Contents

[SHIFTS AND ROTATES 1](#_Toc220176970)

[SHIFT LEFT AND SHIFT RIGHT 3](#_Toc220176971)

[ARITHMETIC SHIFT RIGHT AND ARITHMETIC SHIFT LEFT 12](#_Toc220176972)

[ROTATE LEFT AND ROTATE RIGHT 14](#_Toc220176973)

[SHIFT LEFT DOUBLE AND SHIFT RIGHT DOUBLE 20](#_Toc220176974)

[SHIFTING MULTIPLE DOUBLEWORDS 27](#_Toc220176975)

[BINARY SHIFTING AND MULTIPLICATION 30](#_Toc220176976)

[BINTOASC 35](#_Toc220176977)

[EXTRACTING FILE DATE FIELDS 36](#_Toc220176978)

[MUL OPERATOR 40](#_Toc220176979)

[IMUL OPERATOR 45](#_Toc220176980)

[MEASURING EXECUTION TIMES 54](#_Toc220176981)

[DIV INSTRUCTION 60](#_Toc220176982)

[SIGNED DIV INSTRUCTION 65](#_Toc220176983)

[IDIV INSTRUCTION 68](#_Toc220176984)

[EXTENDED ADDITION AND SUBTRACTION 75](#_Toc220176985)

[ASCII AND UNPACKED DECIMAL ARITHMETIC 85](#_Toc220176986)

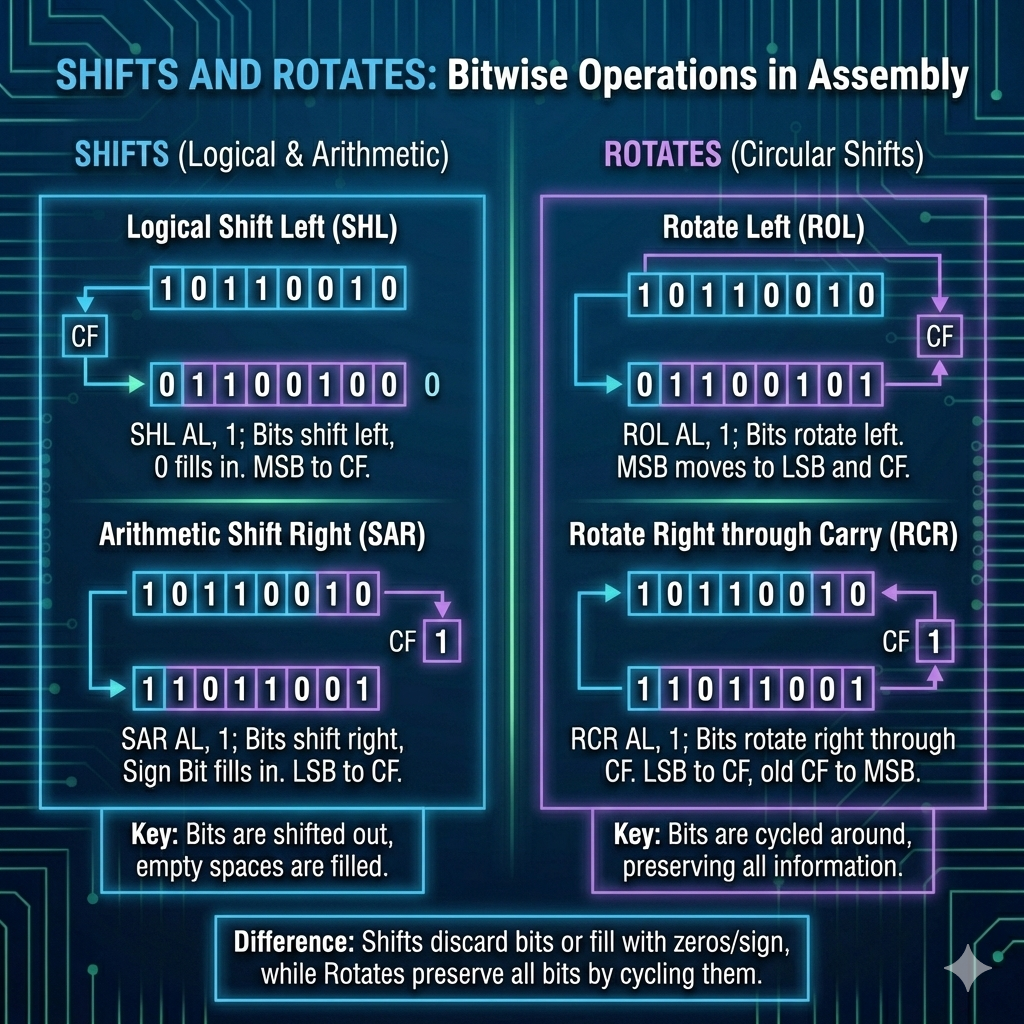
[AAA (ASCII ADJUST AFTER ADDITION) 89](#_Toc220176987)

[AAS, AAM and AAD 91](#_Toc220176988)

[PACKED DECIMAL ARITHMETIC, DAA and DAS 96](#_Toc220176989)

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 2n, and right shifting is like dividing by 2n.



**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!

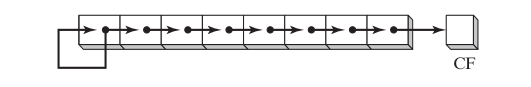
Assembly also lets you do arithmetic on integers of any size, which can be a huge advantage over high-level languages when you're working with large numbers or custom operations.

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

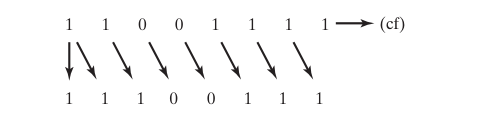
SAL/SAR (Arithmetic Shift Left/Arithmetic Shift Right):

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

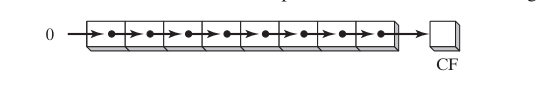
Bit shifting involves moving bits within an operand either to the left or right.

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.



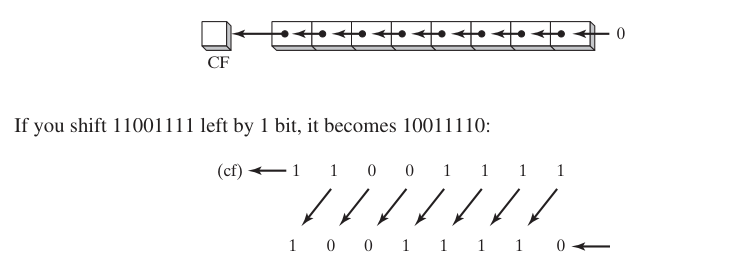
The following illustration shows a **single logical right shift** on the binary value 11001111, producing 01100111.

The lowest bit is **shifted into** the Carry flag:

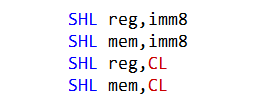


The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.

The highest bit is moved to the Carry flag, and the bit that was in the Carry flag is discarded:







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:



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:



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:



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:

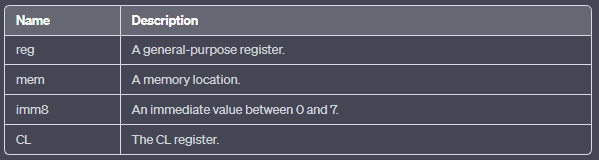


The SHL instruction shifts the bits of the destination operand to the left by the specified number of bits.

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:



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 2² = 4, giving 40.

Shifting left by one position is the same as multiplying by 2, shifting by three positions multiplies by 2³ = 8, and so on.

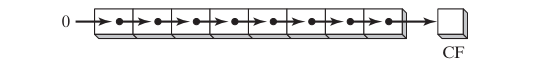
After a single SHL instruction on decimal 10, the BL register will contain 20.

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



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.



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



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:

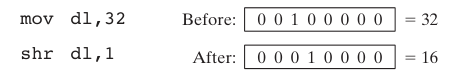
Bitwise division by shifting right is a **fast shortcut** for dividing by powers of 2. It doesn’t work for arbitrary divisors like 21 — only for divisors that are exact powers of 2.

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

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**.



In the following example, 64 is divided by 23:



Write the dividend and divisor in binary form:



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



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



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:



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**.

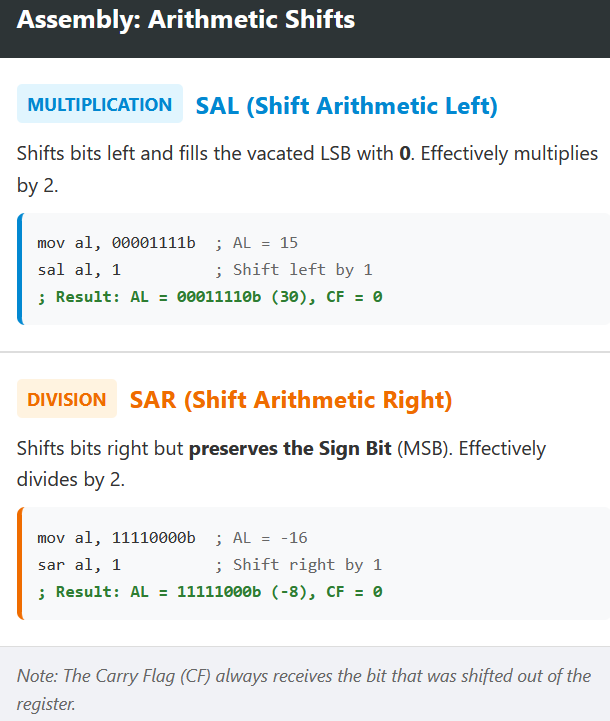
**Effect:** Ensures signed numbers remain correctly signed after shifting.

