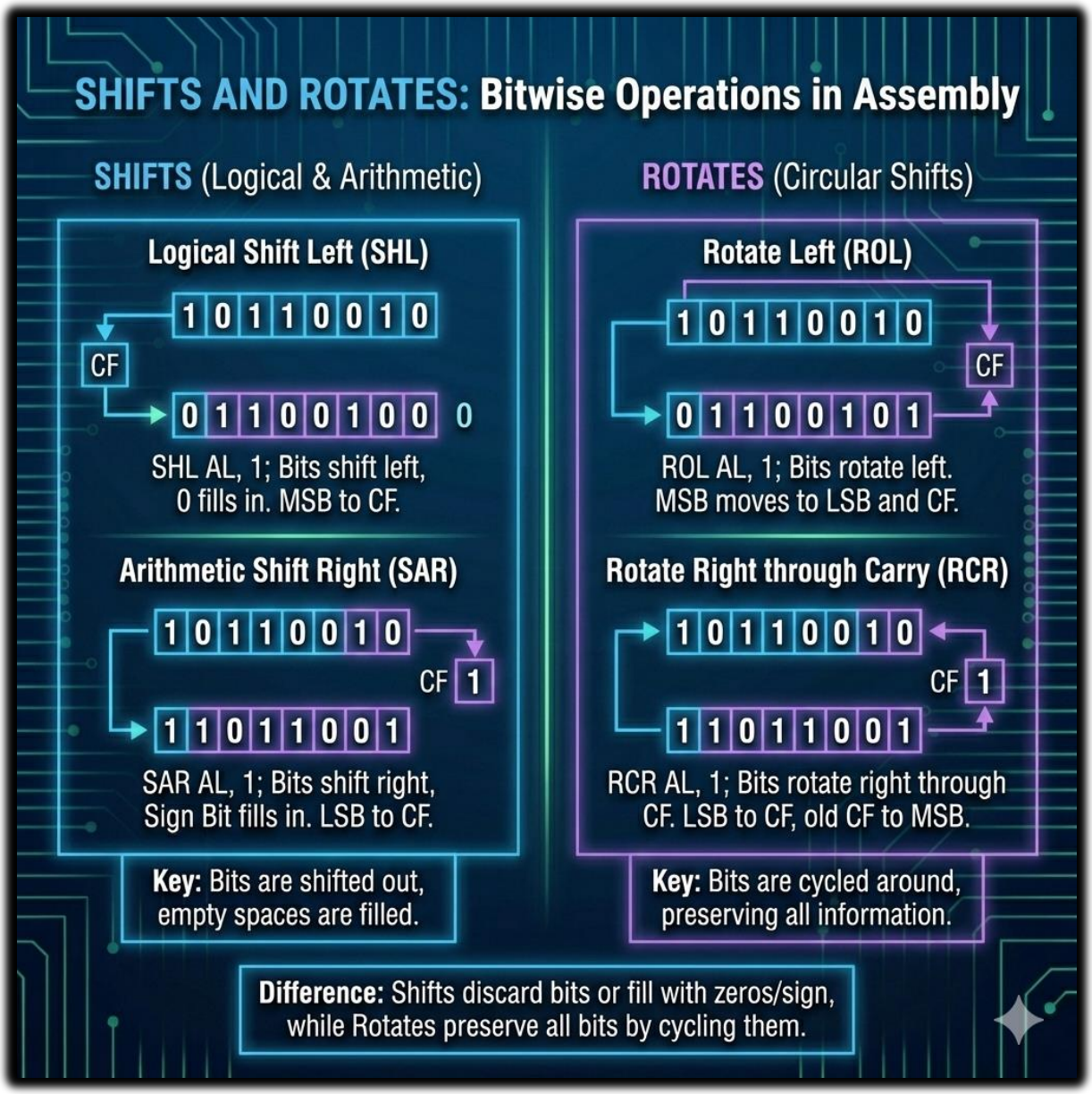
Contents

SHIFTS AND ROTATES 2
SHIFT LEFT AND SHIFT RIGHT 6
ARITHMETIC SHIFT RIGHT AND ARITHMETIC SHIFT LEFT 15
ROTATE LEFT AND ROTATE RIGHT 17
SHIFT LEFT DOUBLE AND SHIFT RIGHT DOUBLE 21
SHIFTING MULTIPLE DOUBLEWORDS 26
BINARY MULTIPLICATION WITH BIT SHIFTS (ASSEMBLY STYLE) 30
BINTOASC 34
EXTRACTING FILE DATE FIELDS 36
MUL OPERATOR 40
IMUL OPERATOR 48
MEASURING EXECUTION TIMES 56
DIV INSTRUCTION 62
SIGNED DIV INSTRUCTION 66
IDIV INSTRUCTION 70
EXTENDED ADDITION AND SUBTRACTION 77
ASCII AND UNPACKED DECIMAL ARITHMETIC 86
AAA (ASCII ADJUST AFTER ADDITION) 90
AAS, AAM and AAD 92
PACKED DECIMAL ARITHMETIC, DAA and DAS 97

SHIFTS AND ROTATES

In assembly, you have direct control over the bits and bytes of data, which lets you optimize code in highly efficient and platform-specific ways.



Let's break down these key concepts:

Bit Shifting:

This involves moving the bits of a number left or right.

- **Left Shift (<<)**: Moving bits left by a set number of positions multiplies the value by 2 for each shift.
- **Right Shift (>>)**: Shifting bits right divides the value by 2 for each shift (if the number is non-negative).
- **Logical Shift**: This just fills the empty bit spots with zeros.
- **Arithmetic Shift**: When shifting right, this keeps the sign bit (for signed numbers) intact.

Bit Rotation:

This is like shifting bits in a circular way, where they wrap around.

- **Left Rotation**: Bits shift left, and the ones that move past the most significant bit (MSB) wrap around to the least significant bit (LSB).
- **Right Rotation**: Bits shift right, and the ones that move past the LSB wrap around to the MSB.

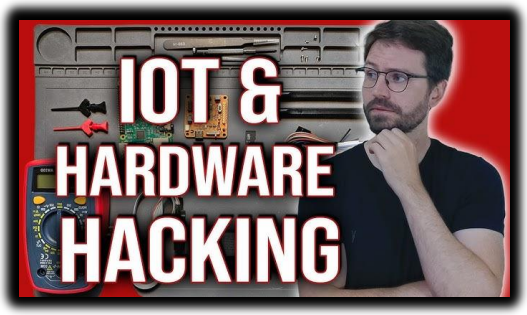
Uses:

Optimized Multiplication and Division: Bit shifting makes multiplication and division faster. Left shifting by **n** positions is the same as multiplying by 2^n, and right shifting is like dividing by 2^n.

Data Encryption: Bit manipulation is key in encryption algorithms like XOR ciphers.

Computer Graphics: Shifting bits is crucial for tasks like image processing and color manipulation.

Hardware Manipulation: In embedded systems, setting or clearing specific bits helps you control hardware and interact with peripherals.



These techniques are powerful and can make your code super efficient!

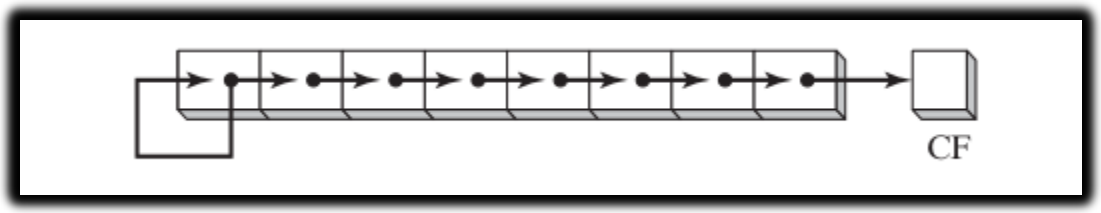
Assembly also lets you do arithmetic on integers of any size, which can be a huge advantage over high-level languages when you're working with large numbers or custom operations.

Now, let's focus on some important shift and rotate instructions, especially on x86 processors:

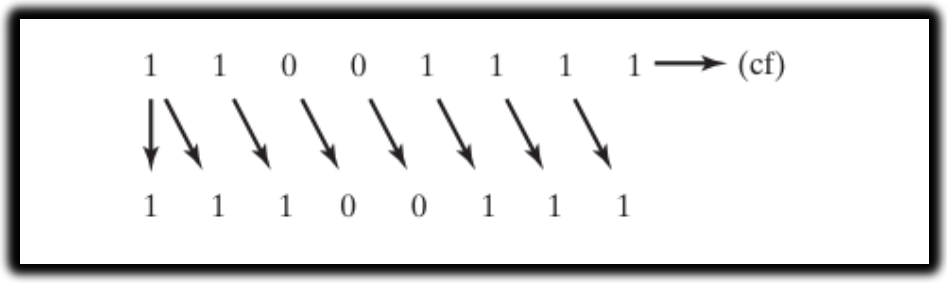
SAL/SAR (Arithmetic Shift Left/Arithmetic Shift Right):

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.

- Instead of filling empty spots with zeros or sign bits, rotation wraps bits around.
- ROL → bits move left, the leftmost bit reappears on the right.
- ROR → bits move right, the rightmost bit reappears on the left.
- Think of it like a circular conveyor belt for bits.

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.

- Similar to ROL/ROR, but they also include the **Carry flag** in the rotation.
- Useful for **multi-precision arithmetic** (when numbers are bigger than one register).
- Example: rotating across registers while keeping track of overflow.

✨ Key takeaway:

- **SAL/SAR** → shifts, with SAR keeping the sign.
- **ROL/ROR** → circular rotations.
- **RCL/RCR** → rotations that include the Carry flag for extended precision.

SHIFT LEFT AND SHIFT RIGHT

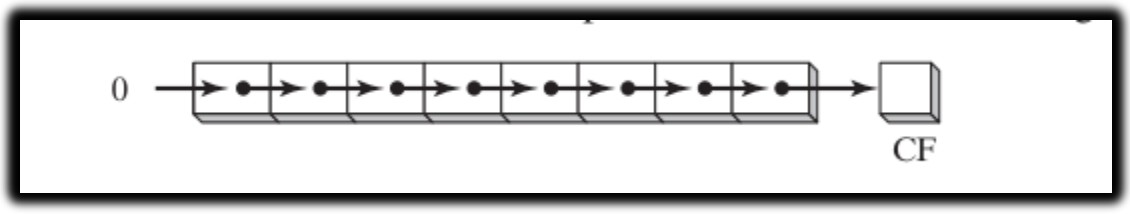
Bit shifting involves moving bits within an operand either to the left or right.

