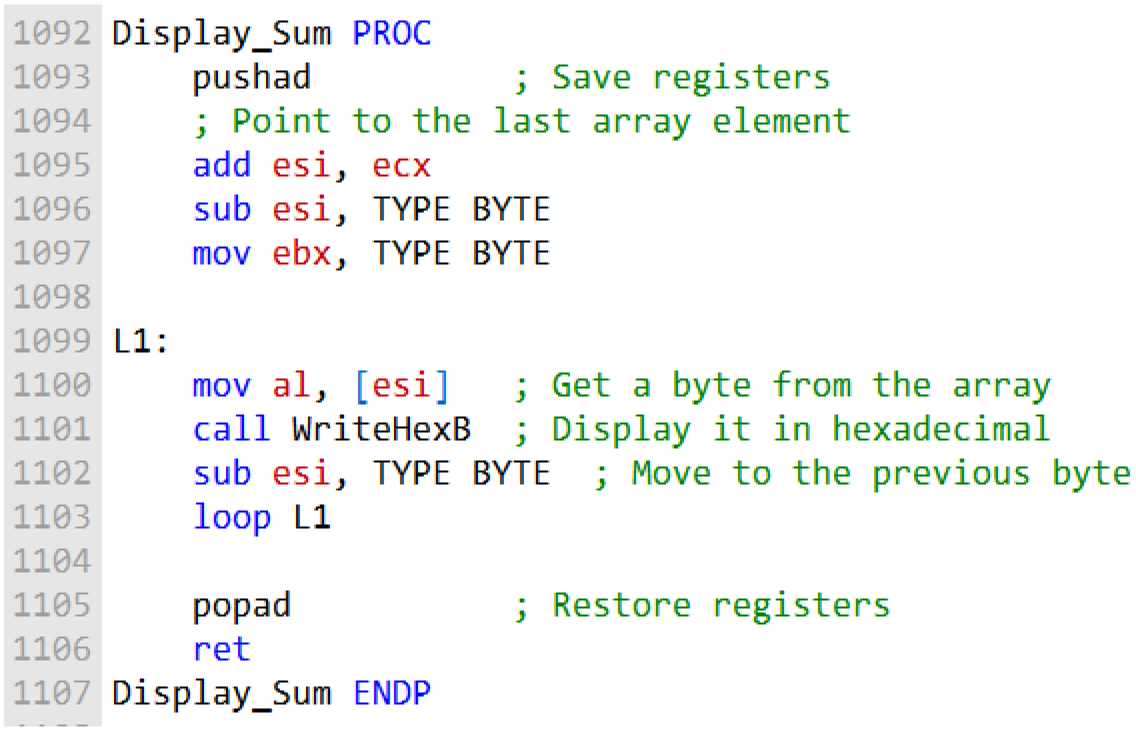
The Display_Sum procedure below is related to the Extended_Add procedure you provided in the previous question.

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:



```
Display_Sum PROC
    pushad          ; Save registers
    ; Point to the last array element
    add esi, ecx
    sub esi, TYPE BYTE
    mov ebx, TYPE BYTE

L1:
    mov al, [esi]   ; Get a byte from the array
    call WriteHexB  ; Display it in hexadecimal
    sub esi, TYPE BYTE  ; Move to the previous byte
    loop L1

    popad           ; Restore registers
    ret
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.

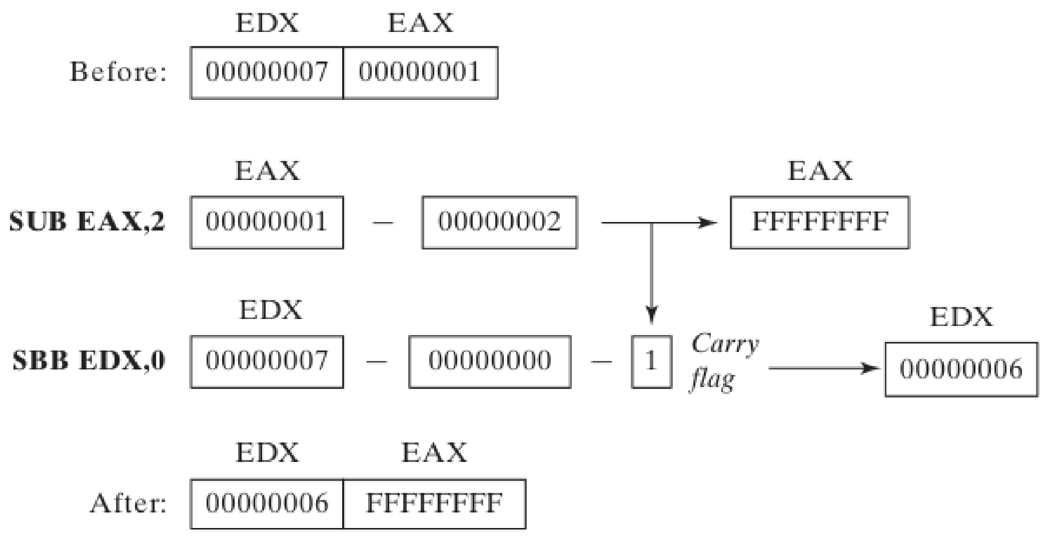
```
sbb edx, 0
; subtract upper half
```

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).

ASCII AND UNPACKED DECIMAL ARITHMETIC

This is a type of arithmetic that can be performed on ASCII decimal strings, without requiring them to be converted to binary.

There are two advantages to using ASCII arithmetic:

Conversion from string format before performing arithmetic is not necessary.

Using an assumed decimal point permits operations on real numbers without danger of the roundoff errors that occur with floating-point numbers.

However, ASCII arithmetic does execute more slowly than binary arithmetic.

There are four instructions that deal with ASCII addition, subtraction, multiplication, and division:

- **AAA (ASCII adjust after addition)**

- **AAS (ASCII adjust after subtraction)**
- **AAM (ASCII adjust after multiplication)**
- **AAD (ASCII adjust before division)**

These instructions are used to adjust the sum, difference, product, or quotient, respectively, to ensure that it is in a valid ASCII decimal format.

Here is an example of how to use ASCII addition to add the numbers 3402 and 1256:

```
; Load the first operand into the AL register.
mov al, 3
; Load the second operand into the AH register.
mov ah, 4
; Add the two operands together.
adc al, 0
; ASCII adjust the sum.
aaa
; Store the sum in the AX register.
mov ax, al
```

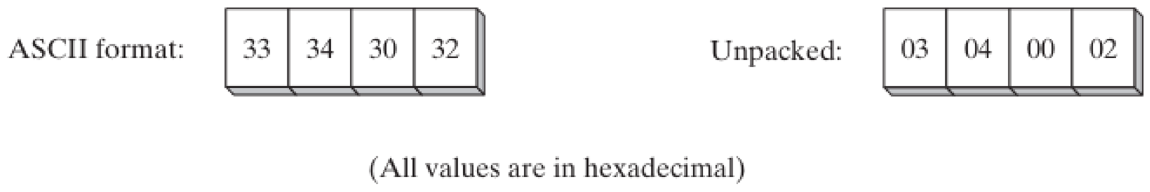
This code will add the two numbers together and store the sum in the AX register.

The AAA instruction is used to adjust the sum to ensure that it is in a valid ASCII decimal format.

ASCII subtraction can be performed in a similar way, using the AAS instruction to adjust the difference.

ASCII multiplication and division are also possible, but they are more complex and require the use of the AAM and AAD instructions, respectively.

ASCII arithmetic can be a useful tool for performing arithmetic on ASCII decimal strings. It is important to note, however, that ASCII arithmetic is slower than binary arithmetic.

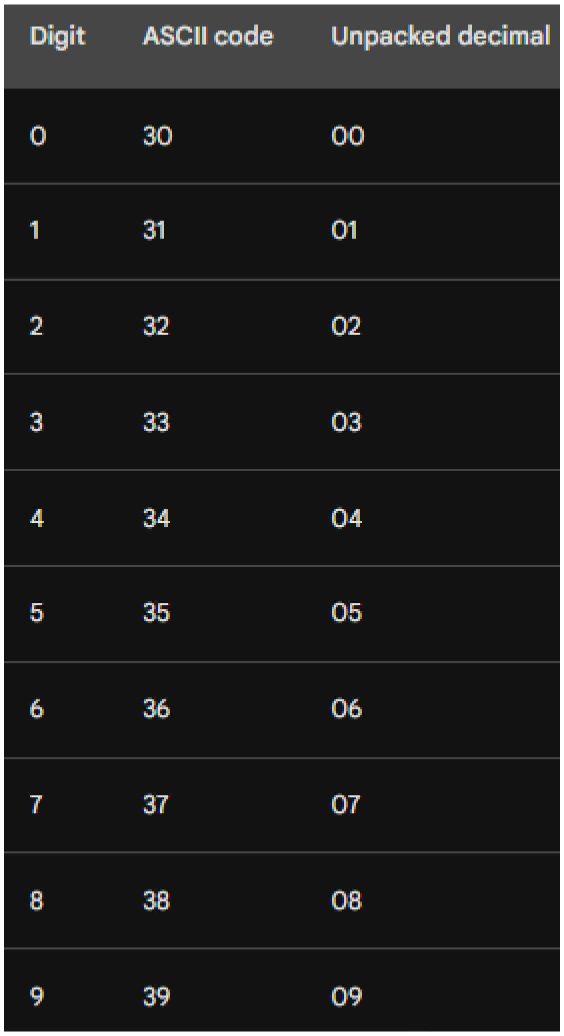


This means that the numbers in the block diagram are represented in two different formats: ASCII and unpacked decimal.

ASCII is a character encoding standard that assigns a unique code to each letter, number, and symbol. The ASCII codes for the digits 0 through 9 are 30 through 39, respectively.

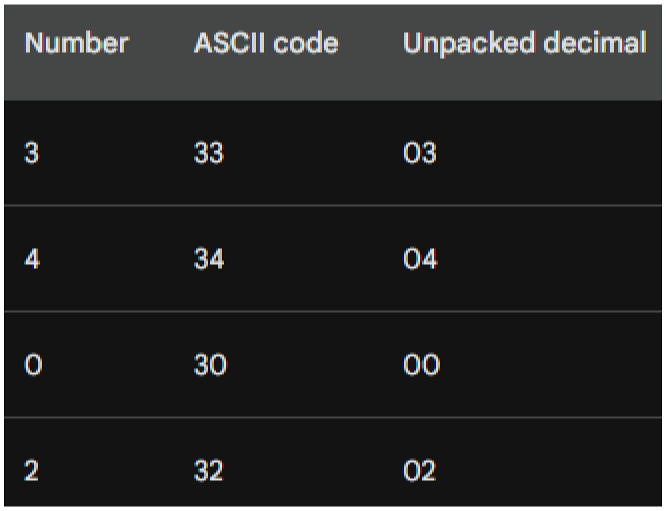
Unpacked decimal is a binary representation of decimal numbers, where one byte is used to represent each digit. The unpacked decimal representation of the number 12 is 000000010010, or 0x0302 in hexadecimal.

The image shows that the four numbers in the block diagram are the same in both ASCII and unpacked decimal formats. This is because the ASCII codes for the digits 0 through 9 are the same as the unpacked decimal representations of those digits.



Digit	ASCII code	Unpacked decimal
0	30	00
1	31	01
2	32	02
3	33	03
4	34	04
5	35	05
6	36	06
7	37	07
8	38	08
9	39	09