**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

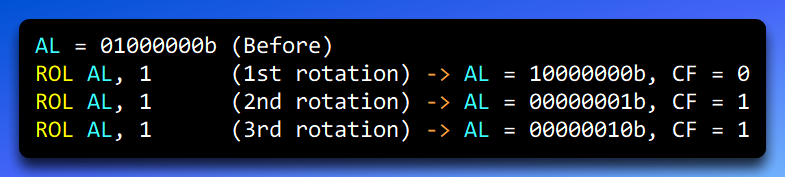
The **ROL** instruction performs a circular bit rotation to the left.

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.

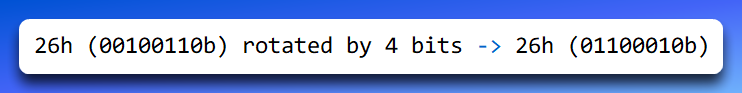
**Example**:

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



**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:



This is useful for data reordering or manipulation tasks.

II. ROR (Rotate Right) Instruction

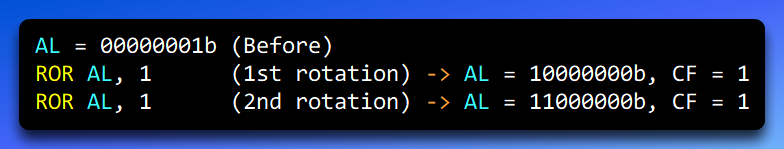
The **ROR** instruction is the reverse of **ROL**—it performs a circular bit rotation to the right.

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:



* **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.

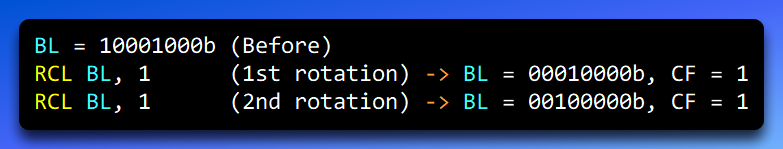
III. RCL (Rotate Carry Left) Instruction

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:



**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.

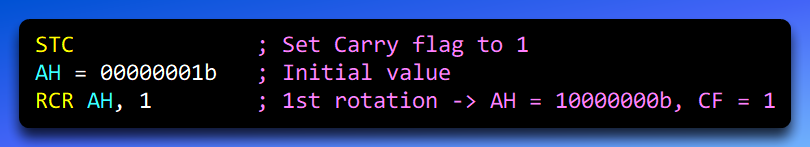
IV. RCR (Rotate Carry Right) Instruction

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:

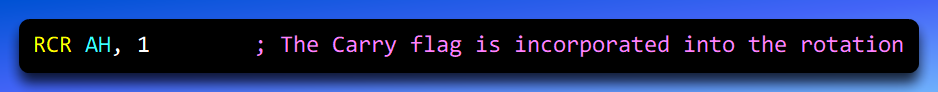


**Use Case**: This instruction is useful when working with bit-level encryption and precise bit manipulation where you need to handle the Carry flag.

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.

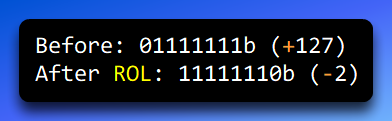
**Example**: If the **Carry flag** is set before performing a **RCR** operation, it influences the resulting bits in the operand:



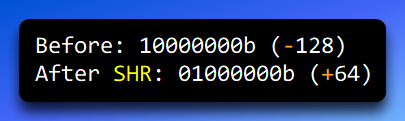
VI. Signed Overflow

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:

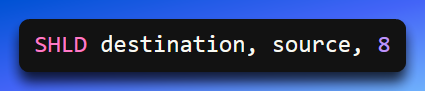
* **Destination operand**: This is the value you want to shift.
* **Source operand**: This is another value that provides the bits to fill the empty spaces when the destination operand is shifted.

**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:



* 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:



**where:**

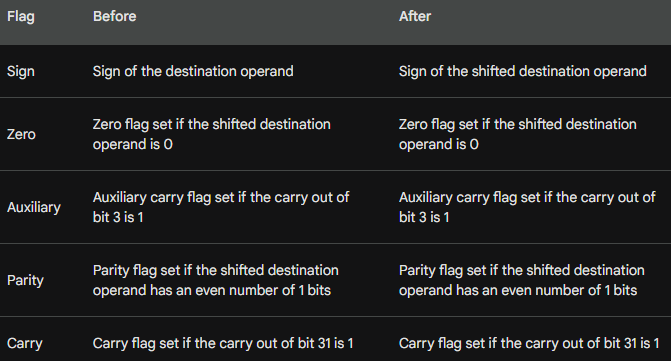
* dest is the destination operand.
* source is the source operand.
* count is the number of bits to shift.

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

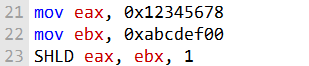
If count is 0, the destination operand is not shifted.

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:



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



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

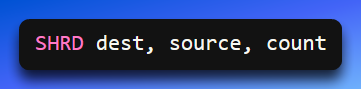
* **SHLD** shifts the **destination** operand to the left.
* The empty bits are filled with bits from the **source operand** (not zeros).
* The **source operand** itself is **not affected** by this operation.

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:**



**Where:**

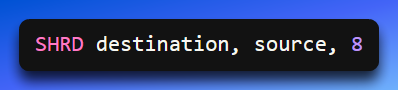
* **dest** is the destination operand.
* **source** is the source operand.
* **count** is how many bits you want to shift.

**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:



* 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:

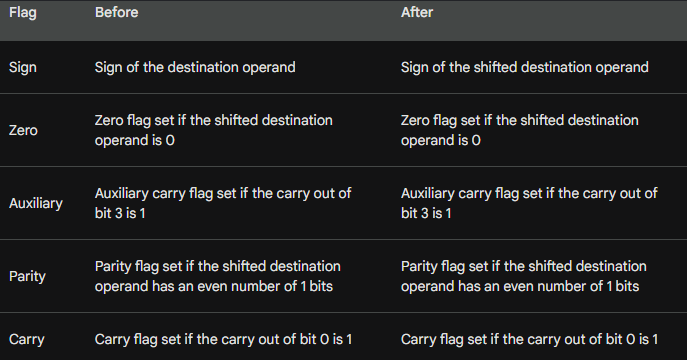
* **Count = 0**: If you set **count** to 0, the **destination operand** won’t shift at all.
* **Count = 31**: If **count** is 31, the **destination operand** will be shifted all the way to the right, and the entire **source operand** will be copied into the **destination operand**.

Key Points:

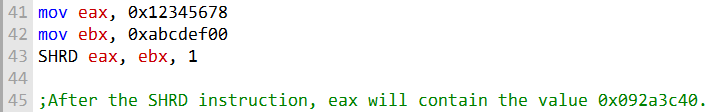
* **SHRD** performs a right shift on the **destination operand**.
* It uses bits from the **source operand** to fill in the empty positions.
* The **source operand** itself is not changed.
* **Count** defines how many positions the shift will happen. It must be between 0 and 31.

Status Flags After SHRD Instruction

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



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



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:

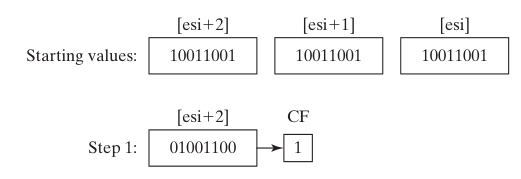
Step 1:

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.

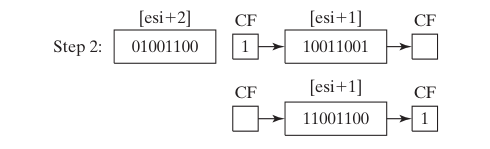


This instruction effectively shifts the byte at [ESI+2] one bit to the right, with the least significant bit being transferred to the Carry flag.