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:

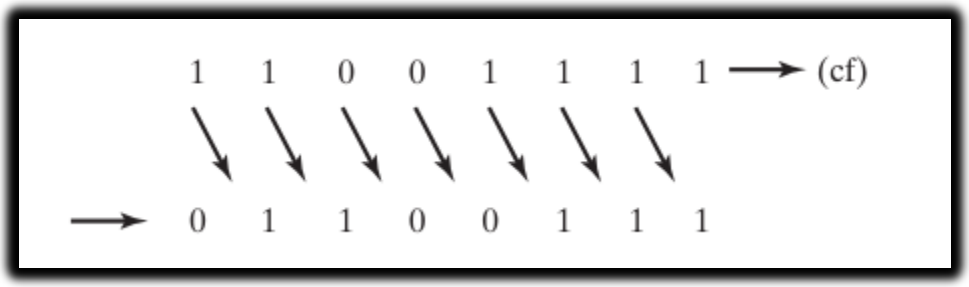
I. SHL (Shift Left)

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



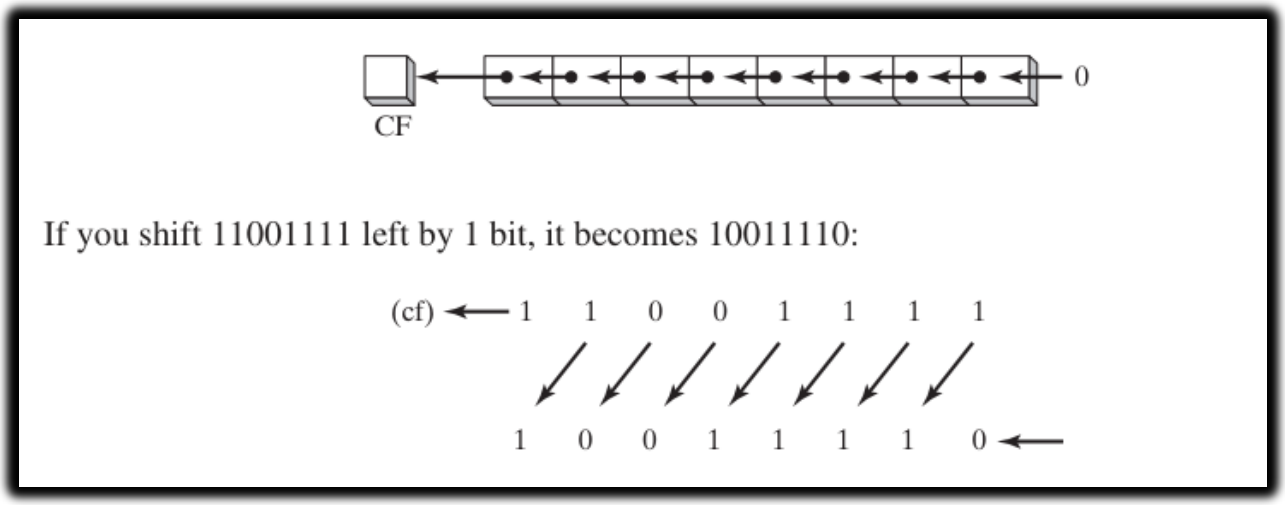
The following illustration shows a **single logical right shift** on the binary value 11001111, producing 01100111.

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:




```
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
```

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.

When a value is shifted right multiple times, the Carry flag stores the last bit shifted out of the least significant bit (LSB). Shifting left works like multiplication by powers of 2.

For example, shifting the binary number 00001010 (decimal 10) left by two positions is the same as multiplying it by $2^2 = 4$, giving 40.

Shifting left by one position is the same as multiplying by 2, shifting by three positions multiplies by $2^3 = 8$, and so on.

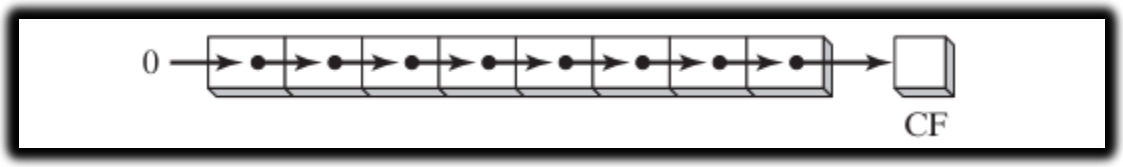
After a single SHL instruction on decimal 10, the BL register will contain 20.

Bitwise multiplication is widely used in graphics and signal processing — for scaling images, rotating them, or applying filters efficiently.

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

II. 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 (Shift Right)

- **What it really means:** Shifting a binary number to the **right by *n* bits** is the same as dividing it by 2n, with the result rounded down (floor division).
- **Key point:** This only works cleanly when the divisor is a **power of 2** (like 2, 4, 8, 16…). If the divisor is not a power of 2, a simple shift won't give the correct quotient.

Correct Example

- Dividend = 32 (binary 00100000).
- Divisor = 2.
- Shift right by 1 bit → 00010000 = 16.
- That matches 32÷2=16.
- Dividend = 32.
- Divisor = 4.
- Shift right by 2 bits → 00001000 = 8.
- That matches 32÷4=8.

Step-by-Step Process

1. Write the dividend in binary.
2. Decide how many bits to shift right (this equals the power of 2 divisor).
3. Perform the shift.
4. The result is the quotient, rounded down.

✨ Key takeaway:

Bitwise division by shifting right is a **fast shortcut** for dividing by powers of 2. It doesn't work for arbitrary divisors like 21 — only for divisors that are exact powers of 2.

Shifting a binary number **right by *n* bits** is the same as dividing it by 2^n.

Example:

- 32÷2=16 → achieved by shifting 32 right by 1 bit.
- 64÷4=16 → achieved by shifting 64 right by 2 bits.

⚠ Important: This only works for **powers of two**.

```
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
```

Write the dividend and divisor in binary form:

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

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

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

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

```
Quotient = 010000 = 16
```

Uses of SHR / SAR

- Fast division by powers of two.
- Efficient scaling in graphics and signal processing.
- Handling signed vs unsigned division correctly.
- Managing overflow with the Carry flag.

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
```

SUMMARY BEFORE WE CONTINUE...

SHR (Shift Right)

What happens: Bits move one position to the right.

The **leftmost bit** is replaced with 0.

The **rightmost bit** (LSB) is shifted out into the **Carry flag**.

Example:

- AX = 0x1234 → binary 0001 0010 0011 0100.
- SHR by 1 → 0000 1001 0001 1010 → hex 0x091A.
- Leftmost bit (1) is lost, rightmost bit (0) goes into Carry.

Uses:

- Fast division by 2 (or powers of 2 if shifted multiple times).
- Converting decimal values into binary form.
- Unpacking or manipulating binary data.
- Testing the Carry flag for conditional jumps.

SHL (Shift Left)

- **What happens:** Bits move one position to the left.
- The **rightmost bit** is replaced with 0.
- The **leftmost bit** (MSB) is shifted out into the **Carry flag**.
- Effectively multiplies the value by 2 for each shift.

Uses:

- Quick multiplication by powers of 2.
- Bitwise scaling in graphics or signal processing.

SAR (Shift Arithmetic Right)

- Similar to SHR, but **preserves the sign bit**.
- Used for signed integers so negative numbers stay negative after shifting.

Bitwise Division & Multiplication

- **Right shift (SHR/SAR):** Equivalent to dividing by 2n.
- **Left shift (SHL):** Equivalent to multiplying by 2n.
- These are inverse operations of each other.

SAL (Shift Arithmetic Left)

- Similar to **SHL** (Shift Left).
- Shifts all bits to the left, filling the rightmost bit with 0.
- The leftmost bit (MSB) is shifted out into the **Carry flag**.
- Effectively multiplies the value by 2n.
- Used for both signed and unsigned integers, but be careful: if the sign bit changes, the result may overflow.

Bitwise shifts are **binary-level operations**. Whether you're working with decimal or hexadecimal numbers, the shift acts on their binary representation.

That's why they're so versatile — they're the building blocks for fast math, data manipulation, and even low-level graphics tricks.

ARITHMETIC SHIFT RIGHT AND ARITHMETIC SHIFT LEFT

SAL (Shift Arithmetic Left) Instruction

Behavior: Works the same as **SHL (Shift Left)**.

Moves each bit in the operand one position to the left.

Example: 11001111 → after one left shift → 10011110.

Carry Flag: The most significant bit (MSB) that gets shifted out is stored in the **Carry flag**.

Lowest Bit: The least significant bit (LSB) is always filled with 0.

Discarding Carry: The bit in the Carry flag is not retained in the operand — it's "lost" from the register.

Use Cases:

- Multiplying values by powers of 2.
- Logical left shifts in binary manipulation.
- Common in arithmetic, graphics, and signal processing tasks.

✨ **Key takeaway:** SAL and SHL are functionally identical on x86. Both multiply by 2n when shifting left, but SAL is named to emphasize its **arithmetic meaning** rather than just logical bit movement.

No need to over-explain, we saw all these in the earlier chapters….

SAR (Shift Arithmetic Right)

Shifts bits to the right while **preserving the sign bit**.

Operation:

- Bits move rightward.
- The **MSB (sign bit)** stays the same (0 for positive, 1 for negative).
- The **LSB** is shifted into the **Carry flag**.

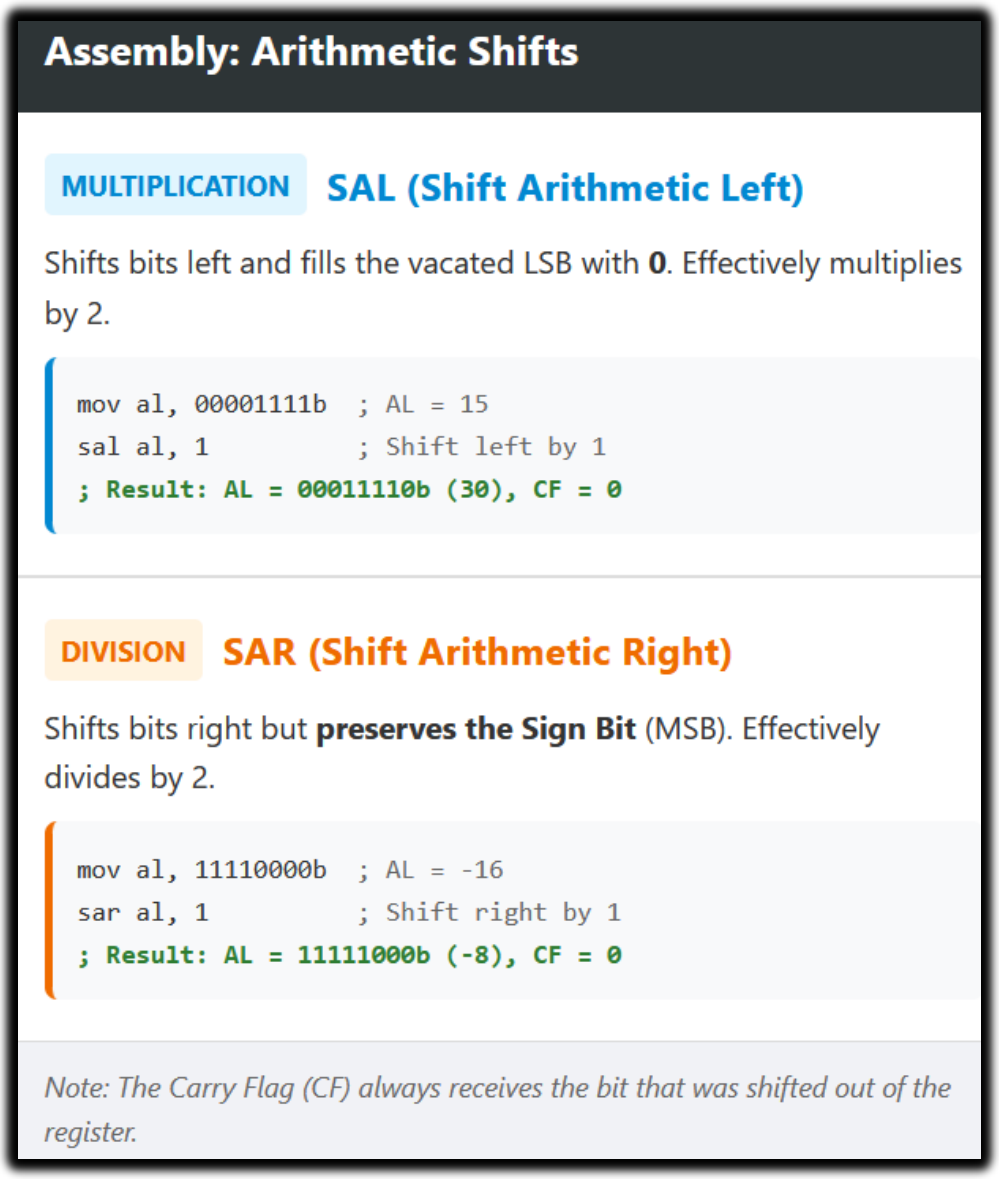
Effect: Ensures signed numbers remain correctly signed after shifting.

Use Cases:

- Signed division by powers of 2.
- Arithmetic operations where sign must be preserved.
- Sign-extension (e.g., extending AX into EAX).

Examples

- **Sign Bit Duplication:** If AL is negative, SAR keeps it negative after shifting.
- **Signed Division:** –128 shifted right by 3 → –16 (equivalent to dividing by 23).
- **Sign-Extend AX → EAX:** Shift EAX left by 16, then SAR right by 16 → sign is preserved.



ROTATE LEFT AND ROTATE RIGHT

I. ROL (Rotate Left) Instruction

The **ROL** instruction performs a circular bit rotation to the left.

This means bits are shifted left, but the leftmost bit (most significant bit) wraps around to the rightmost bit (least significant bit), and the carry flag (CF) gets updated.

How It Works: When you perform a left rotation, each bit in the operand moves left. The highest bit is copied into the Carry flag (CF) and the lowest bit position.

Example:

Let's say we start with the value 01000000b (in register AL), and we perform 3 left rotations:

```
AL = 01000000b (Before)
ROL AL, 1      (1st rotation) -> AL = 10000000b, CF = 0
ROL AL, 1      (2nd rotation) -> AL = 00000001b, CF = 1
ROL AL, 1      (3rd rotation) -> AL = 00000010b, CF = 1
```

Multiple Rotations: The **Carry flag** stores the last bit rotated out of the most significant bit (MSB).

Exchanging Bit Groups: You can swap the upper and lower halves of a byte using **ROL**. For example, rotating 26h (00100110b) 4 bits will swap its nibbles:

```
26h (00100110b) rotated by 4 bits -> 26h (01100010b)
```

This is useful for data reordering or manipulation tasks.

II. ROR (Rotate Right) Instruction

The **ROR** instruction is the reverse of **ROL**—it performs a circular bit rotation to the right.

Each bit shifts right, with the lowest bit wrapping around to the leftmost position, and the Carry flag gets updated.

How It Works: When you rotate right, each bit shifts to the right, and the lowest bit is copied into both the Carry flag (CF) and the highest bit position.

Example:

If we start with 00000001b (in register AL) and perform two right rotations:

```
AL = 00000001b (Before)
ROR AL, 1      (1st rotation) -> AL = 10000000b, CF = 1
ROR AL, 1      (2nd rotation) -> AL = 11000000b, CF = 1
```

- **Multiple Rotations**: After multiple rotations, the **Carry flag** stores the last bit shifted out from the least significant bit (LSB).
- **Use Case**: **ROR** is great for working with low-to-high end byte manipulations and helps with certain encryption algorithms.

III. RCL (Rotate Carry Left) Instruction

The **RCL** instruction performs a bitwise rotation to the left, just like **ROL**, but with one key difference: it incorporates the **Carry flag** (CF) into the rotation.

How It Works: Each bit in the operand shifts 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:

Start with 10001000b in register BL, and perform 2 **RCL** rotations:

```
BL = 10001000b (Before)
RCL BL, 1      (1st rotation) -> BL = 00010000b, CF = 1
RCL BL, 1      (2nd rotation) -> BL = 00100000b, CF = 1
```

Recovering Bits from CF: **RCL** can be used to recover a bit previously shifted into the Carry flag (CF). If the LSB is 1, a jump can be made; otherwise, the original value is restored.

IV. RCR (Rotate Carry Right) Instruction

The **RCR** instruction is the reverse of **RCL**—it rotates bits to the right while incorporating the **Carry flag** (CF).

How It Works: Each bit shifts right, and the Carry flag is copied into the most significant bit (MSB), while the least significant bit (LSB) moves into the Carry flag.

Example:

First, set the **Carry flag** to 1 using the **STC** instruction. Then perform a right rotation on register AH:

```
STC              ; Set Carry flag to 1
AH = 00000001b   ; Initial value
RCR AH, 1        ; 1st rotation -> AH = 10000000b, CF = 1
```

Use Case: This instruction is useful when working with bit-level encryption and precise bit manipulation where you need to handle the Carry flag.

V. Visualizing the Integer with Carry Flag

When using rotate or shift instructions, it can be helpful to visualize the integer as a 9-bit value. The **Carry flag** is treated as an extra bit in the operation.

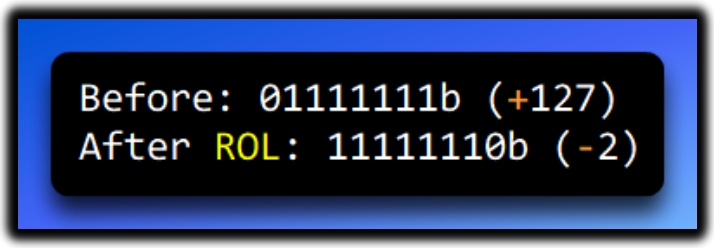
Example: If the **Carry flag** is set before performing a **RCR** operation, it influences the resulting bits in the operand:

```
RCR AH, 1        ; The Carry flag is incorporated into the rotation
```

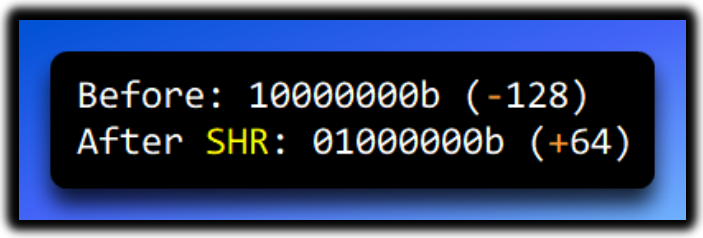
VI. Signed Overflow

When performing shifts or rotations on signed integers, the **Overflow flag** (OF) plays an important role. If the operation causes a signed number to exceed its range (e.g., from positive to negative), the **OF** will be set.

Positive Integer Becoming Negative (ROL): A positive integer, like +127, will turn negative when rotated left:



Negative Integer Changing Sign (SHR): A negative integer, like -128, will flip its sign when shifted right:



Note: The Overflow flag is **undefined** if the shift or rotation count is greater than 1, so keep that in mind when working with multi-bit operations.

These **Rotate** and **Shift** instructions are powerful tools in assembly programming, allowing you to manipulate bits.

SHIFT LEFT DOUBLE AND SHIFT RIGHT DOUBLE

I. SHLD (Shift Left Double)

The **SHLD** instruction shifts the bits of the **destination operand** (the value you want to shift) to the **left** by a certain number of positions.

But instead of just filling the empty spots with zeros (like you would with a regular shift), it fills them with the **most significant bits** of the **source operand**.

Here's the important part:

- **Destination operand**: This is the value you want to shift.
- **Source operand**: This is another value that provides the bits to fill the empty spaces when the destination operand is shifted.

Example:

Let's say we have two 32-bit values:

- **Destination operand**: destination = 0x12345678 (in binary: 00010010001101000101011001111000)
- **Source operand**: source = 0xA5A5A5A5 (in binary: 10100101101001011010101001010101)

If we shift the destination operand left by 8 bits using **SHLD** with the source operand:

```
SHLD destination, source, 8
```

- The **destination operand** will shift left by 8 bits.
- The **leftmost 8 bits** (the most significant bits) from the **source operand** will be copied into the empty bit positions.

After the operation:

Destination operand: The left part of the destination gets shifted left, and the open spots are filled with the top 8 bits of the source. So, the final result might look something like this:

destination = 0x34D5A5A5
(or something similar, depending on the number of bits you shift by)

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:

```
21 mov eax, 0x12345678
22 mov ebx, 0xabcdef00
23 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.

Key Takeaways

- **SHLD** shifts the **destination** operand to the left.
- The empty bits are filled with bits from the **source operand** (not zeros).
- The **source operand** itself is **not affected** by this operation.

II. SHRD (Shift Right Double) Instruction

The **SHRD** instruction is like the opposite of **SHLD**, but it works with **right shifts** instead of left. It shifts the **destination operand** to the **right** by a specified number of bits, and the newly opened-up positions are filled with the **least significant bits** from the **source operand**.

- **Destination operand**: This is the value you're shifting to the right.
- **Source operand**: This provides the bits to fill the empty positions created in the **destination operand**.
- **Count**: This is the number of bits you want to shift.

Syntax:

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

Where:

- **dest** is the destination operand.
- **source** is the source operand.
- **count** is how many bits you want to shift.

Example:

Let's say we have two 32-bit values:

- **Destination operand**: destination = 0x12345678 (in binary: 00010010001101000101011001111000)
- **Source operand**: source = 0xA5A5A5A5 (in binary: 10100101101001011010101001010101)

If we shift the destination operand right by 8 bits using **SHRD** with the source operand:

```
SHRD destination, source, 8
```

- The **destination operand** will shift to the right by 8 bits.
- The **rightmost 8 bits** (the least significant bits) from the **source operand** will fill the empty bit positions on the left side of the **destination operand**.

Special Cases:

- **Count = 0**: If you set **count** to 0, the **destination operand** won't shift at all.
- **Count = 31**: If **count** is 31, the **destination operand** will be shifted all the way to the right, and the entire **source operand** will be copied into the **destination operand**.

Key Points:

- **SHRD** performs a right shift on the **destination operand**.
- It uses bits from the **source operand** to fill in the empty positions.
- The **source operand** itself is not changed.
- **Count** defines how many positions the shift will happen. It must be between 0 and 31.