Here is a table showing the ASCII codes and unpacked decimal representations of the four numbers in the image:



Number	ASCII code	Unpacked decimal
3	33	03
4	34	04
0	30	00
2	32	02

This image is useful for understanding the relationship between ASCII and unpacked decimal formats. It is also a reminder that all numbers are ultimately represented in binary form, regardless of how they are displayed or stored.

As you can see, the ASCII codes for the digits 0 through 9 are simply the decimal values of the digits shifted by 4 bits. This makes it easy to convert between ASCII and unpacked decimal representations of numbers.

For example, to convert the ASCII code 34 to an unpacked decimal representation, we simply shift the ASCII code right by 4 bits. This gives us the unpacked decimal representation 04, which is the decimal value of 4.

To convert the unpacked decimal representation 05 to an ASCII code, we simply shift the unpacked decimal representation left by 4 bits. This gives us the ASCII code 35, which is the ASCII code for the digit 5.

The fact that the ASCII codes for the digits 0 through 9 are the same as their unpacked decimal representations makes it easy to perform arithmetic on ASCII decimal strings.

For example, we can add two ASCII decimal strings by simply adding the corresponding ASCII codes for each digit.

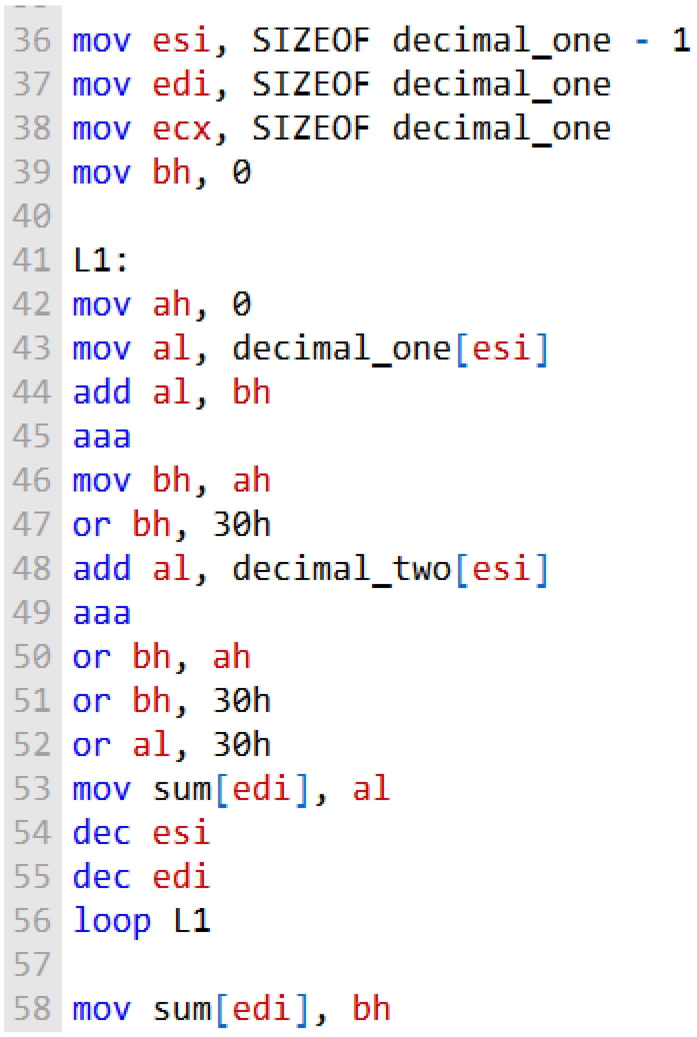
AAA (ASCII ADJUST AFTER ADDITION)

The ASCII addition procedure below is used to add ASCII decimal values with implied decimal points.

It works by iterating through the two operands, adding each corresponding digit and carrying over any carry from the previous iteration.

The procedure uses the AAA instruction to adjust the sum after each addition to ensure that it is in a valid ASCII decimal format.

Here is a more detailed explanation of the procedure:



```
mov esi, SIZEOF decimal_one - 1
mov edi, SIZEOF decimal_one
mov ecx, SIZEOF decimal_one
mov bh, 0

L1:
mov ah, 0
mov al, decimal_one[esi]
add al, bh
aaa
mov bh, ah
or bh, 30h
add al, decimal_two[esi]
aaa
or bh, ah
or bh, 30h
or al, 30h
mov sum[edi], al
dec esi
dec edi
loop L1

mov sum[edi], bh
```

The esi register points to the last digit position of the first operand, and the edi register points to the last digit position of the sum. The ecx register contains the length of the first operand.

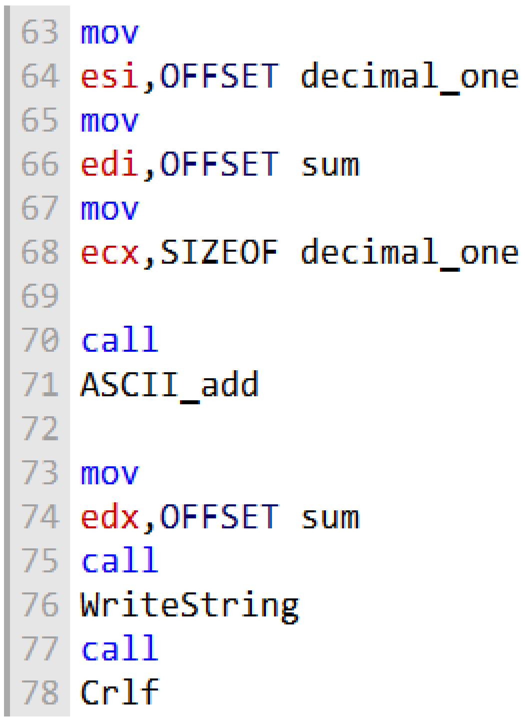
The loop at L1 iterates through the two operands, adding each corresponding digit and carrying over any carry from the previous iteration.

The AAA instruction is used to adjust the sum after each addition to ensure that it is in a valid ASCII decimal format.

The carry digit is saved in the bh register and then converted to ASCII. The ASCII carry digit is then added to the sum.

After the loop has finished iterating, the last carry digit is saved in the sum.

The following example shows how to use the ASCII addition procedure to add the numbers **1001234567.89765** and **9004020765.02015:**



```
mov
esi,OFFSET decimal_one
mov
edi,OFFSET sum
mov
ecx,SIZEOF decimal_one

call
ASCII_add

mov
edx,OFFSET sum
call
WriteString
call
Crlf
```

This code will produce the following output:

1000525533291780.

As you can see, the ASCII addition procedure correctly adds the two numbers, even though they have implied decimal points.

AAS, AAM and AAD

The AAS (ASCII adjust after subtraction) instruction is used to adjust the result of a subtraction operation when the result is negative.

It is typically used after a SUB or SBB instruction that has subtracted one unpacked decimal value from another and stored the result in the AL register.

The AAS instruction works by first checking the Carry flag. If the Carry flag is set, it means that the subtraction resulted in a negative number.

In this case, the AAS instruction subtracts 1 from the AH register and sets the Carry flag again.

It also sets the AL register to the ASCII representation of the negative number.

If the Carry flag is not set, it means that the subtraction resulted in a positive number.

In this case, the AAS instruction simply sets the AL register to the ASCII representation of the positive number.

Here is an example of how to use the AAS instruction:

```
mov ah, 0 ; clear AH before subtraction
mov al, 8
sub al, 9 ; subtract 9 from 8
aas ; adjust the result
or al, 30h ; convert AL to ASCII
```

After the above code has executed, the AL register will contain the ASCII representation of the number -1, which is 45h.

The AAS instruction can be useful for performing arithmetic operations on ASCII decimal strings.

For example, it can be used to subtract two ASCII decimal strings, even if they have different lengths.

Here is an example of how to use the AAS instruction to subtract two ASCII decimal strings:

```
mov esi, offset first_number
mov edi, offset second_number
mov ecx, length_of_first_number

loop:
mov ah, 0 ; clear AH before subtraction
mov al, [esi]
sub al, [edi]
aas ; adjust the result
or al, 30h ; convert AL to ASCII
mov [esi], al

inc esi ; increment the first number pointer
inc edi ; increment the second number pointer
dec ecx ; decrement the loop counter

cmp ecx, 0
jne loop ; continue looping if the loop counter is not zero
```

This code will subtract the two ASCII decimal strings starting at the least significant digits and working their way up to the most significant digits.

The AAS instruction is used to adjust the result of each subtraction operation to ensure that it is in a valid ASCII decimal format.

The AAS instruction is a powerful tool for performing arithmetic operations on ASCII decimal strings. It is easy to use and can be used to implement a variety of arithmetic algorithms.

===

AAM (ASCII adjust after multiplication)

===

he AAM (ASCII adjust after multiplication) instruction is used to convert the binary product produced by the MUL instruction to unpacked decimal format.

The MUL instruction must be used to multiply two unpacked decimal values.

The AAM instruction works by dividing the product by 100 and storing the quotient in the AH register and the remainder in the AL register.

The quotient represents the most significant digit of the unpacked decimal result, and the remainder represents the least significant digit.

Here is an example of how to use the AAM instruction:

```
mov bl, 5 ; first operand
mov al, 6 ; second operand
mul bl ; AX = 001Eh (binary product)
aam ; AX = 0300h (unpacked decimal result)
```

After the above code has executed, the AX register will contain the unpacked decimal representation of the product of 5 and 6, which is 30.

The AAM instruction can be useful for performing arithmetic operations on ASCII decimal strings. For example, it can be used to multiply two ASCII decimal strings, even if they have different lengths.

Here is an example of how to use the AAM instruction to multiply two ASCII decimal strings:

```
mov esi, offset first_number
mov edi, offset second_number
mov ecx, length_of_first_number

loop:
mov bl, [esi]
mov al, [edi]
mul bl ; AX = binary product of two digits
aam ; AX = unpacked decimal representation of product

mov [esi], al ; store the least significant digit of the product
mov [edi], ah ; store the most significant digit of the product

inc esi ; increment the first number pointer
inc edi ; increment the second number pointer
dec ecx ; decrement the loop counter

cmp ecx, 0
jne loop ; continue looping if the loop counter is not zero
```

This code will multiply the two ASCII decimal strings starting at the least significant digits and working their way up to the most significant digits.

The AAM instruction is used to convert the binary product of each multiplication operation to unpacked decimal format.

The AAM instruction is a powerful tool for performing arithmetic operations on ASCII decimal strings. It is easy to use and can be used to implement a variety of arithmetic algorithms.

=====================================

AAD (ASCII adjust before division)

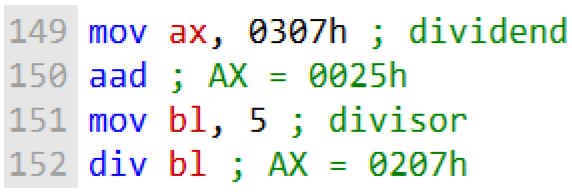
==

The AAD (ASCII adjust before division) instruction is used to convert an unpacked decimal dividend in AX to binary in preparation for executing the DIV instruction. This is necessary because the DIV instruction can only divide binary numbers.

The AAD instruction works by multiplying the AL register by 100 and adding the result to the AH register.

This ensures that the AH register contains the most significant digit of the unpacked decimal dividend and the AL register contains the least significant digit.

Here is an example of how to use the AAD instruction:



```
mov ax, 0307h ; dividend
aad ; AX = 0025h
mov bl, 5 ; divisor
div bl ; AX = 0207h
```

After the above code has executed, the AX register will contain the quotient and remainder of the division operation, respectively. The quotient is stored in the AL register, and the remainder is stored in the AH register.

The AAD instruction can be useful for performing arithmetic operations on ASCII decimal strings. For example, it can be used to divide two ASCII decimal strings, even if they have different lengths.

Here is an example of how to use the AAD instruction to divide two ASCII decimal strings:

```
mov esi, offset first_number
mov edi, offset second_number
mov ecx, length_of_first_number

loop:
mov bl, [esi]
aad ; AX = unpacked decimal representation of first number

mov ah, 0 ; clear AH before division
mov al, [edi]
div bl ; AX = quotient and remainder of division

mov [esi], al ; store the quotient
mov [edi], ah ; store the remainder

inc esi ; increment the first number pointer
inc edi ; increment the second number pointer
dec ecx ; decrement the loop counter

cmp ecx, 0
jne loop ; continue looping if the loop counter is not zero
```

This code will divide the two ASCII decimal strings starting at the most significant digits and working their way down to the least significant digits.

The AAD instruction is used to convert the unpacked decimal representation of the first number to binary before each division operation.

The AAD instruction is a powerful tool for performing arithmetic operations on ASCII decimal strings. It is easy to use and can be used to implement a variety of arithmetic algorithms.

Questions:

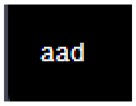
Question: Write a single instruction that converts a two-digit unpacked decimal integer in AX to ASCII decimal.

Answer: To convert a two-digit unpacked decimal integer in AX to ASCII decimal, you can use the AAM (ASCII Adjust AX After Multiplication) instruction:

```
aam
```

Question: Write a single instruction that converts a two-digit ASCII decimal integer in AX to unpacked decimal format.

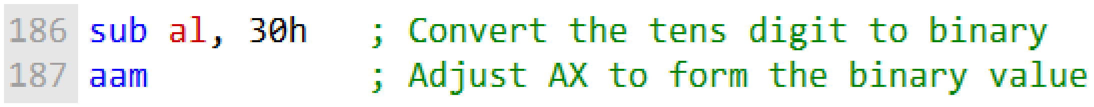
Answer: To convert a two-digit ASCII decimal integer in AX to unpacked decimal format, you can use the AAD (ASCII Adjust AX Before Division) instruction:



```
aad
```

Question: Write a two-instruction sequence that converts a two-digit ASCII decimal number in AX to binary.

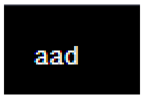
Answer: To convert a two-digit ASCII decimal number in AX to binary, you can use the following two-instruction sequence:



```
sub al, 30h    ; Convert the tens digit to binary
aam            ; Adjust AX to form the binary value
```

Question: Write a single instruction that converts an unsigned binary integer in AX to unpacked decimal.

Answer: To convert an unsigned binary integer in AX to unpacked decimal, you can use the AAD (ASCII Adjust AX Before Division) instruction:



```
aad
```

PACKED DECIMAL ARITHMETIC, DAA and DAS

Packed decimal arithmetic is a way of representing and performing arithmetic on decimal numbers using a binary format.

Packed decimal numbers store two decimal digits per byte, with each digit represented by 4 bits.

This makes packed decimal arithmetic a more efficient way to store and manipulate decimal numbers than traditional binary arithmetic, which stores each decimal digit as a separate byte.

Packed decimal arithmetic is often used in financial and business applications, where it is important to be able to perform accurate calculations on large numbers.

It is also used in some scientific and engineering applications, where it is necessary to perform calculations on numbers with a high degree of precision.

There are two main advantages to using packed decimal arithmetic:

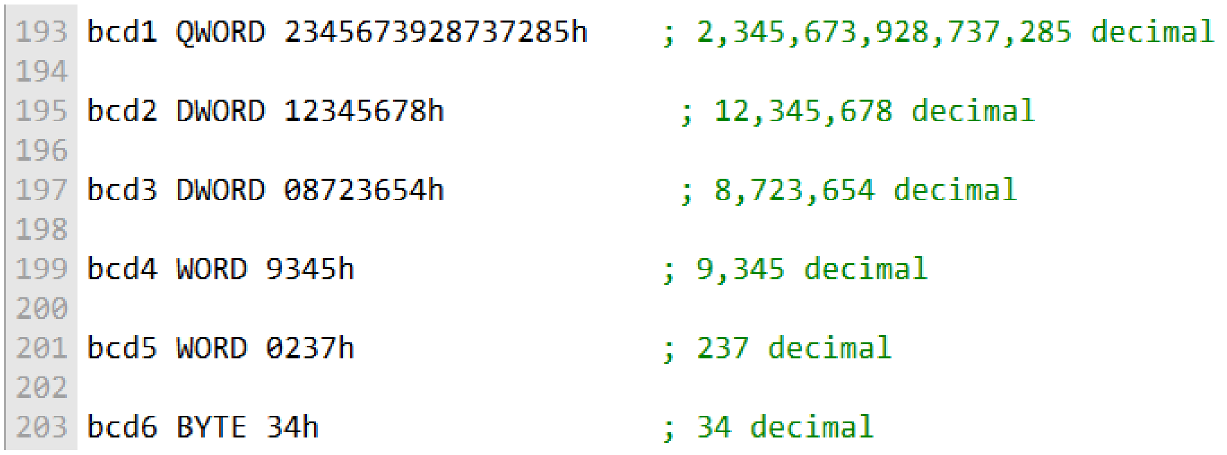
Efficiency: Packed decimal numbers require less storage space than traditional binary numbers. This is because packed decimal numbers store two decimal digits per byte, while traditional binary numbers store each decimal digit as a separate byte.

Accuracy: Packed decimal arithmetic can be used to perform calculations with a high degree of accuracy. This is because packed decimal numbers store two decimal digits per byte, which allows for more precise calculations than traditional binary arithmetic.

However, there is one main disadvantage to using packed decimal arithmetic:

Performance: Packed decimal arithmetic can be slower than traditional binary arithmetic. This is because packed decimal arithmetic requires additional instructions to convert packed decimal numbers to and from binary numbers, which is necessary for performing arithmetic operations. Overall, packed decimal arithmetic is a powerful tool for performing arithmetic on decimal numbers. It is especially useful for financial and business applications, where it is important to be able to perform accurate calculations on large numbers.

Here are some examples of packed decimal numbers:



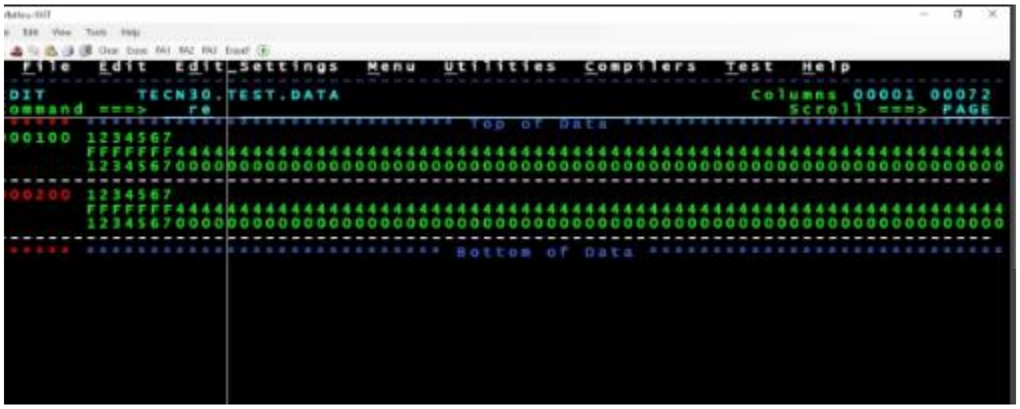
```
bcd1 QWORD 2345673928737285h    ; 2,345,673,928,737,285 decimal