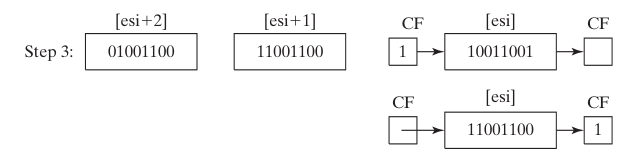
Step 2:

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:

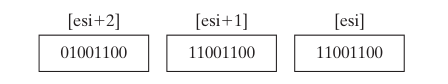


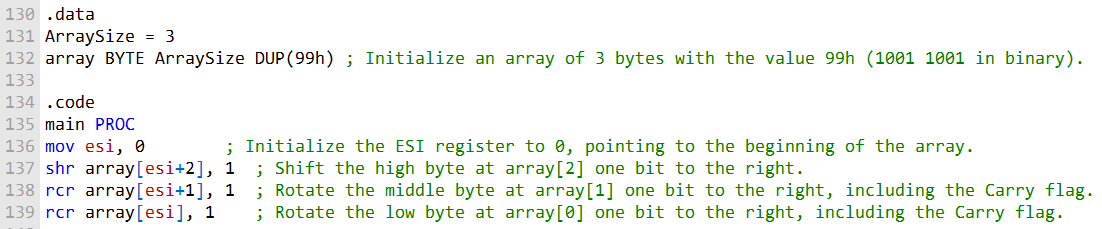
Step 3:

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:





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.

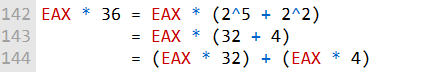
Shift by *n* bits = multiply by 2n.

Example:

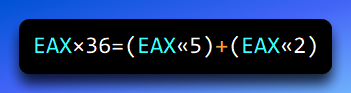
* EAX = 5
* SHL EAX, 1 → result = 10 (5 × 2).
* SHL EAX, 3 → result = 40 (5 × 8).

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.

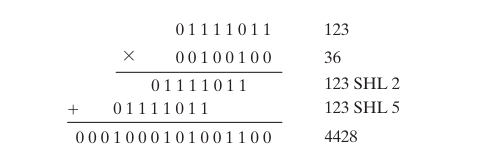


So,



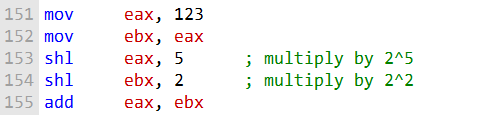
III. Example: 123 × 36 **🧩**

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.



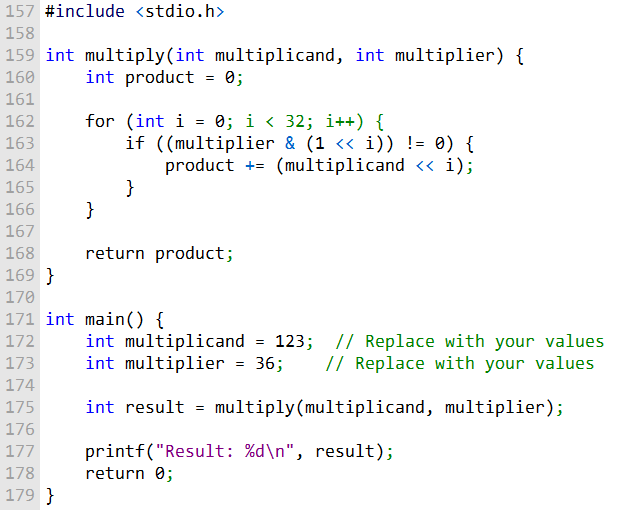
IV. General Algorithm (Unsigned 32‑bit Multiply)

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.



👉 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:



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.

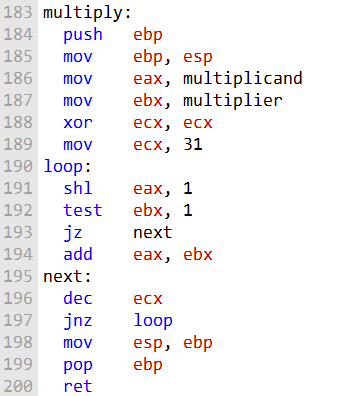
Then, we use a loop that runs 32 times, one for each bit of the **multiplier**. For each iteration, we check if the current bit of the multiplier is set to 1.

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:



This little routine is basically a DIY multiplication function.

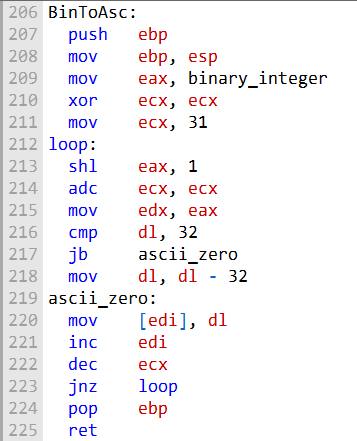
Instead of using the MUL instruction, it builds the product by shifting and adding, bit by bit.

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



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

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

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

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

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

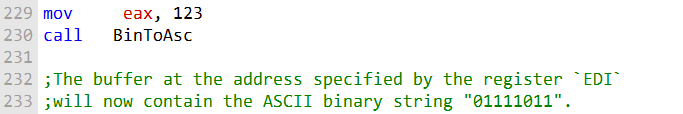
In this case, the procedure will move the ASCII character '1' (0x31) to the buffer at the address specified by the register EDI.

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

***Example 2 Usage***

:

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



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

EXTRACTING FILE DATE FIELDS

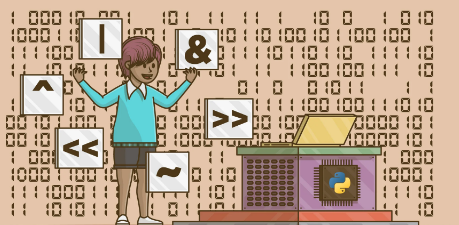
**Shifting and masking:** The two most important operations used to extract bit strings are shifting and masking.



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



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

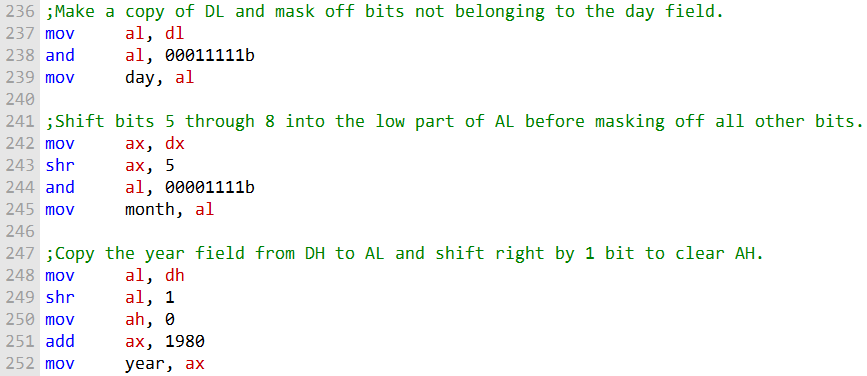


This can be useful for extracting bit strings that are spread across two bytes, such as the month and day fields of a date stamp.

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

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

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



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

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

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

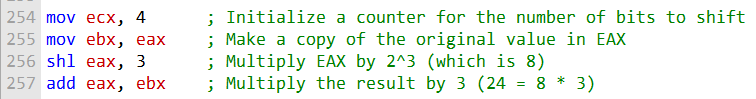
Finally, it adds 1980 to the value to account for the fact that the year field is relative to 1980. The code snippet then copies the final value to the year variable.

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

----------------------------------

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

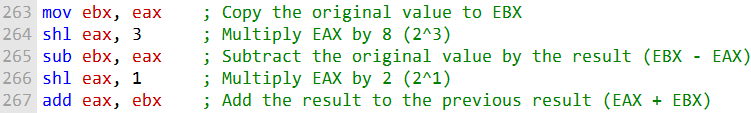
Here's how you can calculate **EAX \* 24** in assembly language using binary multiplication:



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

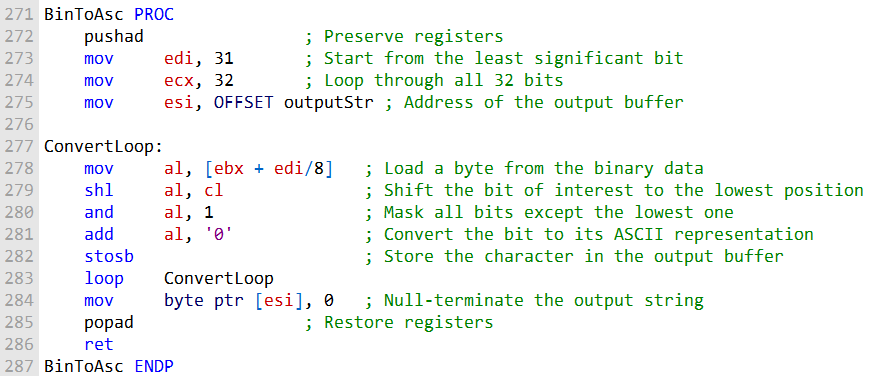


To calculate EAX \* 21, you can use binary multiplication based on the hint provided:



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

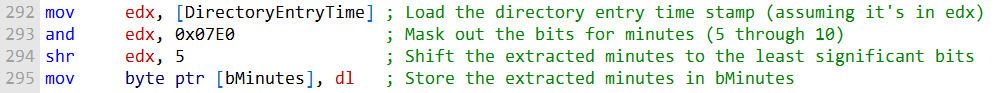
To display the binary bits in reverse order in the BinToAsc procedure, you can modify the loop that processes the bits. Instead of starting from the most significant bit (bit 31) and moving towards the least significant bit (bit 0), you can reverse the loop to start from the least significant bit and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



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

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

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



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

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

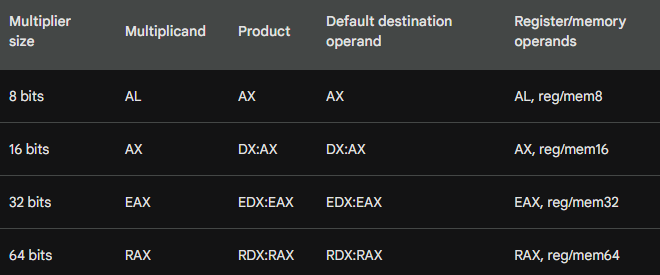
MUL OPERATOR

The MUL instruction performs unsigned integer multiplication.