Status Flags After SHRD Instruction

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:

```
41 mov eax, 0x12345678
42 mov ebx, 0xabcdef00
43 SHRD eax, ebx, 1
44
45 ;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.

Practice Questions on Shift & Rotate Instructions

1. **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?**
2. **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?**
3. **Which instruction performs the following operation (CF = Carry flag)?** *Rotates the bits to the left through the Carry flag.*
4. **What happens to the Carry flag when the** SHR AX,1 **instruction is executed?**
5. **Challenge:** Write a series of instructions that shift the lowest bit of AX into the highest bit of BX **without using SHRD**. Then, perform the same operation **using SHRD**.
6. **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 code that does this, and set the Parity flag accordingly.
    - If the count is even, PF should be set; if it's odd, PF should be cleared.

SHIFTING MULTIPLE DOUBLEWORDS

Shifting Multiple Doublewords

In assembly programming, extended-precision integers are often stored in arrays of bytes, words, or doublewords. To work with them correctly, it's important to understand how they're laid out in memory.

On x86 systems, values are stored in **little-endian order**. This means the lowest-order byte comes first at the starting address, followed by the next higher byte, and so on, until the highest-order byte is stored last. This rule applies whether you're dealing with bytes, words, or doublewords.

Now, consider the process of shifting an array of bytes one bit to the right:

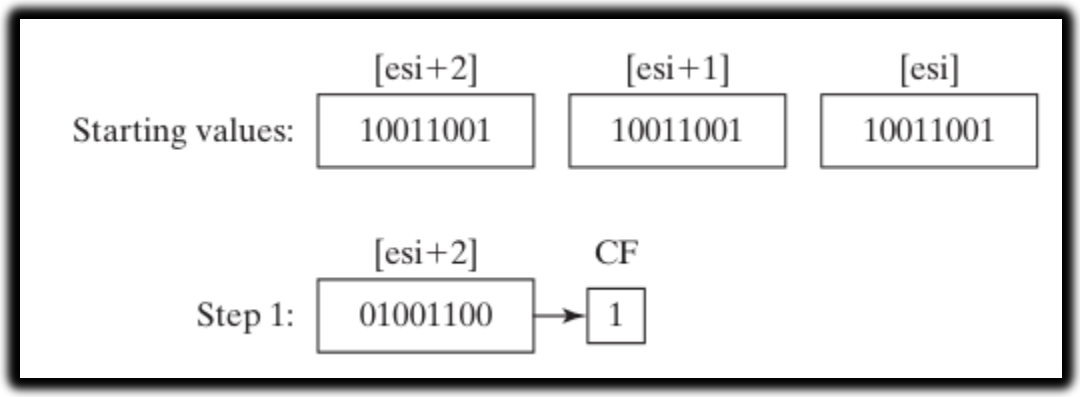
Step 1:

Begin by shifting the highest byte, located at [ESI+2], to the right. When this happens, the lowest bit of that byte is automatically copied into the **Carry flag**. This is the standard behavior of instructions like **SHR (Shift Right)** in x86 assembly.

To illustrate this, you can write the operation in assembly code, assuming the **ESI register** points to the base address of the array.

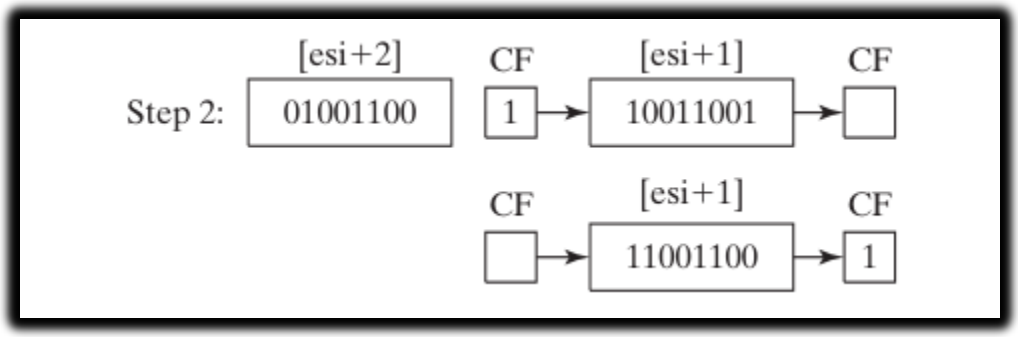
```
SHR byte ptr [ESI+2], 1
```

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.



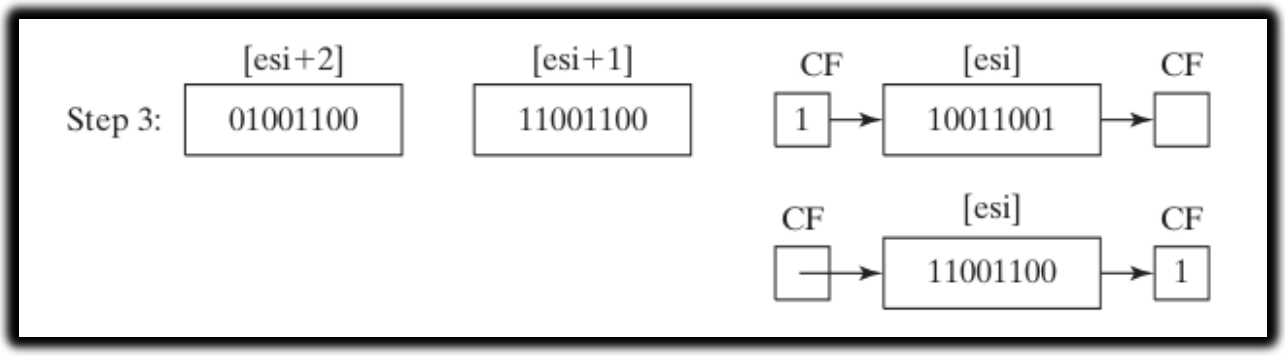
Step 2:

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:

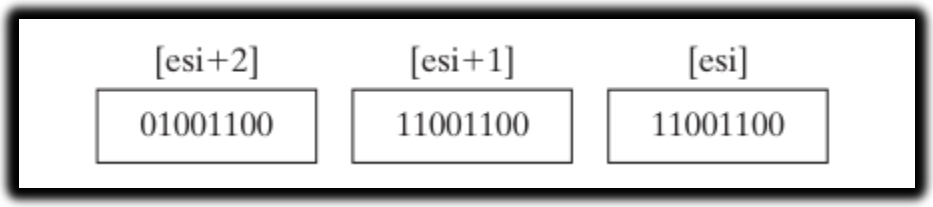


Step 3:

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.
```

What's Really Happening in This Code

- First, the program sets up an array of three bytes, each with the value 99h (binary 1001 1001).
- The ESI register is initialized to point at the start of that array.

Now the fun part — the shifting:

1. **Top byte gets nudged right**
    - The highest byte (array[2]) is shifted right by one bit.
    - Its lowest bit doesn't just vanish — it's copied into the **Carry flag**.
2. **Middle byte rotates with Carry**
    - The middle byte (array[1]) rotates right.
    - That rotation pulls in the bit from the high byte's Carry, so the chain continues.
3. **Lowest byte rotates with Carry too**
    - Finally, the lowest byte (array[0]) rotates right.
    - It picks up the bit that was shifted out of the middle byte, again through the Carry flag.

End Result

The entire array has been shifted one bit to the right **as if it were a single multi-byte number**. The Carry flag now holds the "extra" bit that fell off the very end.

Why It's Cool

- This trick lets you treat arrays of bytes like big integers.
- With a loop, you can scale it up to words, doublewords, or even huge arrays.
- It's the assembly equivalent of saying: *"Shift this whole big number right by one, even though it's split into pieces."*

BINARY MULTIPLICATION WITH BIT SHIFTS (ASSEMBLY STYLE)

🚀 Why bother?

Multiplying in assembly doesn't always need the heavy MUL instruction. If your multiplier is a power of 2, you can cheat: just shift bits left. It's faster, simpler, and the CPU loves it.

I. Power of 2 Multipliers

SHL (Shift Left) moves bits left.

Each left shift = multiply by 2.

Shift by *n* bits = multiply by 2n.

Example:

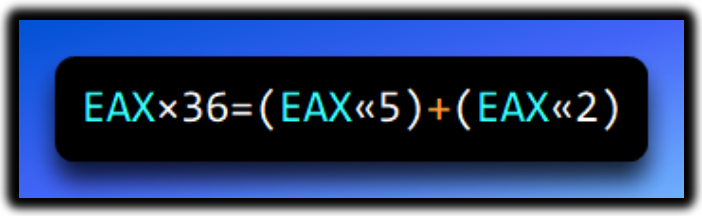
- EAX = 5
- SHL EAX, 1 → result = 10 (5 × 2).
- SHL EAX, 3 → result = 40 (5 × 8).

II. Non-Power of 2 Multipliers

Not every multiplier is a neat power of 2. Trick: break it down into sums of powers of 2.

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

So,



III. Example: 123 × 36 🧩

1. Load 123 into EAX.
2. Copy EAX into EBX.
3. SHL EAX, 5 → multiply by 32.
4. SHL EBX, 2 → multiply by 4.
5. Add them together → result = 4428.

```
          01111011      123
    ×     00100100      36
    ----------------
          01111011      123 SHL 2
+   01111011            123 SHL 5
    ----------------
  0001000101001100      4428
```

IV. General Algorithm (Unsigned 32-bit Multiply)

1. Start with product = 0.
2. For each bit in the multiplier (LSB → MSB):
   - If the bit is **1**, shift the multiplicand left by that bit index and add it to product.
   - If the bit is **0**, skip.
3. Return product.

```
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 way, multiplication becomes just **shifts + adds**. It's like building multiplication out of Lego blocks instead of firing up a bulldozer (MUL).

V. 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 create a function called **multiply** that takes two numbers—**multiplicand** and **multiplier**—and calculates their product. Inside the function, we start by setting a **product** variable to 0, which will hold our final result.

Then, we use a loop that runs 32 times, one for each bit of the **multiplier**. For each iteration, we check if the current bit of the multiplier is set to 1.

If it is, we shift the **multiplicand** to the left by a number of bits equal to the current iteration (this is like multiplying the number by powers of 2), and then add that result to the **product**.

Once the loop finishes, the **product** contains the final result of the multiplication.

In the main part of the program, you can change the **multiplicand** and **multiplier** to whatever numbers you want to multiply. When you run the program, it calculates the product and displays the result.

The following assembly code implements this algorithm:

```
183 multiply:
184   push    ebp
185   mov     ebp, esp
186   mov     eax, multiplicand
187   mov     ebx, multiplier
188   xor     ecx, ecx
189   mov     ecx, 31
190 loop:
191   shl     eax, 1
192   test    ebx, 1
193   jz      next
194   add     eax, ebx
195 next:
196   dec     ecx
197   jnz     loop
198   mov     esp, ebp
199   pop     ebp
200   ret
```

This little routine is basically a DIY multiplication function.

Instead of using the MUL instruction, it builds the product by shifting and adding, bit by bit.

The final answer ends up in **EAX**, which is the usual "return value" register in assembly.

1. **Function setup**
    - push ebp / mov ebp, esp → classic stack frame boilerplate. Think of it as the function saying: *"Okay, I'm ready to work, let's save the current state."*
2. **Load the numbers**
    - mov eax, multiplicand → put the first number in EAX.
    - mov ebx, multiplier → put the second number in EBX.
3. **Prepare the loop**
    - mov ecx, 31 → we're going to check all 32 bits of the multiplier.
    - ECX acts as the countdown timer for the loop.

4. **The loop itself**
    - shl eax, 1 → shift left, doubling EAX each time.
    - test ebx, 1 → look at the lowest bit of EBX.
    - If that bit is **1**, we add eax, ebx → meaning "yep, include this chunk in the product."
    - If it's **0**, we skip.
    - dec ecx / jnz loop → keep going until all bits are processed.
5. **Cleanup**
    - mov esp, ebp / pop ebp → restore the stack frame.
    - ret → return, with the product sitting in EAX.

This is the **shift-and-add algorithm** for multiplication. It's like long multiplication in grade school, but instead of writing numbers on paper, you're sliding bits around and adding them up.

BINTOASC

Converting an Integer to a Binary String (Assembly Procedure)

