Comparing Floating-Point Values

The passage below describes how to compare floating-point values and branch to a label based on the conditions.

To compare floating-point values, you can use the **FCOM instruction.**

The **FCOM instruction** compares the value in **ST(0) to the source operand**, which can be a memory operand or an FPU register.

After executing FCOM, you can use the **FNSTSW instruction** to move the FPU status word into AX.

The FPU status word contains the condition codes, which indicate the results of the comparison.

Once the **condition codes are in AX**, you can use the SAHF instruction to copy AH into the EFLAGS register.

The **EFLAGS register** contains the CPU status flags, including the Zero, Parity, and Carry flags.

Finally, you can use **conditional jump instructions to branch** to a label based on the condition codes.

For example, the following code branches to the label greater_than if the value in ST(0) is greater than the value in ST(1):

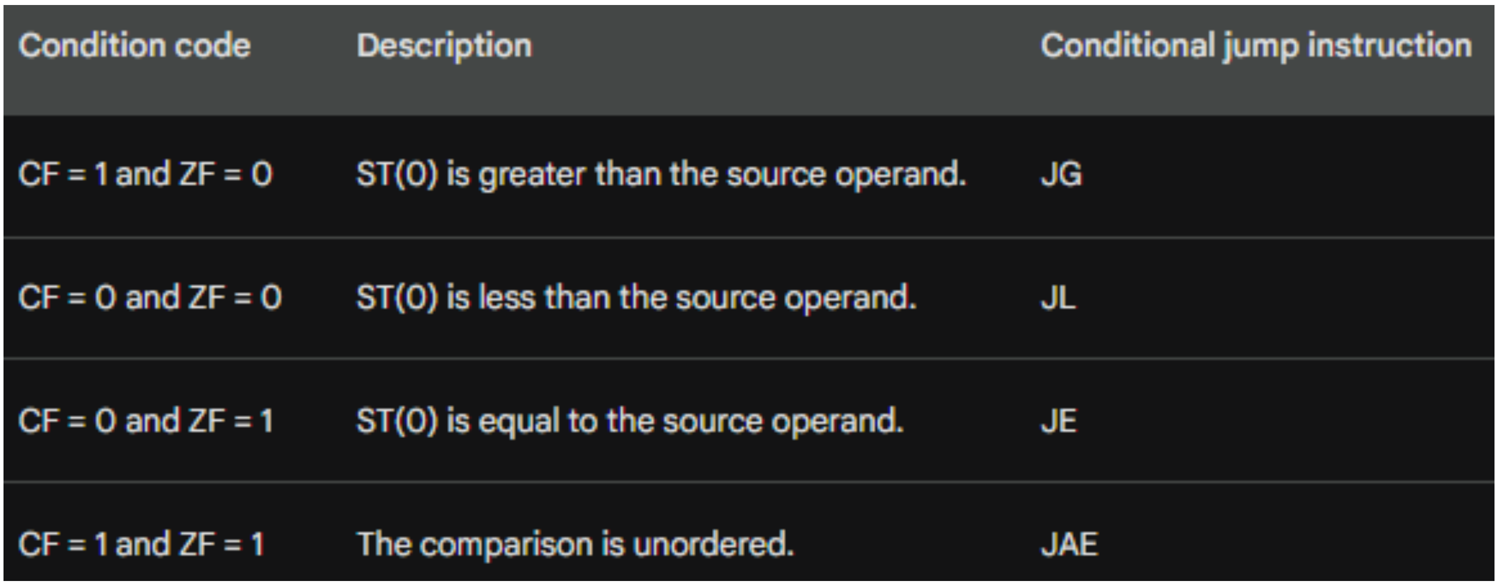
```
; Load the value in ST(0) to ST(0) (no change)
fld st(0)

; Load the value in ST(1) to ST(0)
fld st(1)

; Compare ST(0) to ST(1) and set FPU condition codes
fcom

; Move the FPU status word into the AX register
fnstsw ax

; Copy the AH register (containing FPU condition codes) to EFLAGS
sahf

; Check if the result of the comparison was greater (JG stands for "jump if greater")
jg greater_than

; Your code for handling the case where the comparison result was not greater would go here

greater_than:
; Your code for handling the case where the comparison result was greater goes here
```

In this code, the greater_than label marks the point where you'll continue if the result of the comparison (using FCOM) is greater.

If the result is not greater, you can place your code for handling that case in the section labeled "Your code for handling the case where the comparison result was not greater goes here."

This code sequence loads two values from the FPU stack, compares them using FCOM, transfers the FPU status word into the AX register (to obtain the FPU condition codes), copies the relevant condition code to EFLAGS using SAHF, and then uses the JG instruction to check if the result was greater, jumping to the greater_than label accordingly.