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

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

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



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

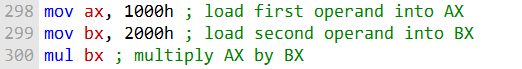
However, the MUL instruction sets the Carry and Overflow flags if the upper half of the product is not equal to zero.

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

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

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

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



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

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

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



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



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

-------------------------------------

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

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

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

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

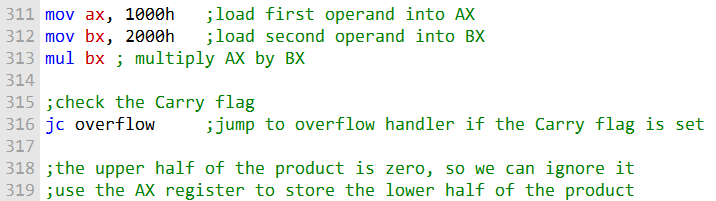
If the upper 16 bits of the product are zero, then you can safely ignore them. However, if the upper 16 bits of the product are non-zero, then you will need to use the DX register to store the entire product.

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

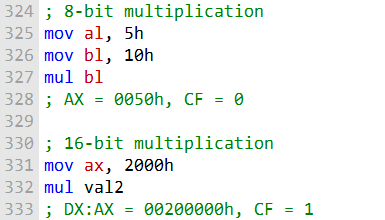
If the Carry flag is clear, then the upper half of the product is zero and you can safely ignore it.

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

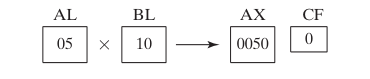
Here is an example of how to check the Carry flag after executing MUL:



***Example 1:***

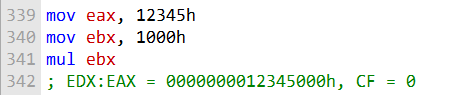


For this code, this is the data flow:





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

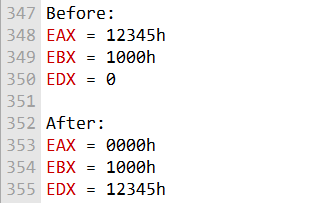


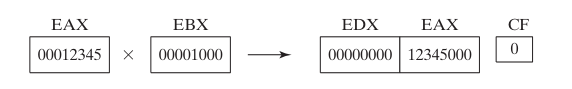
The MUL instruction multiplies two unsigned integers and stores the product in two registers: the low-order half of the product is stored in the EAX register, and the high-order half of the product is stored in the EDX register.

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

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

The following diagram illustrates the movement between registers:





***Here is a summary of the MUL instruction:***

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

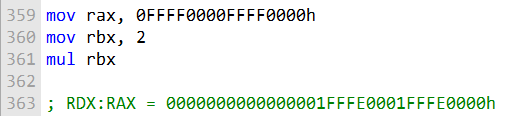
***====================***

***MUL in 64-bit mode***

***=====================***

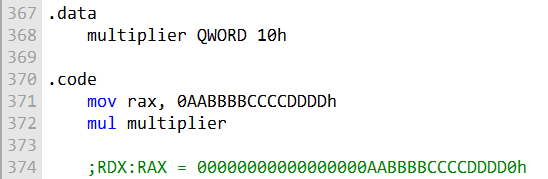
In 64-bit mode, the MUL instruction can be used to multiply two 64-bit operands. The result is a 128-bit product, which is stored in the RDX:RAX register pair.

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



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:



In this example, the product is a **128-bit value**, but both halves of the product fit in RAX and RDX because the product is less than **2128.**

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

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

IMUL OPERATOR

The IMUL instruction performs signed integer multiplication.

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

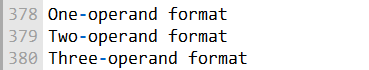
How does the IMUL instruction work?

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

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

What are the different formats of the IMUL instruction?

The IMUL instruction has three formats:



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

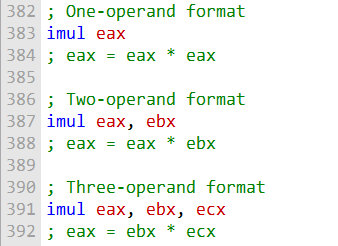
The **two-operand format of the IMUL instruction** multiplies the two operands and stores the product in the first operand. The second operand can be a register, a memory operand, or an immediate value.

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

***When should I use the IMUL instruction?***

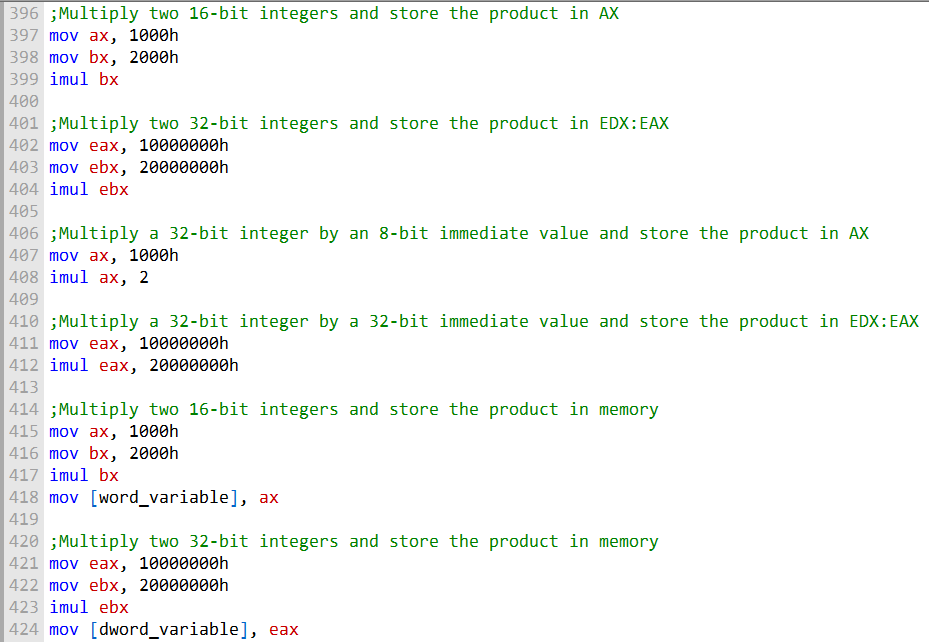
You should use the IMUL instruction whenever you need to perform signed integer multiplication. This is especially important when you need to preserve the sign of the product.

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



***Important things to keep in mind:***

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



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

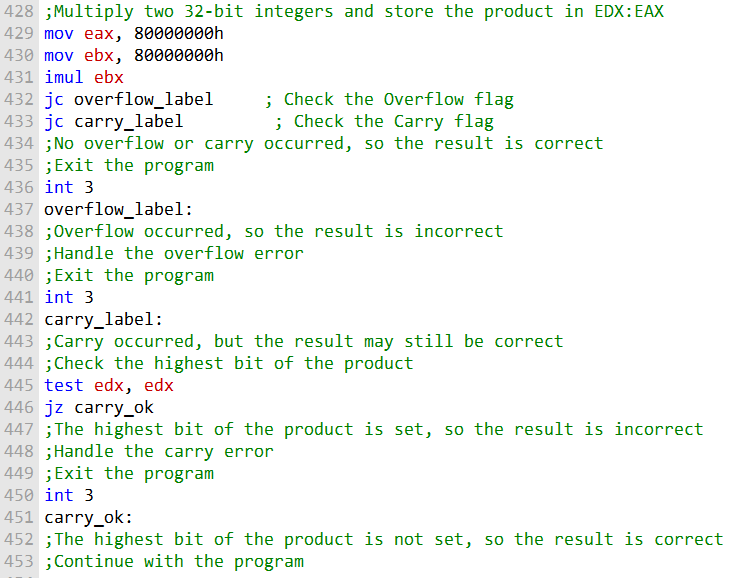
The next two lines of code do the same thing, but with 32-bit integers. The IMUL instruction in this case stores the product in the EDX:EAX register pair.