```
206 BinToAsc:
207   push    ebp
208   mov     ebp, esp
209   mov     eax, binary_integer
210   xor     ecx, ecx
211   mov     ecx, 31
212 loop:
213   shl     eax, 1
214   adc     ecx, ecx
215   mov     edx, eax
216   cmp     dl, 32
217   jb      ascii_zero
218   mov     dl, dl - 32
219 ascii_zero:
220   mov     [edi], dl
221   inc     edi
222   dec     ecx
223   jnz     loop
224   pop     ebp
225   ret
```

Here's what the procedure is really doing:

1. **Bit by bit scan** It starts at the **most significant bit** of the integer and works its way down.
2. **Shift and carry** Each time, the integer is shifted left by one. The **Carry flag** keeps track of whether a 1 was pushed out during the shift.
3. **Deciding between '0' and '1'**
    - If the bit shifted out was 0, the Carry flag is clear, and the procedure writes the ASCII '0' (0x30) into the buffer at EDI.
    - If the bit shifted out was 1, the Carry flag is set, and the procedure writes the ASCII '1' (0x31) into the buffer.
4. **Building the string** This repeats for every bit. By the end, the buffer at EDI holds the full ASCII binary string representation of the original integer.

Big Picture

It's basically a loop that "peels off" each bit of the number and writes it as a character '0' or '1'. When it's done, you've got a neat binary string sitting in memory, ready to be printed or used.

Example 2 – Using BinToAsc

This snippet demonstrates how to call the **BinToAsc** procedure to convert the binary integer 123 (01111011) into its ASCII binary string form:

```
229 mov     eax, 123
230 call    BinToAsc
231
232 ;The buffer at the address specified by the register `EDI`
233 ;will now contain the ASCII binary string "01111011".
```

Why It's Useful

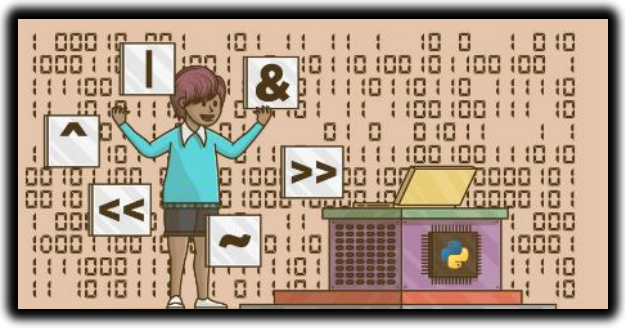
The **BinToAsc** procedure provides a quick way to turn raw binary values into human-readable strings. This makes it handy for:

- Displaying binary data on the console.
- Writing binary representations into files.
- Debugging or inspecting values during low-level programming.

EXTRACTING FILE DATE FIELDS

When you're working with bit-level operations, **shifting** and **masking** are your best friends for extracting specific bit strings from larger data values.

Shifting: Moves bits around to get them in the right place.



Masking: Clears out unwanted bits, so you're left with only the bits you care about.

I. Using the AX Register for Extracting Date Fields

The **AX register** is a great choice for extracting bits because it's 16 bits wide, meaning it can hold two 8-bit bytes.

This is particularly useful when you need to pull out different parts of a value, like the **day**, **month**, and **year** from a date stamp that's packed into a 16-bit integer.

Here's a quick overview of how you can use assembly to extract the **day**, **month**, and **year** from a date stamp stored in the **DX** register:

1. **Day**: Bits 0-4 of the DL register hold the day of the month.
2. **Month**: Bits 5-8 of the DX register hold the month.
3. **Year**: Bits 9-15 of the DH register hold the year (relative to 1980).

II. Example Code to Extract Date Fields from a Date Stamp

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

```
; Assuming DX contains the date stamp
; The day, month, and year will be extracted and stored in corresponding variables

; Extract Day (bits 0-4)
mov al, dl          ; Copy the DL register (lower byte of DX) into AL
and al, 1Fh         ; Mask off all bits except for the first 5 bits (day)
mov [day], al       ; Store the extracted day in the 'day' variable

; Extract Month (bits 5-8)
mov ax, dx          ; Copy the entire DX register into AX
shr ax, 5           ; Shift AX to the right by 5 to move the month to the low part
and al, 0Fh         ; Mask off all bits except for the first 4 bits (month)
mov [month], al     ; Store the extracted month in the 'month' variable

; Extract Year (bits 9-15)
mov al, dh          ; Copy the DH register (high byte of DX) into AL
shr al, 1           ; Shift the year to the low byte and clear AH
add al, 1980        ; Add 1980 to get the full year
mov [year], al      ; Store the extracted year in the 'year' variable
```

III. Multiplying EAX by 24 and 21 Using Binary Multiplication

To multiply **EAX** by **24** and **21**, we can use **binary multiplication** based on bit shifts. Let's first see how you can multiply **EAX by 24**:

Multiply by 24:

24 can be represented as $2^3 + 2^4$, so you can use shifts to multiply by 8 (2^3) and 16 (2^4), and then add them together.

```
; Multiply EAX by 24
shl eax, 3          ; EAX = EAX * 8
mov ebx, eax        ; Copy EAX into EBX
shl ebx, 1          ; EBX = EAX * 16 (shifted by 1 more)
add eax, ebx        ; EAX = EAX * 24 (adding the two results)
```

Multiply by 21: 21 can be represented as $2^4 + 2^2 + 2^1 + 2^0$, so you can use shifts and additions similarly.

```
; Multiply EAX by 21
mov ebx, eax          ; Copy EAX to EBX
shl eax, 4            ; EAX = EAX * 16
shl ebx, 2            ; EBX = EAX * 4 (shifted by 2)
add eax, ebx          ; EAX = EAX * 20
shl ebx, 1            ; EBX = EAX * 2 (shifted by 1)
add eax, ebx          ; EAX = EAX * 21
```

IV. Displaying Binary Bits in Reverse Order

To display the binary bits of a value in reverse order in the BinToAsc procedure, you simply need to reverse the loop that processes each bit.

Instead of starting from the **most significant bit (MSB)** and working towards the **least significant bit (LSB)**, you can iterate from the **least significant bit (LSB)** to the **most significant bit (MSB)**.

Here's how to modify the loop:

```
BinToAscReverse:
    ; We assume EAX holds the value to be converted
    mov ecx, 32           ; Start with 32 bits to process
    mov ebx, 0            ; Clear EBX (to store the reversed binary)
    mov esi, 0            ; Clear the index counter for output

reverse_loop:
    ; Get the least significant bit
    test eax, 1           ; Test if the LSB is set
    jz no_bit_set         ; If it's 0, skip setting the bit
    or ebx, 1             ; Set the corresponding bit in EBX
no_bit_set:
    shl ebx, 1            ; Shift EBX left for the next bit
    shr eax, 1            ; Shift EAX right to examine the next bit
    loop reverse_loop     ; Repeat until all bits are processed
    ; Now EBX contains the reversed binary representation of the number
    ; Add the necessary code to convert EBX to ASCII or display it
```

V. Extracting Minutes from a Time Stamp (File Directory Entry)

To extract the minutes from a time stamp that uses bits 5 through 10 for the minutes, here's the assembly code:

```
; Assuming the time stamp is in a 16-bit register like DX
; Extract minutes (bits 5-10)
mov ax, dx          ; Copy the time stamp to AX
shr ax, 5           ; Shift the minutes to the right, aligning bits 5-10
and al, 3Fh         ; Mask off all but the least significant 6 bits (minutes)
mov [bMinutes], al  ; Store the extracted minutes in bMinutes
```

- **Shifting**: shr ax, 5 shifts the bits to the right by 5, so bits 5-10 align with the lower part of the register.
- **Masking**: and al, 3Fh ensures that only the 6 bits representing the minutes are kept, clearing any other bits.

VI. Summary

- **Shifting** and **masking** are key operations for extracting specific bit strings, such as date fields or time stamps.
- **Multiplying** by constants like 24 or 21 can be done efficiently with **bit shifts** and **additions**.
- To **reverse the order of binary bits**, you simply reverse the bit-checking loop in your conversion function.
- **Extracting specific fields** (like minutes) from a timestamp can be done by **shifting** and **masking** the relevant bits.

MUL OPERATOR

I. Understanding the MUL Instruction for Unsigned Integer Multiplication

The **MUL** instruction in assembly is used to multiply unsigned integers. It has different versions for different operand sizes:

- **AL**: 8-bit multiplication (8-bit operand)
- **AX**: 16-bit multiplication (16-bit operand)
- **EAX**: 32-bit multiplication (32-bit operand)

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

II. How the MUL Instruction Works

The **MUL** instruction multiplies the operand with a specific register (AL, AX, or EAX), and stores the result in a larger destination register.

The key point is that the product will always be **twice the size** of the operand (since multiplication with an unsigned number could result in a value that overflows the original size).

Here's the rule for the default destination of the product:

- **8-bit operand**: Product is stored in **AX**.
- **16-bit operand**: Product is stored in **DX:AX** (combined 32-bit result).
- **32-bit operand**: Product is stored in **EDX:EAX** (combined 64-bit result).

III. Overflow Detection with the Carry Flag

The **MUL** instruction does **not generate an overflow exception** if the result doesn't fit into the destination registers. However, if the product is too large to fit into the destination register, the **Carry flag (CF)** is set.

For example:

- When you multiply two **16-bit numbers** (e.g., in AX), the product will be **32 bits**.
  - The lower 16 bits go into **AX**.
  - The upper 16 bits go into **DX**.

If the **DX** register is **not zero**, it indicates that the product is too large to fit into **AX** alone, and the **Carry flag** will be set.

IV. Code to Use the MUL Instruction - Multiplying Two 16-bit Numbers

Let's consider two 16-bit numbers, where one is in **AX** and the other is in memory.

```
; Example: Multiply AX by the value at memory location MY_DATA
mov ax, [MY_DATA]  ; Load the operand into AX
mov bx, [MULTIPLE] ; Load the multiplier into BX

mul bx             ; Multiply AX by BX (16-bit multiplication)

; After MUL:
; DX:AX contains the product
; If DX is not 0, the product is too large for AX alone
; Check the Carry flag:
jc overflow_detected ; If the Carry flag is set, handle overflow
```

Explanation:

- **MUL BX** multiplies **AX** by **BX** and stores the result in **DX:AX**.
- **JC (jump if carry)** checks the Carry flag. If set, the upper part of the product (in **DX**) is non-zero, indicating overflow, and you should handle it accordingly.

V. Example 2: Handling Overflow

If the **Carry flag** is set, the upper half of the result is non-zero. You can use the **DX** register to store the full 32-bit product.