The following table shows the condition codes and the corresponding conditional jump instructions:



Condition code	Description	Conditional jump instruction
CF = 1 and ZF = 0	ST(0) is greater than the source operand.	JG
CF = 0 and ZF = 0	ST(0) is less than the source operand.	JL
CF = 0 and ZF = 1	ST(0) is equal to the source operand.	JE
CF = 1 and ZF = 1	The comparison is unordered.	JAE

The comparison is unordered if either operand is a NaN (Not a Number) or if the two operands have different signs and one of them is zero.

The ability to compare floating-point values and branch to a label based on the conditions is essential for many applications, such as graphics, scientific computing, and financial modeling.

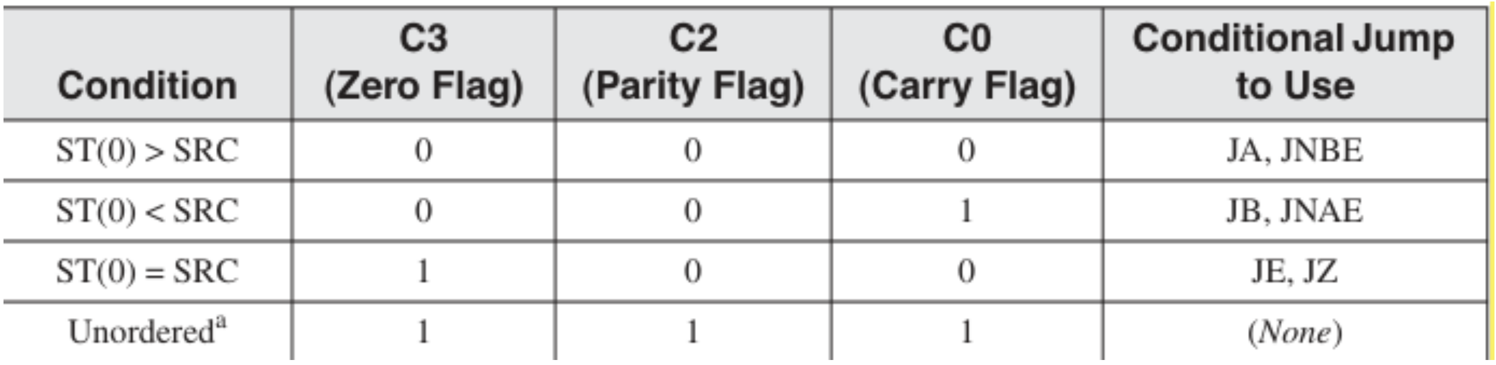
FCOM and FCOMI instruction

Let's describe the FCOMI instruction and how to use it to branch to a label based on the condition codes after comparing two floating-point values.

The FCOMI instruction is a P6 family instruction that compares floating-point values and sets the Zero, Parity, and Carry flags directly. This eliminates the need to use the FNSTSW and SAHF instructions to move the FPU status word into AX and copy AH into the EFLAGS register.

To use the FCOMI instruction, you simply pass the two floating-point values that you want to compare as operands. The FCOMI instruction will then set the Zero, Parity, and Carry flags based on the result of the comparison.

The following table shows the condition codes set by the FCOMI instruction:



Condition	C3 (Zero Flag)	C2 (Parity Flag)	C0 (Carry Flag)	Conditional Jump to Use
ST(0) > SRC	0	0	0	JA, JNBE
ST(0) < SRC	0	0	1	JB, JNAE
ST(0) = SRC	1	0	0	JE, JZ
Unordered[a]	1	1	1	*(None)*

The unordered condition is a special case that occurs when either operand is a NaN (Not a Number) or if the two operands have different signs and one of them is zero.

In this case, the condition codes are set to 111, which does not correspond to any conditional jump instruction.

Once the condition codes have been set by the FCOMI instruction, you can use conditional jump instructions to branch to a label based on the result of the comparison.

For example, the following code branches to the L1 label if the value in ST(0) is less than the value in ST(1):

```
; Load the value of Y onto the FPU stack
fld Y           ; ST(0) = Y

; Load the value of X onto the FPU stack, creating a stack with X on top and Y below
fld X           ; ST(0) = X, ST(1) = Y

; Compare the values on the FPU stack (X and Y)
fcomi ST(0), ST(1) ; Compare ST(0) to ST(1)

; Jump to label L1 if not below (if X is not less than Y), skipping the next instruction
jnb L1          ; Jump if not below (ST(0) not < ST(1)?), skip to L1

; If the jump condition is not met, set the integer N to 1
mov N, 1        ; N = 1

; Label L1 for reference
L1:
```

Here's a more detailed explanation of the code snippet:

fld Y: This instruction loads the value of Y (the second floating-point number) onto the FPU stack. As a result, ST(0) now contains the value of Y.

fld X: The next instruction loads the value of X (the first floating-point number) onto the FPU stack. This action shifts Y into ST(1), and X occupies ST(0), making it ready for comparison.

fcomi ST(0), ST(1): The fcomi instruction performs a comparison between the values in ST(0) and ST(1). In this case, it compares X (ST(0)) to Y (ST(1)). If ST(0) is not less than ST(1) (in other words, X is not less than Y), it sets the carry flag (CF) to 1.

jnb L1: The jnb (jump if not below) instruction checks the carry flag (CF) and transfers control to the label L1 if CF is not set. In this context, it means that if X is not less than Y, the code will skip the next instruction and move to the label L1.

mov N, 1: When the comparison determines that X is indeed less than Y, the code sets the integer N to 1. This assignment means that N will take the value 1 when the condition is met.