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

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

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

----------------------------------------------------



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

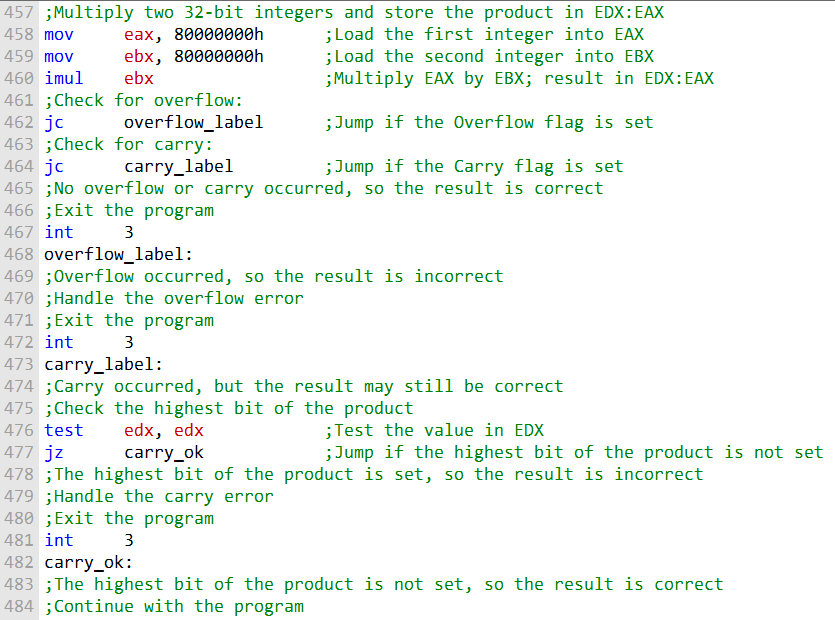
The next two lines of code do the same thing, but with 32-bit integers. The IMUL instruction in this case stores the product in the EDX:EAX register pair.

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

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

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

-------------------------------------------------



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

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

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

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

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

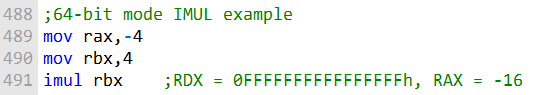
In this example, the highest bit of the product is not set, so the result is correct. We can then continue with the program.

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

***=====================***

***IMUL in 64-Bit Mode***

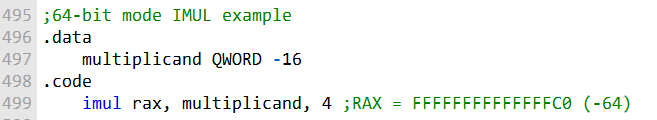
***=====================***



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

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

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



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

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

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

***Unsigned multiplication***

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

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

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

***MUL Examples in Depth***

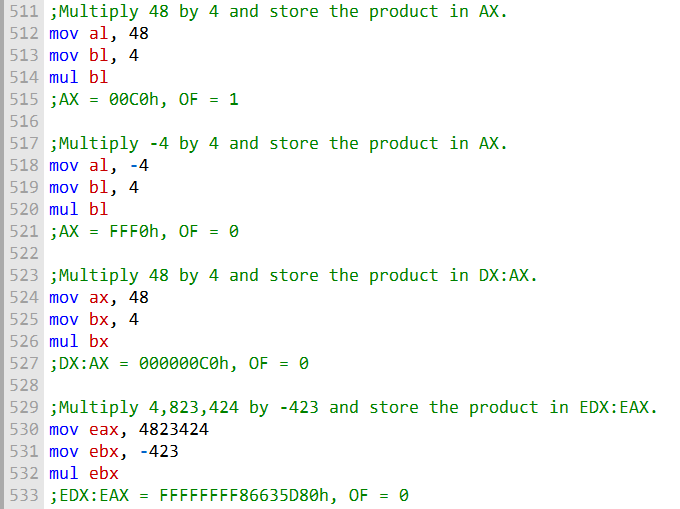
***MUL Overflow***

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



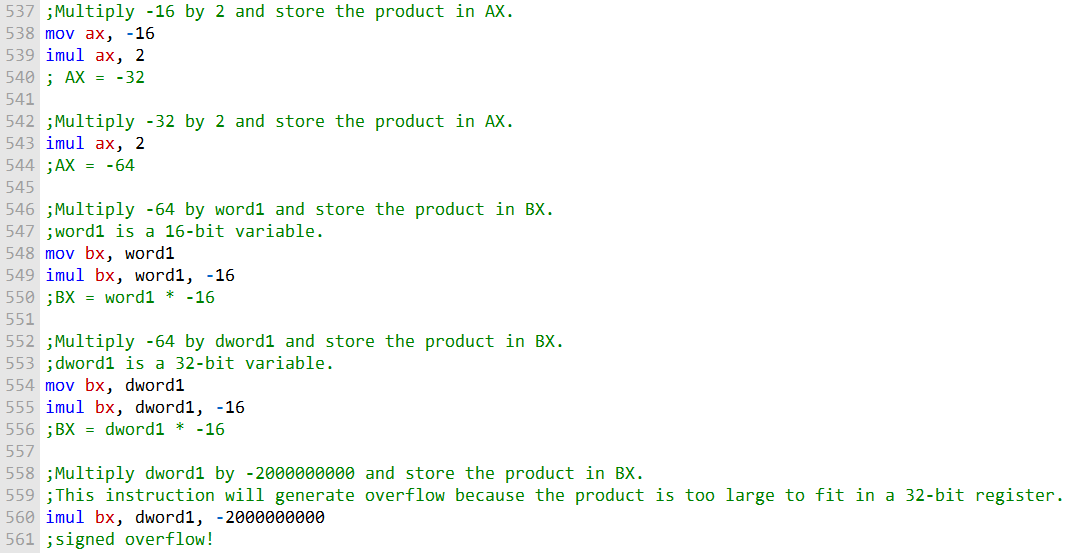
The product of -32000 and 2 is -64000, which is too large to fit in a 16-bit register. Therefore, the Overflow flag will be set after the IMUL instruction is executed.

***MUL Signed and Unsigned Examples***



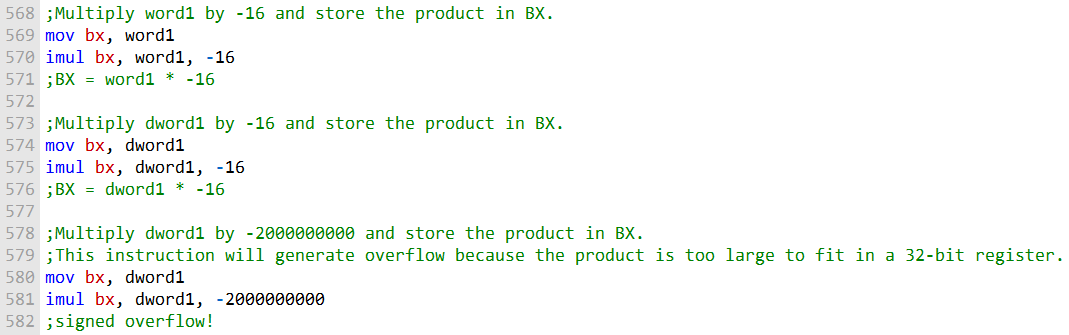
***Two-Operand IMUL Instructions***

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



***Three-Operand IMUL Instructions***

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



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

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

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

MEASURING EXECUTION TIMES

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

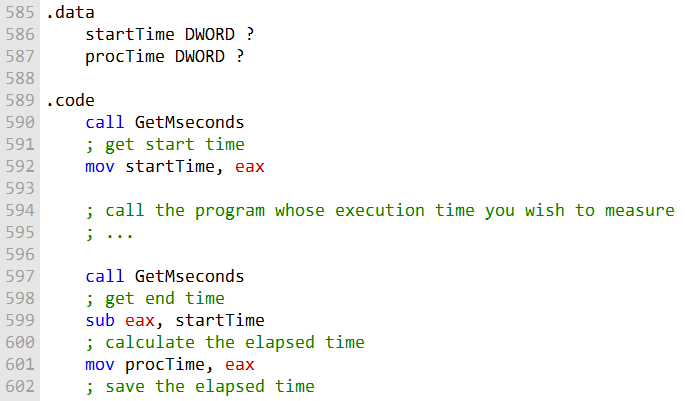
The **GetMseconds procedure** returns the number of system milliseconds that have elapsed since midnight.