```
; Example: Handle overflow after multiplication
mul bx            ; Multiply AX by BX (16-bit multiplication)

jc overflow_detected  ; If the Carry flag is set, the product overflows

; If no overflow (Carry flag is clear), we can safely ignore DX
mov [RESULT], ax   ; Store the lower 16 bits of the product

overflow_detected:
; If the Carry flag is set, handle the overflow:
mov [HIGH_RESULT], dx ; Store the upper 16 bits of the product in HIGH_RESULT
```

- **mul bx**: This multiplies **AX** by **BX**, and stores the result in **DX:AX**.
  The **lower 16 bits** of the product go into **AX**.
  The **upper 16 bits** go into **DX**.
- **jc overflow_detected**: This checks the **Carry flag**. If it's set, it means the product doesn't fit into **AX**, so **DX** contains the upper half of the product.
- If there's no overflow (i.e., **Carry flag is clear**), you can safely store the result in **AX**. If there's overflow, you need to consider both **DX** and **AX** for the full result.

Why Check the Carry Flag?

- The **Carry flag** is crucial when you need to determine whether the product can fit into the lower half of the result register (e.g., **AX** for 16-bit multiplication).
- If the **Carry flag is clear**, it means the upper half of the product is zero, and you can safely use just the **AX** register.
- If the **Carry flag is set**, it means the upper half of the product is non-zero, and you need to use both **DX** and **AX** (or **EDX** and **EAX** for 32-bit multiplication) to store the complete result.

VI. Final Example: Complete 16-bit Multiplication with Overflow Handling

```
;Example: Multiply two 16-bit numbers with overflow detection

mov ax, [operand1]    ; Load the first operand into AX
mov bx, [operand2]    ; Load the second operand into BX

mul bx                ; Perform unsigned multiplication (AX * BX)

; Check if overflow occurred (i.e., if DX is non-zero)
jc overflow_detected  ; Jump to overflow detection if Carry flag is set

; No overflow, store result in AX
mov [result], ax      ; Store the lower 16 bits of the product

overflow_detected:
; Overflow happened, store both parts of the result
mov [high_result], dx ; Store the upper 16 bits of the product in DX
```

VII. Conclusion

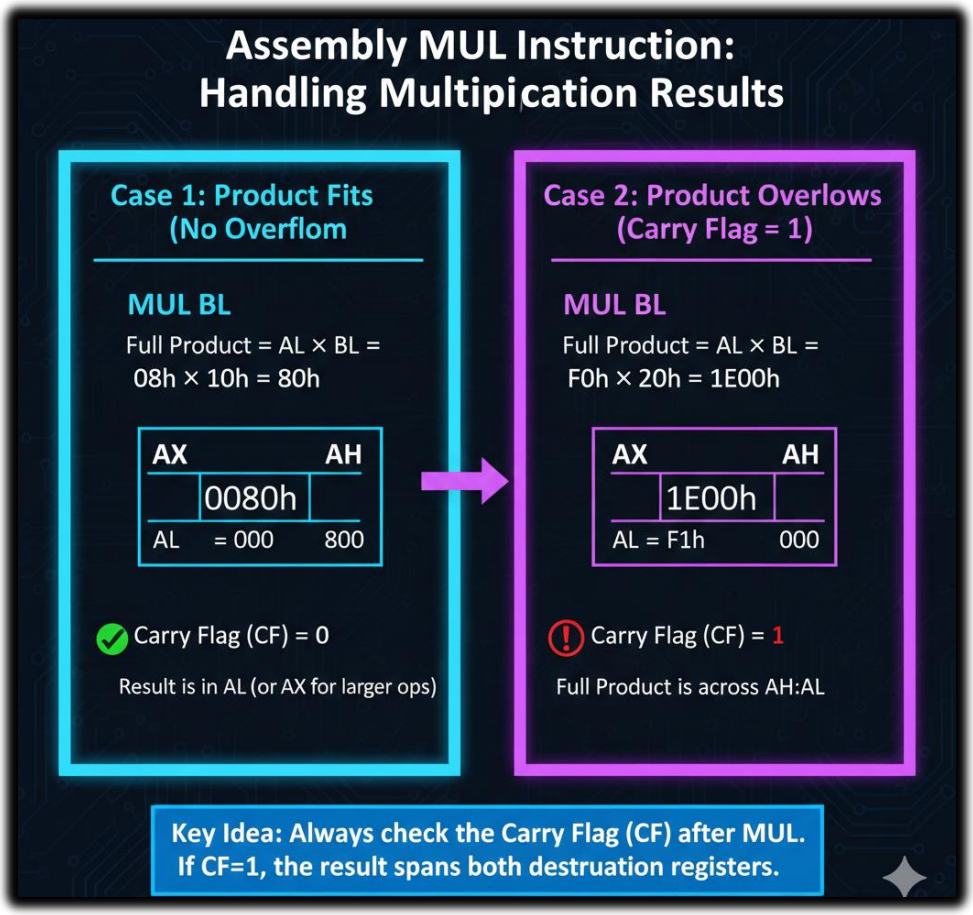
The **MUL** instruction handles unsigned integer multiplication and places the result in registers that are twice the size of the operand.

If the product fits neatly into the lower register, you're done.

But if it doesn't, the **Carry flag** will be set, signaling that the result overflowed and now spans both the upper and lower halves of the destination register.

That's why it's important to always check the Carry flag after using **MUL**.

It tells you whether you need to grab the full product from both registers instead of just the lower half.



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

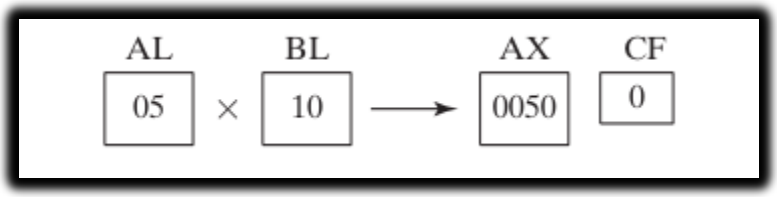
```
311 mov ax, 1000h   ;load first operand into AX
312 mov bx, 2000h   ;load second operand into BX
313 mul bx ; multiply AX by BX
314
315 ;check the Carry flag
316 jc overflow     ;jump to overflow handler if the Carry flag is set
317
318 ;the upper half of the product is zero, so we can ignore it
319 ;use the AX register to store the lower half of the product
```

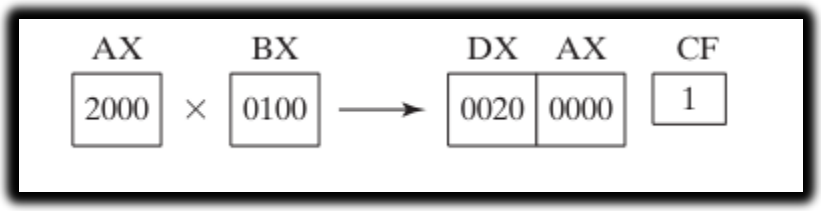
VIII. Example 1:

```
; 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* numbers.

When it does this, the result can be bigger than what fits in a single register, so the CPU splits the answer across two registers.

The **lower half** of the result goes into **EAX**, and the **upper half** goes into **EDX**.

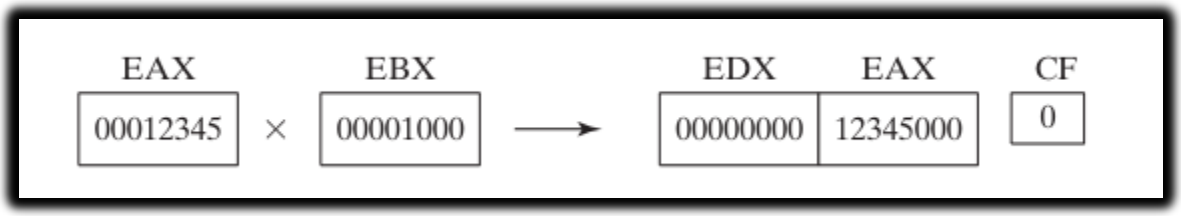
If the multiplication produces a value that's too big to fit, the **Carry flag** is set to let you know there was overflow.

In this example, multiplying **12345h** by **1000h** gives **12345000h**. That result fits cleanly across **EDX:EAX**, so there's no overflow—and the Carry flag stays clear.

The diagram below shows how the result is divided and placed into the two registers.

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

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



The **MUL** instruction multiplies two *unsigned* numbers. One of those numbers is always taken from a specific register: **AL** for 8-bit operations, or **AX** for 16-bit operations.

This value is called the *multiplicand*.

The second number, called the *multiplier*, can come from another register or from memory.

Because multiplication can produce a result that's larger than the original operands, the CPU splits the product across two registers.

For an 8-bit multiply, the lower part of the result goes into **AL** and the upper part goes into **AH**.

For a 16-bit multiply, the lower half ends up in **AX**, while the upper half is placed in **DX**.

If the final result is too big to fit cleanly into these registers, the processor sets the **Carry flag** to signal that an overflow occurred.

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:

```
359 mov rax, 0FFFF0000FFFF0000h
360 mov rbx, 2
361 mul rbx
362
363 ; 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:

```
367 .data
368     multiplier QWORD 10h
369
370 .code
371     mov rax, 0AABBBBCCCCDDDDh
372     mul multiplier
373
374     ;RDX:RAX = 00000000000000000AABBBBCCCCDDDD0h
```

In this example, the multiplication produces a **128-bit result**. However, the value is still small enough that both halves fit neatly into **RAX** and **RDX**, since the product is less than 2^{128}.

When the **MUL** instruction runs in **64-bit mode**, here's what's really happening under the hood:

The processor multiplies the value in **RAX** by the specified operand. The result is a full 128-bit number. The **lower 64 bits** of that result are placed into **RAX**, while the **upper 64 bits** are stored in **RDX**.

If the upper half of the result isn't zero—meaning the value is too large to fit entirely in **RAX**—the processor sets the **Carry flag** to indicate an overflow. In this case, everything fits cleanly, so no overflow occurs.

IMUL OPERATOR

IMUL Instruction (Signed Multiplication)

The **IMUL** instruction is used for **signed integer multiplication**. Unlike **MUL**, which treats all values as unsigned, **IMUL** respects the sign of the numbers, so positive and negative values behave the way you'd expect.

How IMUL Works

When **IMUL** performs a multiplication, it keeps the sign of the result correct by **sign-extending** the value. In simple terms, the processor copies the sign bit (the highest bit) into the upper part of the result so that the final product has the proper sign.

I. IMUL Instruction Formats

IMUL comes in three common formats:

- **One-operand form**
  Multiplies the operand by itself and stores the full result in the implicit destination registers (for example, AX, DX:AX, or EDX:EAX depending on operand size). This is effectively squaring the value.
- **Two-operand form**
  Multiplies two values and stores the result in the first operand. The second operand can be a register, memory location, or immediate value.
- **Three-operand form**
  Multiplies the first and third operands and stores the result in the second operand. The destination must be a register.