bcd2 DWORD 12345678h             ; 12,345,678 decimal

bcd3 DWORD 08723654h             ; 8,723,654 decimal

bcd4 WORD 9345h                 ; 9,345 decimal

bcd5 WORD 0237h                 ; 237 decimal

bcd6 BYTE 34h                   ; 34 decimal
```

The following two instructions are used to adjust the result of an addition or subtraction operation on packed decimals:

- **DAA (decimal adjust after addition)**
- **DAS (decimal adjust after subtraction)**

These instructions are necessary because packed decimal numbers store two decimal digits per byte. After an addition or subtraction operation, it is possible that the result will be a three-digit number. The DAA and DAS instructions adjust the result to ensure that it is a valid two-digit packed decimal number.

Packed decimal arithmetic can be used to perform all of the basic arithmetic operations, including addition, subtraction, multiplication, and division. However, there are no specific instructions for multiplication and division of packed decimal numbers. This means that packed decimal numbers must be unpacked before these operations can be performed, and then repacked after the operations are complete.



```
 File   Edit   Edit_Settings   Menu   Utilities   Compilers   Test   Help
DIT          TECN30.TEST.DATA                              Columns 00001 00072
ommand ===>    re                                             Scroll ===> PAGE
****** ***************************** Top of Data ******************************
00100  1234567
       FFFFFFF44444444444444444444444444444444444444444444444444444444444444444
       1234567000000000000000000000000000000000000000000000000000000000000000000
--------------------------------------------------------------------------------
00200  1234567
       FFFFFFF44444444444444444444444444444444444444444444444444444444444444444
       1234567000000000000000000000000000000000000000000000000000000000000000000
--------------------------------------------------------------------------------
****** **************************** Bottom of Data ****************************
```

Despite this disadvantage, packed decimal arithmetic is a powerful tool for performing arithmetic on decimal numbers. It is especially useful for financial and business applications, where it is important to be able to perform accurate calculations on large numbers.

===

DAA (decimal adjust after addition)

===

The DAA (decimal adjust after addition) instruction is used to convert the binary sum produced by the ADD or ADC instruction in the AL register to packed decimal format.

This is necessary because packed decimal numbers store two decimal digits per byte. After an addition operation, it is possible that the result will be a three-digit number.

The DAA instruction adjusts the result to ensure that it is a valid two-digit packed decimal number.

Here is an example of how to use the DAA instruction:

```
mov al, 35h
add al, 48h ; AL = 7Dh
daa ; AL = 83h (adjusted result)
```

In this example, the ADD instruction adds the packed decimal numbers 35 and 48. The result is 7Dh, which is a three-digit binary number.

The DAA instruction adjusts the result to 83h, which is the packed decimal representation of the sum of 35 and 48.

The DAA instruction can be used to perform packed decimal addition on any number of digits.

However, it is important to note that the sum variable must contain space for one more digit than the operands. This is because the DAA instruction can generate a carry digit.

The following program adds two 16-bit packed decimal integers and stores the sum in a packed doubleword:

```
; Packed Decimal Example
(AddPacked.asm)
; Demonstrate packed decimal addition.
INCLUDE Irvine32.inc
.data
packed_1 WORD 4536h
packed_2 WORD 7207h
sum DWORD ?
.code
main PROC
   ; Initialize sum and index.
   mov sum, 0
   xor esi, esi

   ; Add low bytes and handle carry.
   add al, BYTE PTR packed_1[esi]
   daa
   mov BYTE PTR sum[esi], al

   ; Add high bytes and include carry.
   inc esi
   add al, BYTE PTR packed_1[esi]
   adc al, BYTE PTR packed_2[esi]
   daa
   mov BYTE PTR sum[esi], al

   ; Add final carry, if any.
   inc esi
   adc al, 0
   mov BYTE PTR sum[esi], al

   ; Display the sum in hexadecimal.
   mov eax, sum
   call WriteHex
   call Crlf
   exit
main ENDP
END main
```

This program uses a loop to add the two packed decimal integers one digit at a time. The DAA instruction is used to adjust the result of each addition operation.

The sum variable is a packed doubleword, which is large enough to store the sum of two 16-bit packed decimal integers.

The DAA instruction is a powerful tool for performing packed decimal arithmetic. It is easy to use and can be used to implement a variety of arithmetic algorithms.

==

DAS (decimal adjust after subtraction)

==

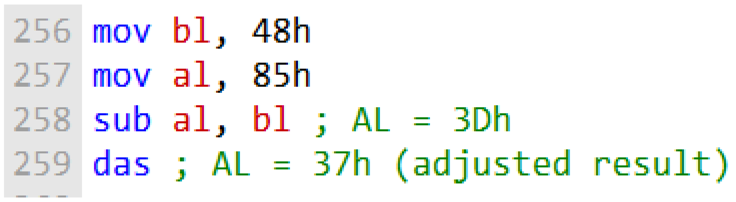
The DAS (decimal adjust after subtraction) instruction is used to convert the binary result of a SUB or SBB instruction in the AL register to packed decimal format.

This is necessary because packed decimal numbers store two decimal digits per byte.

After a subtraction operation, it is possible that the result will be a negative three-digit number.

The DAS instruction adjusts the result to ensure that it is a valid two-digit packed decimal number.

Here is an example of how to use the DAS instruction:



```
mov bl, 48h
mov al, 85h
sub al, bl ; AL = 3Dh
das ; AL = 37h (adjusted result)
```

In this example, the SUB instruction subtracts the packed decimal numbers 85 and 48. The result is 3Dh, which is a negative three-digit binary number.

The DAS instruction adjusts the result to 37h, which is the packed decimal representation of the difference of 85 and 48.

The DAS instruction can be used to perform packed decimal subtraction on any number of digits.

However, it is important to note that the result variable must contain space for one more digit than the operands.

This is because the DAS instruction can generate a borrow digit.

Here is a pseudocode implementation of the DAS instruction:

```
DAS(AL):
  if AL < 10:
    return AL
  else:
    AL -= 10
    AH += 1
    if AH >= 10:
      AH -= 10
      CF = 1
    else:
      CF = 0
    return AL
```

Explanation:

The DAS instruction begins by checking if the value in the AL register (the low decimal digit) is less than 10. If it is, it means there's no need for adjustment, and it returns AL as it is.

If AL is greater than or equal to 10, it means there's a carry or overflow in the low digit. To correct this:

Subtract 10 from AL, effectively "borrowing" from the low digit. Increment AH (the high digit) to account for the borrow from AL.

Check if AH itself requires adjustment. If AH is now greater than or equal to 10, it means there's a carry in the high digit as well.

If AH needs adjustment, subtract 10 from AH to bring it within the valid range.

Finally, set the Carry Flag (CF) to 1 to indicate that there was a carry or borrow operation.

If AH does not require adjustment, set CF to 0 to indicate that no carry occurred.

In summary, the DAS instruction ensures that after a subtraction operation, the AL and AH registers contain valid packed decimal digits, taking into account any borrows or carries to maintain the integrity of the packed decimal representation.

It is a crucial instruction in packed decimal arithmetic, commonly used in financial and decimal data processing applications.