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

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

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

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



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

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

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

------------------------------------

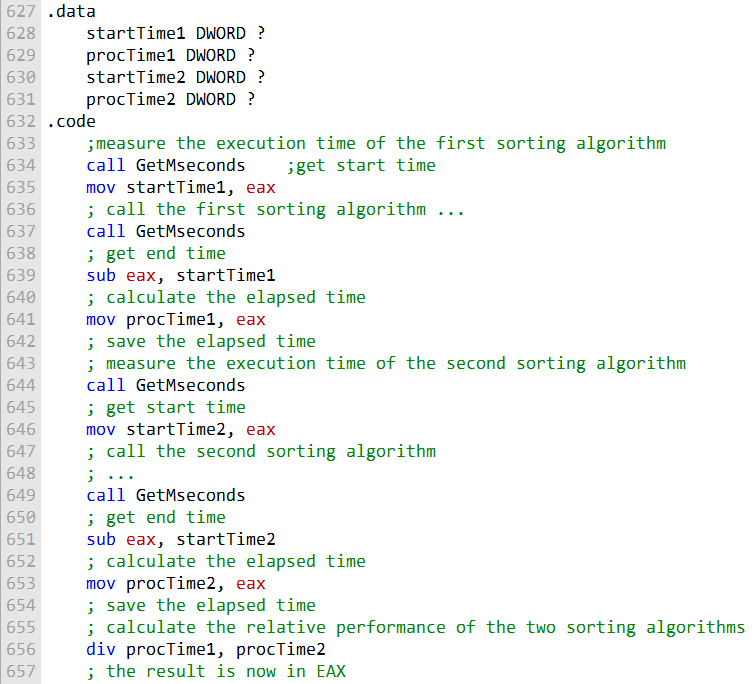
***Relative Performance***

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

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

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

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



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

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

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

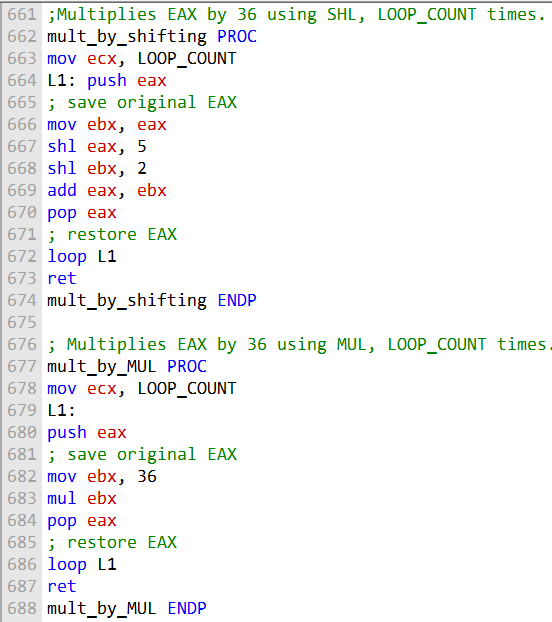
-----------------------------------------

***Comparing MUL and IMUL to Bit Shifting in Depth***

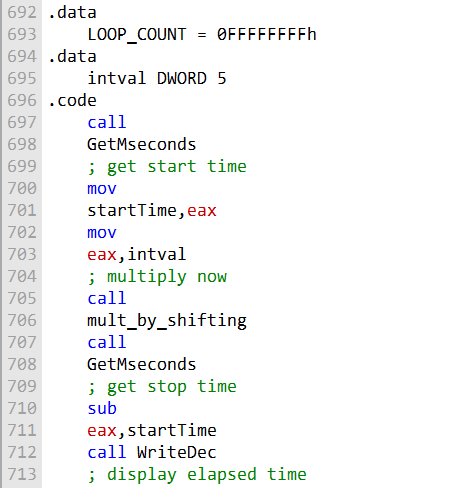
In older x86 processors, there was a significant difference in performance between multiplication by bit shifting and multiplication using the MUL and IMUL instructions.

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

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



The following code calls the mult\_by\_shifting procedure and displays the timing results:



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

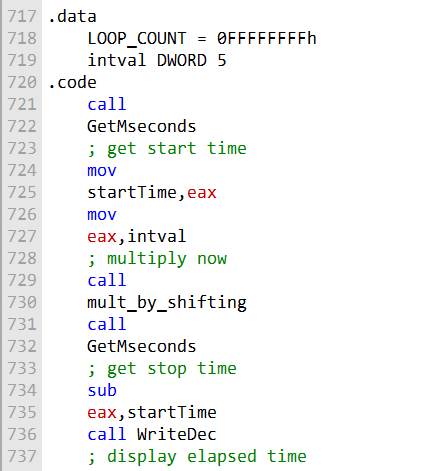
The two .data segments in the program are used to define two variables: LOOP\_COUNT and intval. LOOP\_COUNT is a constant that specifies the number of times to repeat the multiplication operation. intval is the integer that is multiplied by 36.

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

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

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

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



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

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

-------------------------------------------------

On a legacy 4-GHz Pentium 4 processor, the mult\_by\_shifting procedure executed in 6.078 seconds, while the mult\_by\_MUL procedure executed in 20.718 seconds.

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

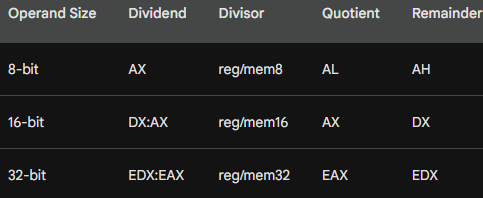
This example shows that Intel has managed to greatly optimize the MUL and IMUL instructions in recent processors.

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

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

DIV INSTRUCTION

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



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

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

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

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

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

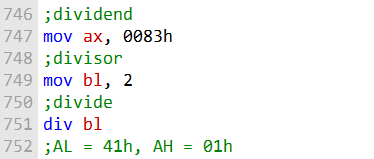
The table also shows that the dividend and divisor can be stored in registers or memory.

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

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

***DIV Examples***

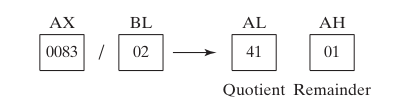
The following instructions perform 8-bit unsigned division (83h/2), producing a quotient of 41h and a remainder of 1:



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

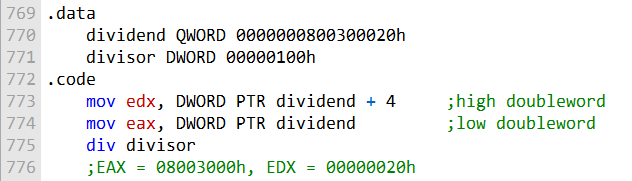
The DIV instruction divides the dividend by the divisor and stores the quotient in the AL register and the remainder in the AH register.

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:



Explanation:

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

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

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

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

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

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

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



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

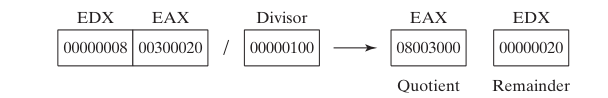
The **EAX:EDX registers** are a pair of 32-bit registers that can be used to store a 64-bit value. The EAX register stores the lower 32 bits of the value, and the EDX register stores the higher 32 bits of the value.

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

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

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

The following diagram illustrates the movement between registers:



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

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

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

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

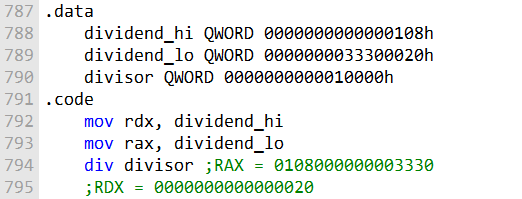
The DIV instruction divides the dividend in the **EAX:EDX registers** by the divisor in the divisor variable. The quotient is stored in the EAX register and the remainder is stored in the EDX register.

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

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

***DIV Example 3:***

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



***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

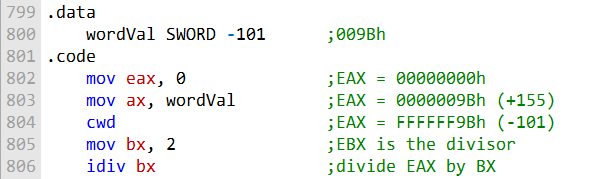
Signed integer division in MASM is similar to unsigned integer division, with one key difference: the dividend must be sign-extended before the division takes place. This is because the IDIV instruction (signed integer division) treats the dividend as a signed integer, and the result of the division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

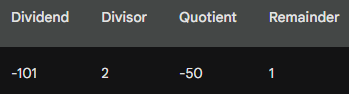


In this code, the CWD instruction is used to sign-extend the AX register into the EAX register.

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:



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)***

***================================================***

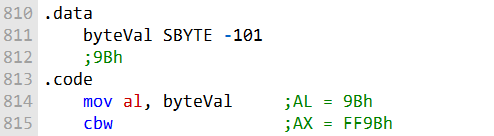
The CBW, CWD, and CDQ instructions are sign extension instructions that are used to extend the sign bit of a smaller integer to a larger integer.

***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:



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.

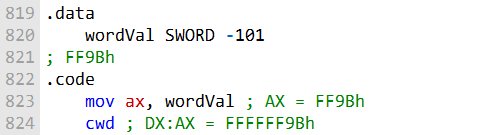
-----------------------------------------

***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:



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:



-----------------------------------------

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***

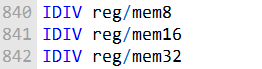
Sign extension instructions are a powerful tool that can be used to ensure that signed integers are handled correctly. By understanding how to use these instructions, you can write more efficient and reliable code.

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:***



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.

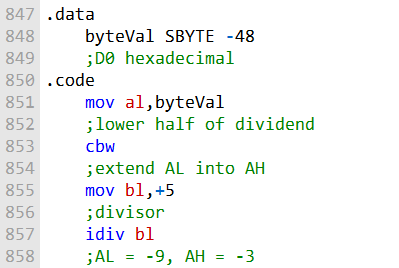
***Operation:***

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:



***Explanation:***

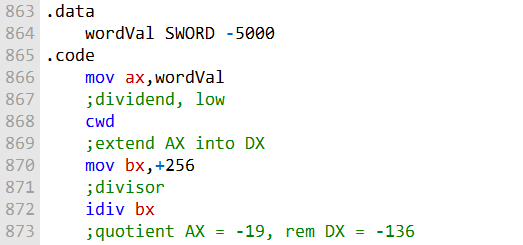
The CBW instruction sign-extends the AL register into the AX register.