```
382 ; One-operand format
383 imul eax
384 ; eax = eax * eax
385
386 ; Two-operand format
387 imul eax, ebx
388 ; eax = eax * ebx
389
390 ; Three-operand format
391 imul eax, ebx, ecx
392 ; eax = ebx * ecx
```

II. When to Use IMUL

Use **IMUL** whenever you are working with **signed numbers** and need the sign of the result to be correct. This is especially important when dealing with negative values.

III. Flags and Overflow Behavior

IMUL can set both the **Overflow (OF)** and **Carry (CF)** flags:

- **Overflow Flag (OF)** is set when the product is too large to fit in the destination operand.
- **Carry Flag (CF)** is set when significant bits are lost during the operation.

After an IMUL instruction, it's good practice to check these flags to verify whether the result is valid.

IV. Example Scenario (Overflow Case)

If two 32-bit values such as 80000000h and 80000000h are multiplied, the result is too large to fit in a 32-bit register.

Because of this, the **Overflow flag is set**, indicating that the result cannot be represented correctly in the destination register.

In this situation, the program must handle the overflow—either by reporting an error, adjusting the calculation, or using a larger data size. If no overflow occurs and the flags are clear, the program can safely continue using the result.

```
;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
```

This code:

- Demonstrates using **IMUL** with both **16-bit and 32-bit** operands.
- Shows multiplication using **register-to-register** operations and **immediate values**.
- Results are stored in **AX** for 16-bit operations and **EDX:EAX** for 32-bit operations.
- Includes examples where multiplication results are stored **directly in memory** (word and dword variables).
- Highlights that **IMUL can cause overflow** if the result is too large for the destination.
- Emphasizes the need to **check the Overflow flag** after IMUL to ensure the result is valid.

```
;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
```

This code:

- Moves **16-bit values** into AX and BX, then uses **IMUL** to multiply them and store the result in AX.
- Repeats the same process using **32-bit values**, with the result stored in **EDX:EAX**.
- Demonstrates multiplying a **32-bit value by immediate operands** (8-bit and 32-bit), storing results in **AX** or **EDX:EAX** depending on size.
- Shows **IMUL storing results in memory**, using word and dword variables.
- Notes that **IMUL can cause overflow** if the result exceeds the destination size.
- Stresses the importance of **checking the Overflow flag** after the operation to confirm the result is valid.

```
;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
```

This code:

- Two **32-bit integers** (80000000h × 80000000h) are multiplied using **IMUL**.
- The result (6400000000h) is **too large to fit in a 32-bit register**.
- Because of this, the **Overflow flag (OF) is set**, indicating the result is invalid.
- The **Carry flag (CF)** is checked next.
- CF is only set if the **highest bit of the result is 1**; in this case, it is **not set**.
- Since **overflow occurred**, the multiplication result cannot be trusted.
- The program must **handle the overflow**, such as reporting an error or adjusting the calculation.
- If no overflow had occurred, the Carry flag would then be checked to help determine whether the result is usable.
- In this example, even though CF is clear, **OF being set means the result is incorrect**.

These examples simply demonstrates how **IMUL** sets the **Overflow** and **Carry** flags in MASM; in practice, there are many different ways to handle these conditions.

IMUL in 64-Bit Mode

```
488 ;64-bit mode IMUL example
489 mov rax,-4
490 mov rbx,4
491 imul rbx    ;RDX = 0FFFFFFFFFFFFFFFFh, RAX = -16
```

- The **IMUL** instruction multiplies the 64-bit registers **RBX** and **RAX**.
- The result of the multiplication is **–64**, which is represented as a **128-bit value**.
- The product is stored across the **RDX:RAX** register pair.
- **RDX** contains the **upper 64 bits** of the result.
- **RAX** contains the **lower 64 bits** of the result.
- Because the result is negative, the upper 64 bits are **all 1s**, so **RDX = 0xFFFFFFFFFFFFFFFFh**.
- The lower 64 bits represent **–64**, so **RAX = 0xFFFFFFFFFFFFFFC0h**.

```
495 ;64-bit mode IMUL example
496 .data
497     multiplicand QWORD -16
498 .code
499     imul rax, multiplicand, 4 ;RAX = FFFFFFFFFFFFFFC0 (-64)
```

- **IMUL** multiplies a 64-bit **memory operand** by the immediate value 4.
- The result is **–64**, stored in the **64-bit RAX** register.
- The memory operand is a **QWORD variable** (64 bits) defined in the data section.
- The **immediate value 4** is 32-bit but is automatically **promoted to 64 bits** for the operation.
- After multiplication, **RAX = 0xFFFFFFFFFFFFFFC0h**, representing the product –64.

IMUL can perform **unsigned multiplication**, but it has a limitation.

Carry (CF) and **Overflow (OF)** flags do **not reliably indicate** whether the upper half of the product is zero for unsigned results.

This happens because **IMUL always sign-extends** the product, even with unsigned operands.

As a result, CF and OF may be set if the **high-order bit is 1**, even when the unsigned product is valid.

IMUL works for both **signed and unsigned multiplication** in 64-bit mode, but care is needed when interpreting the flags for unsigned operations.

MUL Examples in Depth

I. 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.

II. MUL Signed and Unsigned Examples

```
511 ;Multiply 48 by 4 and store the product in AX.
512 mov al, 48
513 mov bl, 4
514 mul bl
515 ;AX = 00C0h, OF = 1
516
517 ;Multiply -4 by 4 and store the product in AX.
518 mov al, -4
519 mov bl, 4
520 mul bl
521 ;AX = FFF0h, OF = 0
522
523 ;Multiply 48 by 4 and store the product in DX:AX.
524 mov ax, 48
525 mov bx, 4
526 mul bx
527 ;DX:AX = 000000C0h, OF = 0
528
529 ;Multiply 4,823,424 by -423 and store the product in EDX:EAX.
530 mov eax, 4823424
531 mov ebx, -423
532 mul ebx
533 ;EDX:EAX = FFFFFFFF86635D80h, OF = 0
```

III. Two-Operand IMUL Instructions

Two-operand IMUL multiplies two values and stores the result in the **destination operand**.

The destination operand is **the same size as the multiplier**.

Because of this, **signed overflow can occur**.

Always **check the Overflow (OF) flag** after executing a two-operand IMUL.

Code example:

```
;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!
```

IV. Three-Operand IMUL Instructions

Three-operand IMUL multiplies the **first operand** by the **second operand**.

The **product** is stored in the **third operand**.

The **destination operand** (third operand) is the **same size as the first operand**.

Code example:

```
;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!
```

MUL and **IMUL** can multiply signed and unsigned numbers, but they can overflow. Always check the **Overflow flag** to make sure the result is valid.

MEASURING EXECUTION TIMES

The code uses the **GetMseconds** procedure from the **Irvine32 library** to measure program execution time.

GetMseconds returns the number of **milliseconds since midnight**.

To measure execution time:

- Call **GetMseconds** to record the **start time**.
- Run the program whose execution time you want to measure.
- Call **GetMseconds** again to record the **end time**.

The **execution time** is calculated as **end time – start time**.

I. GetMseconds Code example:

```
585 .data
586     startTime DWORD ?
587     procTime DWORD ?
588
589 .code
590     call GetMseconds
591     ; get start time
592     mov startTime, eax
593
594     ; call the program whose execution time you wish to measure
595     ; ...
596
597     call GetMseconds
598     ; get end time
599     sub eax, startTime
600     ; calculate the elapsed time
601     mov procTime, eax
602     ; save the elapsed time
```

The variable **procTime** stores the program's **execution time in milliseconds**.

This method works for measuring **any program**, no matter how complex.

The **overhead** of calling **GetMseconds** twice is usually **negligible**.

Relative performance of two implementations can be measured using **GetMseconds**.

Measure the execution time of each implementation, then **divide the first by the second** to get a **relative performance ratio**.

II. Example: Comparing the performance of two 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
```

EAX stores the **relative performance** of two sorting algorithms.

Interpretation of values:

- 1.0 → both algorithms perform the same.
- >1.0 → first algorithm is faster.
- <1.0 → first algorithm is slower.

This technique works for **any two code implementations**, regardless of complexity.

III. Performance comparison of multiplication methods:

- Older x86 CPUs were **slower with MUL/IMUL** compared to bit-shifting for powers-of-two multiplication.
- Recent Intel processors have **optimized MUL/IMUL**, making them as fast as bit-shifting.

Example: multiplying a number by **36** using **bit shifting** vs. **MUL**.

```
661 ;Multiplies EAX by 36 using SHL, LOOP_COUNT times.
662 mult_by_shifting PROC
663 mov ecx, LOOP_COUNT
664 L1: push eax
665 ; save original EAX
666 mov ebx, eax
667 shl eax, 5
668 shl ebx, 2
669 add eax, ebx
670 pop eax
671 ; restore EAX
672 loop L1
673 ret
674 mult_by_shifting ENDP
675
676 ; Multiplies EAX by 36 using MUL, LOOP_COUNT times.
677 mult_by_MUL PROC
678 mov ecx, LOOP_COUNT
679 L1:
680 push eax
681 ; save original EAX
682 mov ebx, 36
683 mul ebx
684 pop eax
685 ; restore EAX
686 loop L1
687 ret
688 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 demonstrates measuring **program execution time** using **GetMseconds** from the Irvine32 library.

The program multiplies **5 × 36** using the mult_by_shifting procedure and displays the execution time.

Two **.data segments** define:

- LOOP_COUNT → number of times to repeat the multiplication.
- intval → the integer to multiply by 36.

Having **two .data segments** is not strictly necessary; both variables could be defined in the **same .data segment**.

Possible reasons for two segments: logical organization or potential Irvine32 optimizations.

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
```

The **revised program** is more concise, easier to read, and works the same as the original.

Guidelines for segments:

- You can have multiple segments for **.data, .code, .bss, .text**.
- **Group related data and code** for clarity.
- Avoid excessive segments to improve **readability and performance**.

Performance comparison (legacy Pentium 4, 4 GHz):

- mult_by_shifting → **6.078 seconds**
- mult_by_MUL → **20.718 seconds** (≈241% slower)

On **modern processors**, both procedures execute **at the same speed**.

Intel has **optimized MUL/IMUL**, removing the need for bit-shifting for powers-of-two multiplication.

Using **MUL/IMUL** is now **preferred** for readability and maintainability.

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.

The **divisor** can be a **64-bit register** or **memory operand**.

After division:

- **Quotient → RAX**
- **Remainder → RDX**

Definitions:

- **Dividend** → number being divided
- **Divisor** → number dividing the dividend
- **Quotient** → result of the division
- **Remainder** → leftover after division

The **operand size** of the dividend and divisor determines the size of the **quotient and remainder** e.g. 8-bit dividend/divisor → 8-bit quotient and remainder.

Both dividend and divisor can be stored in **registers or memory** e.g. Dividend in **AX**, divisor in **BL**.