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:



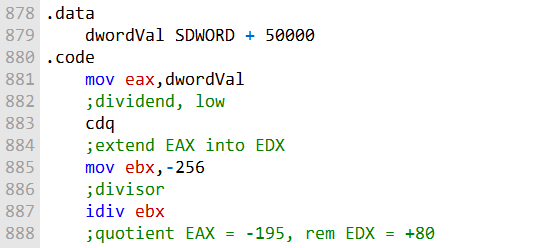
***Explanation:***

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:



***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.

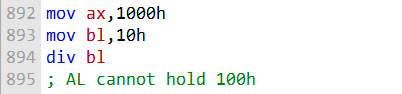
***==================***

***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:

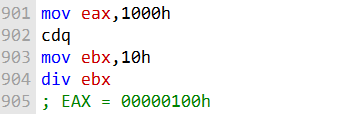


***Avoiding Divide Overflow***:

**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:

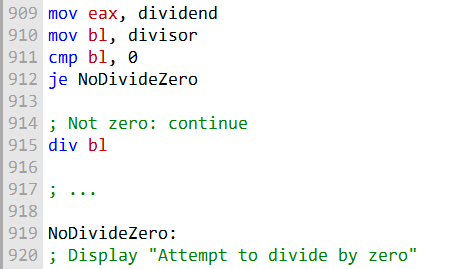


***Explanation:***

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:



***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:



can be broken down into the following assembly language instructions:



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:

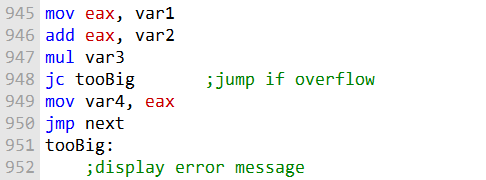


will multiply the value in the EAX register by the value of var3 and store the result in the EAX register. If the product of the multiplication is too large to fit into the EAX register, overflow will occur.

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:



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.

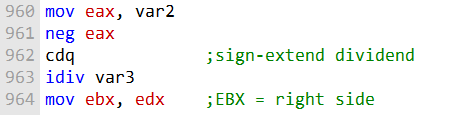
For example, the following assembly language instruction:



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:



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:



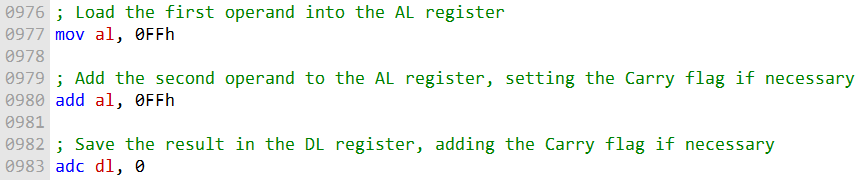
This code first sign-extends the 16-bit value to a 32-bit value in the EAX register before performing a signed division with the IDIV instruction.

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.

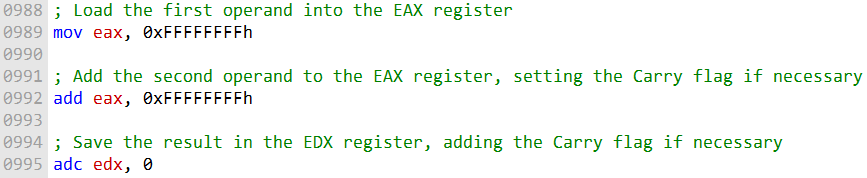
The ADC instruction is typically used to perform multi-byte or multi-word addition and subtraction operations.



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:



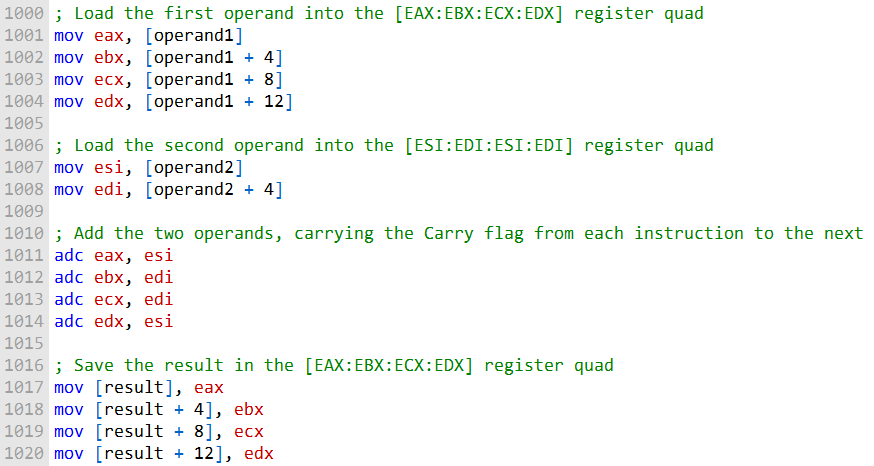
After the first instruction, the EAX register contains the value FFFFFFFFh and the Carry flag is set.

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:



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.

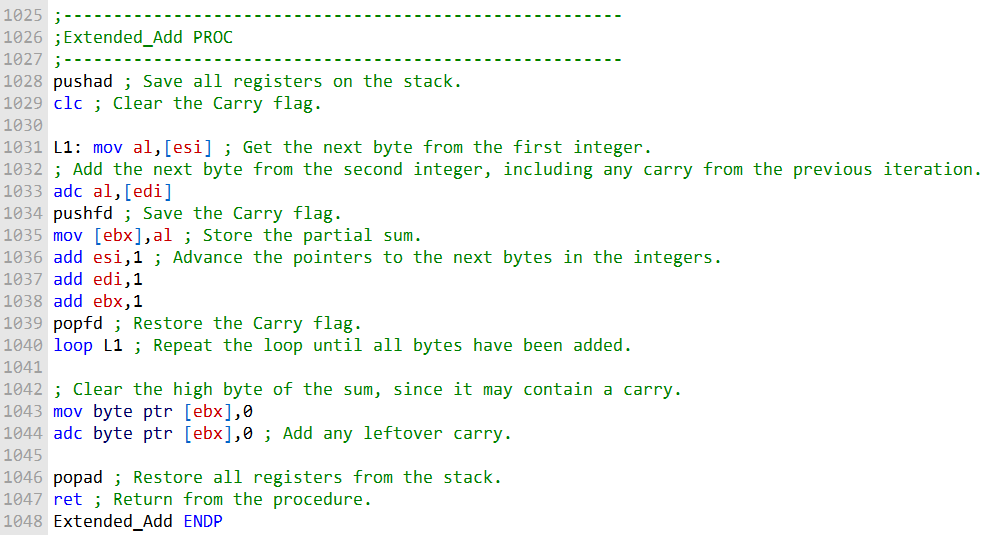
It works by iterating through the two integers, adding each corresponding byte and carrying over any carry from the previous iteration. The procedure takes four arguments:

**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:



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:



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:



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.

---------------------------------------------

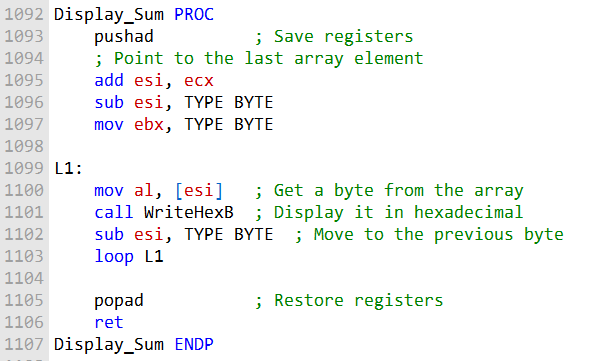
The Display\_Sum procedure below is related to the Extended\_Add procedure you provided in the previous question.

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:



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.

***======================***

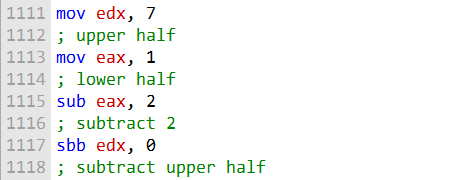
***Subtract with Borrow***

***======================***

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:



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:



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.



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.



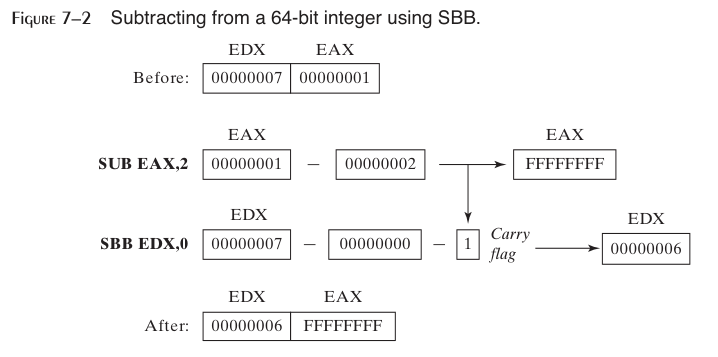
This instruction subtracts the value 2 from the lower 32 bits of the integer, which are stored in the EAX register. This may set the Carry flag if a borrow is required.



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.



**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.

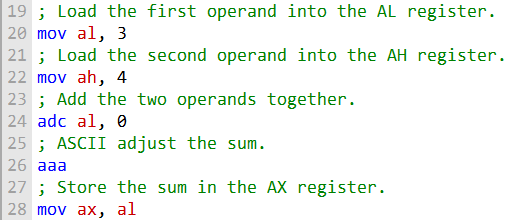
However, ASCII arithmetic does execute more slowly than binary arithmetic.

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:



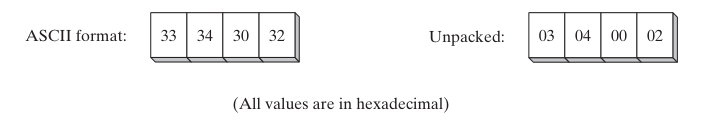
This code will add the two numbers together and store the sum in the AX register.

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

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

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

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

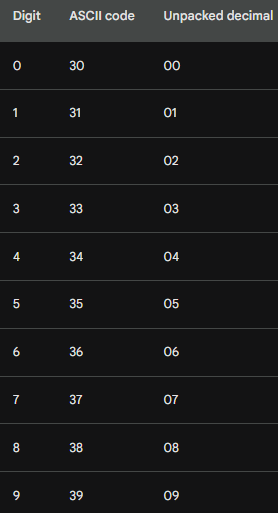


This means that the numbers in the block diagram are represented in two different formats: ASCII and unpacked decimal.

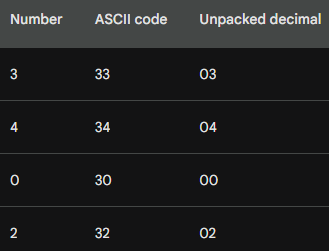
ASCII is a character encoding standard that assigns a unique code to each letter, number, and symbol. The ASCII codes for the digits 0 through 9 are 30 through 39, respectively.

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

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



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



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.

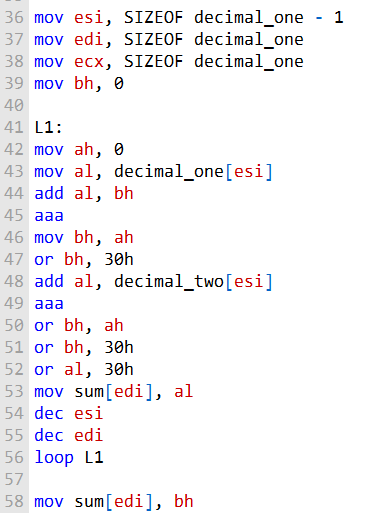
AAA (ASCII ADJUST AFTER ADDITION)

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:



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.

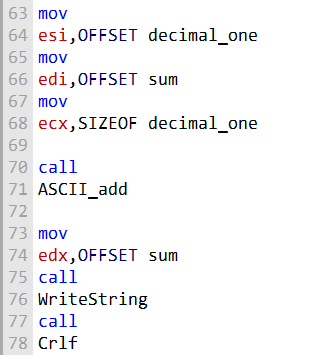
The loop at L1 iterates through the two operands, adding each corresponding digit and carrying over any carry from the previous iteration.

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:**



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.

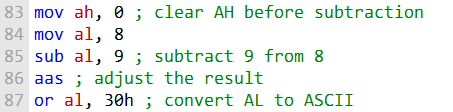
In this case, the AAS instruction subtracts 1 from the AH register and sets the Carry flag again.

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:

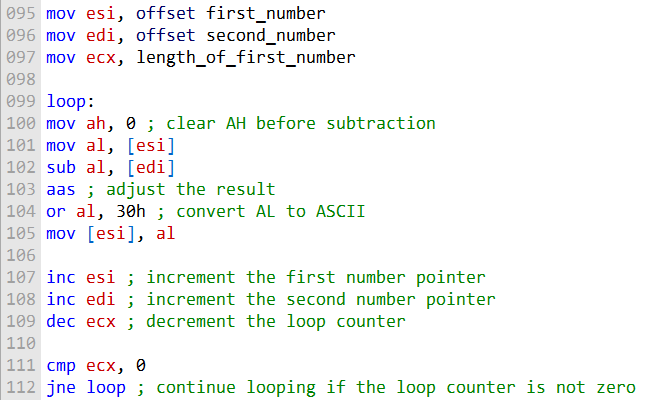


After the above code has executed, the AL register will contain the ASCII representation of the number -1, which is 45h.

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:



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)***

***========================================***

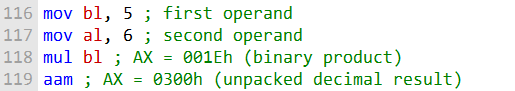
he AAM (ASCII adjust after multiplication) instruction is used to convert the binary product produced by the MUL instruction to unpacked decimal format.

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.

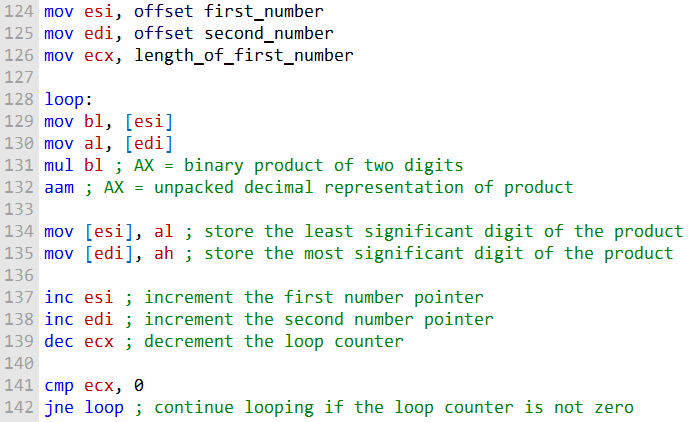
Here is an example of how to use the AAM instruction:



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:



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)***

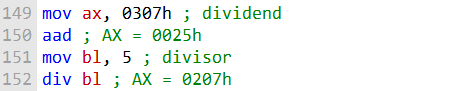
***========================================***

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.

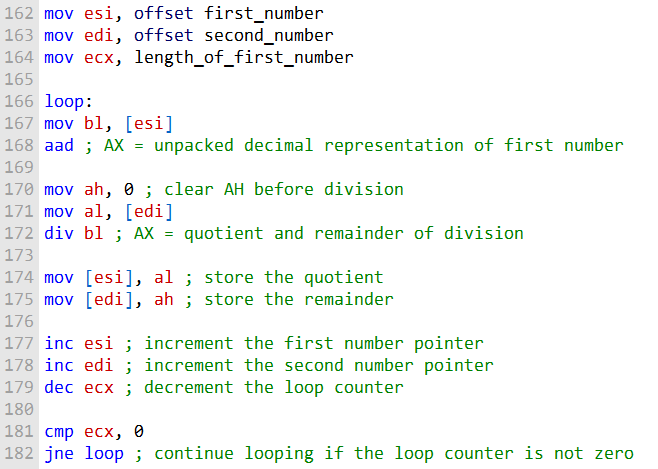
Here is an example of how to use the AAD instruction:



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:



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:



**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:



**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:



**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:



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.

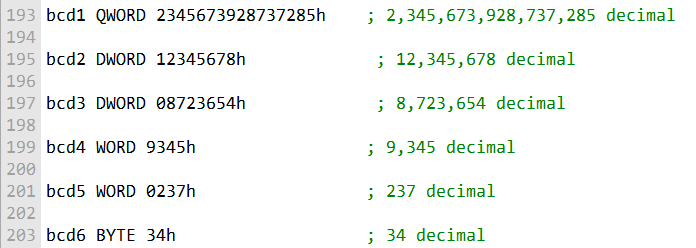


***However, there is one main disadvantage to using packed decimal arithmetic:***

**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:***

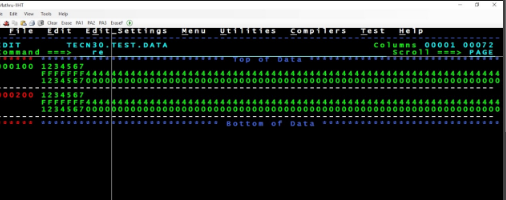


The following two instructions are used to adjust the result of an addition or subtraction operation on packed decimals:

* **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)***

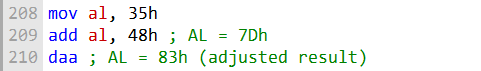
***========================================***

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:



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)***

***========================================***

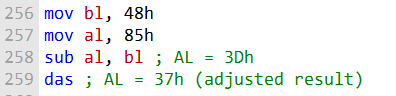
The DAS (decimal adjust after subtraction) instruction is used to convert the binary result of a SUB or SBB instruction in the AL register to packed decimal format.

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:



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.

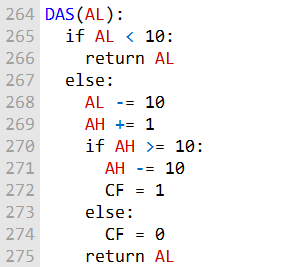
The DAS instruction adjusts the result to 37h, which is the packed decimal representation of the difference of 85 and 48.

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:



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.