Here's us using DIV instruction to perform 8-bit unsigned division

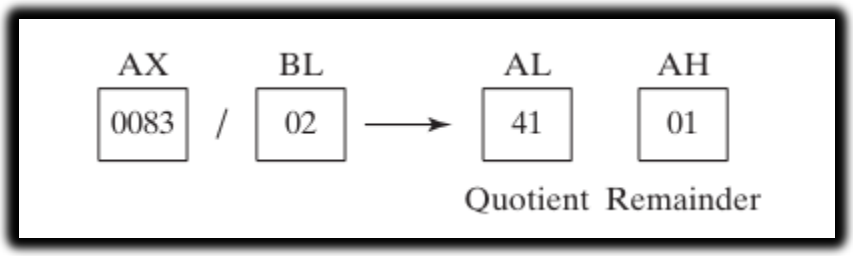
The following instructions perform 8-bit unsigned division (83h/2), producing a quotient of 41h and a remainder of 1:

```
746 ;dividend
747 mov ax, 0083h
748 ;divisor
749 mov bl, 2
750 ;divide
751 div bl
752 ;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:

```
769 .data
770     dividend QWORD 0000000800300020h
771     divisor DWORD 00000100h
772 .code
773     mov edx, DWORD PTR dividend + 4     ;high doubleword
774     mov eax, DWORD PTR dividend         ;low doubleword
775     div divisor
776     ;EAX = 08003000h, EDX = 00000020h
```

.data section defines two variables:

- dividend → 64-bit integer (QWORD)
- divisor → 32-bit integer (DWORD)

.code section marks the start of the program's instructions.

Loading the dividend:

- mov edx, DWORD PTR dividend + 4 → loads **high 32 bits** into EDX
- mov eax, DWORD PTR dividend → loads **low 32 bits** into EAX

DIV instruction:

- Divides the **64-bit dividend in EDX:EAX** by the 32-bit divisor.
- Stores **quotient in EAX** and **remainder in EDX**.

Example result:

- Dividend = 0000000800300020h, Divisor = 00000100h
- Quotient → EAX = 08003000h
- Remainder → EDX = 00000020h

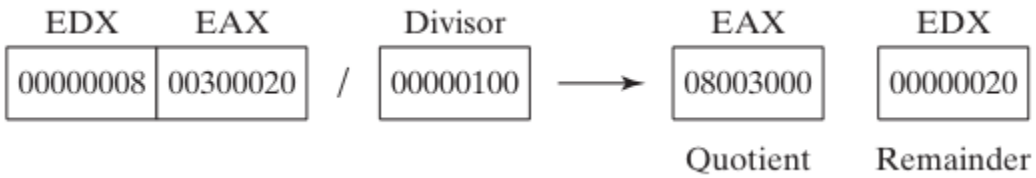
EAX:EDX usage:

- EAX → lower 32 bits of 64-bit value
- EDX → upper 32 bits of 64-bit value

Analogy: EAX:EDX acts as a **64-bit register pair**, storing **quotient and remainder** after division.

Simple example: 100 ÷ 10 → EAX = 10 (quotient), EDX = 0 (remainder).

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:

```
787 .data
788     dividend_hi QWORD 0000000000000108h
789     dividend_lo QWORD 0000000033300020h
790     divisor QWORD 0000000000010000h
791 .code
792     mov rdx, dividend_hi
793     mov rax, dividend_lo
794     div divisor ;RAX = 0108000000003330
795     ;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:

```
840 IDIV reg/mem8
841 IDIV reg/mem16
842 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:

```
847 .data
848     byteVal SBYTE -48
849     ;D0 hexadecimal
850 .code
851     mov al,byteVal
852     ;lower half of dividend
853     cbw
854     ;extend AL into AH
855     mov bl,+5
856     ;divisor
857     idiv bl
858     ;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:

```
878 .data
879     dwordVal SDWORD + 50000
880 .code
881     mov eax,dwordVal
882     ;dividend, low
883     cdq
884     ;extend EAX into EDX
885     mov ebx,-256
886     ;divisor
887     idiv ebx
888     ;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:

```
892 mov ax,1000h
893 mov bl,10h
894 div bl
895 ; 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:

```
901 mov eax,1000h
902 cdq
903 mov ebx,10h
904 div ebx
905 ; 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:

```
909 mov eax, dividend
910 mov bl, divisor
911 cmp bl, 0
912 je NoDivideZero
913
914 ; Not zero: continue
915 div bl
916
917 ; ...
918
919 NoDivideZero:
920 ; 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:

```
928 mov eax, var1
929 add eax, var2
930 mul var3
931 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:

```
945 mov eax, var1
946 add eax, var2
947 mul var3
948 jc tooBig        ;jump if overflow
949 mov var4, eax
950 jmp next
951 tooBig:
952     ;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:

```
960 mov eax, var2
961 neg eax
962 cdq             ;sign-extend dividend
963 idiv var3
964 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:

```
970 movsx  eax, word ptr [your_16_bit_variable] ; Sign extend 16-bit value to 32 bits
971 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.

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

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

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

Here is another example of using the ADC instruction:

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

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

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

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

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

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

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

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

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

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

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

Loading the First Operand (operand1):

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

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

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

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

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

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

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

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

Adding the Two Operands with Carry Propagation:

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

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

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

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

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

Storing the Result in the result Array:

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

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

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

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

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

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

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

==========

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

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

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

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

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

Here is a more detailed explanation of the code:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    ; Call the Extended_Add function
    call Extended_Add

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

    ; Call a function to output a newline.
    call Crlf

    ; Exit the program
    invoke ExitProcess, 0

main ENDP
```

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

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

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

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

Next, the main procedure calls the Extended_Add procedure.

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

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

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

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

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

The output of the program is the following:

```
1087  0122C32B0674BB5736
```

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

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

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

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

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

Here is a more detailed explanation of the Display_Sum procedure:

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

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

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

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

The L1: label marks the beginning of the loop.

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

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

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

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

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

****

Subtract with Borrow

==========

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

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

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

```
1111 mov edx, 7
1112 ; upper half
1113 mov eax, 1
1114 ; lower half
1115 sub eax, 2
1116 ; subtract 2
1117 sbb edx, 0
1118 ; 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:

```
1123 mov edx, 7
1124 ; 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.

```
1128 mov eax, 1
1129 ; 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.

```
01 sub eax, 2
02 ; 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.

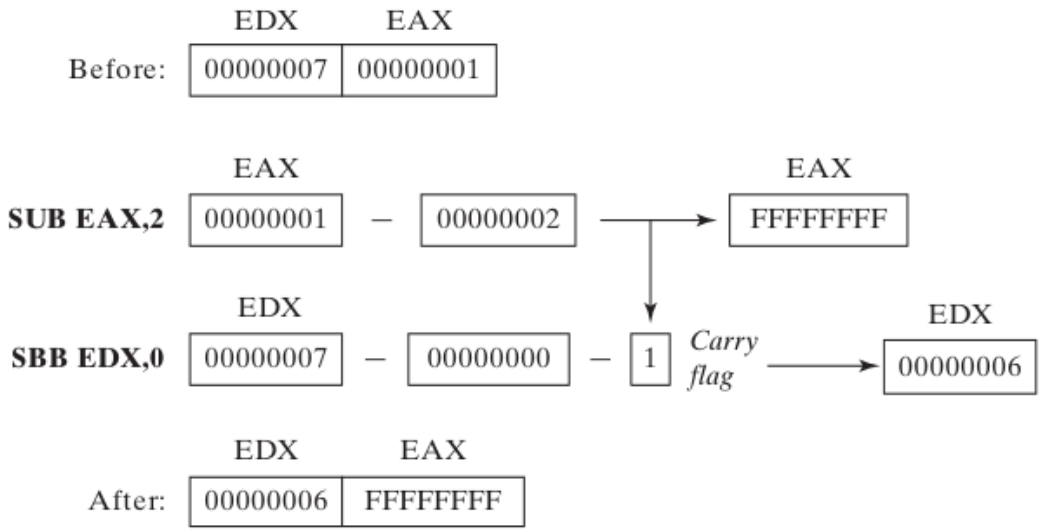
```
sbb edx, 0
; subtract upper half
```

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.

(All values are in hexadecimal)

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:

```
36 mov esi, SIZEOF decimal_one - 1
37 mov edi, SIZEOF decimal_one
38 mov ecx, SIZEOF decimal_one
39 mov bh, 0
40
41 L1:
42 mov ah, 0
43 mov al, decimal_one[esi]
44 add al, bh
45 aaa
46 mov bh, ah
47 or bh, 30h
48 add al, decimal_two[esi]
49 aaa
50 or bh, ah
51 or bh, 30h
52 or al, 30h
53 mov sum[edi], al
54 dec esi
55 dec edi
56 loop L1
57
58 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:**

```
63 mov
64 esi,OFFSET decimal_one
65 mov
66 edi,OFFSET sum
67 mov
68 ecx,SIZEOF decimal_one
69
70 call
71 ASCII_add
72
73 mov
74 edx,OFFSET sum
75 call
76 WriteString
77 call
78 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:

```
83 mov ah, 0 ; clear AH before subtraction
84 mov al, 8
85 sub al, 9 ; subtract 9 from 8
86 aas ; adjust the result
87 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:

```
095 mov esi, offset first_number
096 mov edi, offset second_number
097 mov ecx, length_of_first_number
098
099 loop:
100 mov ah, 0 ; clear AH before subtraction
101 mov al, [esi]
102 sub al, [edi]
103 aas ; adjust the result
104 or al, 30h ; convert AL to ASCII
105 mov [esi], al
106
107 inc esi ; increment the first number pointer
108 inc edi ; increment the second number pointer
109 dec ecx ; decrement the loop counter
110
111 cmp ecx, 0
112 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:

```
149 mov ax, 0307h ; dividend
150 aad ; AX = 0025h
151 mov bl, 5 ; divisor
152 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:

```
186 sub al, 30h   ; Convert the tens digit to binary
187 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:

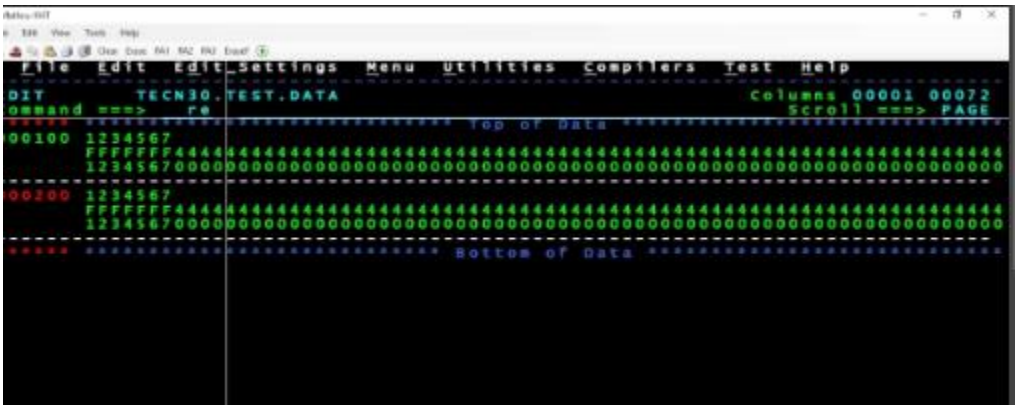
```
193 bcd1 QWORD 2345673928737285h    ; 2,345,673,928,737,285 decimal
194
195 bcd2 DWORD 12345678h             ; 12,345,678 decimal
196
197 bcd3 DWORD 08723654h             ; 8,723,654 decimal
198
199 bcd4 WORD 9345h                  ; 9,345 decimal
200
201 bcd5 WORD 0237h                  ; 237 decimal
202
203 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.



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:

```
208 mov al, 35h
209 add al, 48h ; AL = 7Dh
210 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:

```
256 mov bl, 48h
257 mov al, 85h
258 sub al, bl ; AL = 3Dh
259 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.