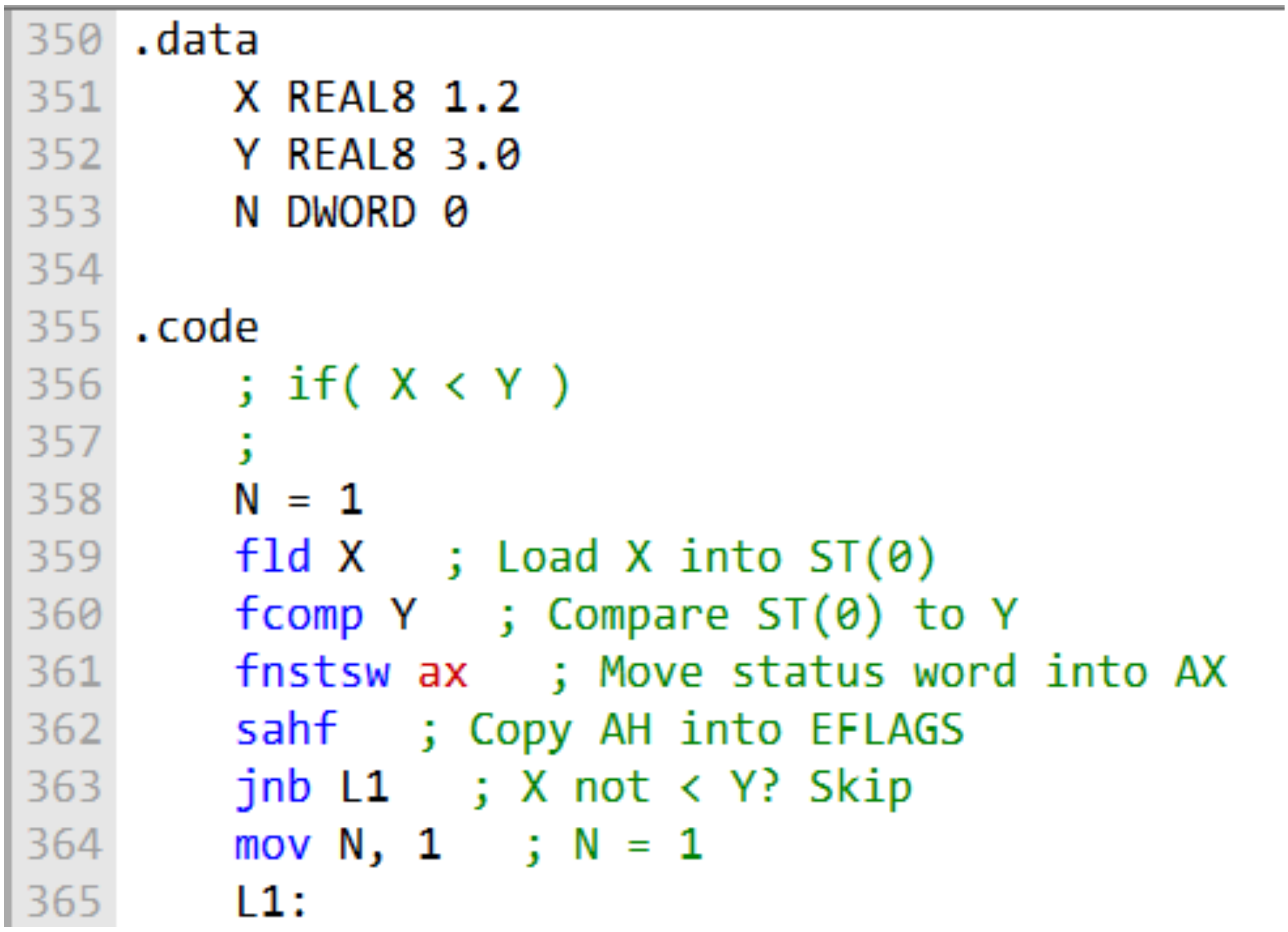
L1:: This label serves as a marker in the code, allowing you to return to this point if needed. In this particular scenario, it signifies the end of the conditional block.

The code snippet effectively compares two floating-point values (X and Y) and, depending on the result, sets the integer N to 1 if X is less than Y.

It demonstrates how the FPU condition codes, particularly the carry flag, can be used to control program flow based on floating-point comparisons.

FCOM compares two floating-point values on the FPU stack and sets condition flags based on the result.

Here's an example that illustrates how to perform a comparison between two double-precision floating-point numbers, X and Y, and then set an integer N based on the result. In this case, if X is less than Y, N is set to 1.



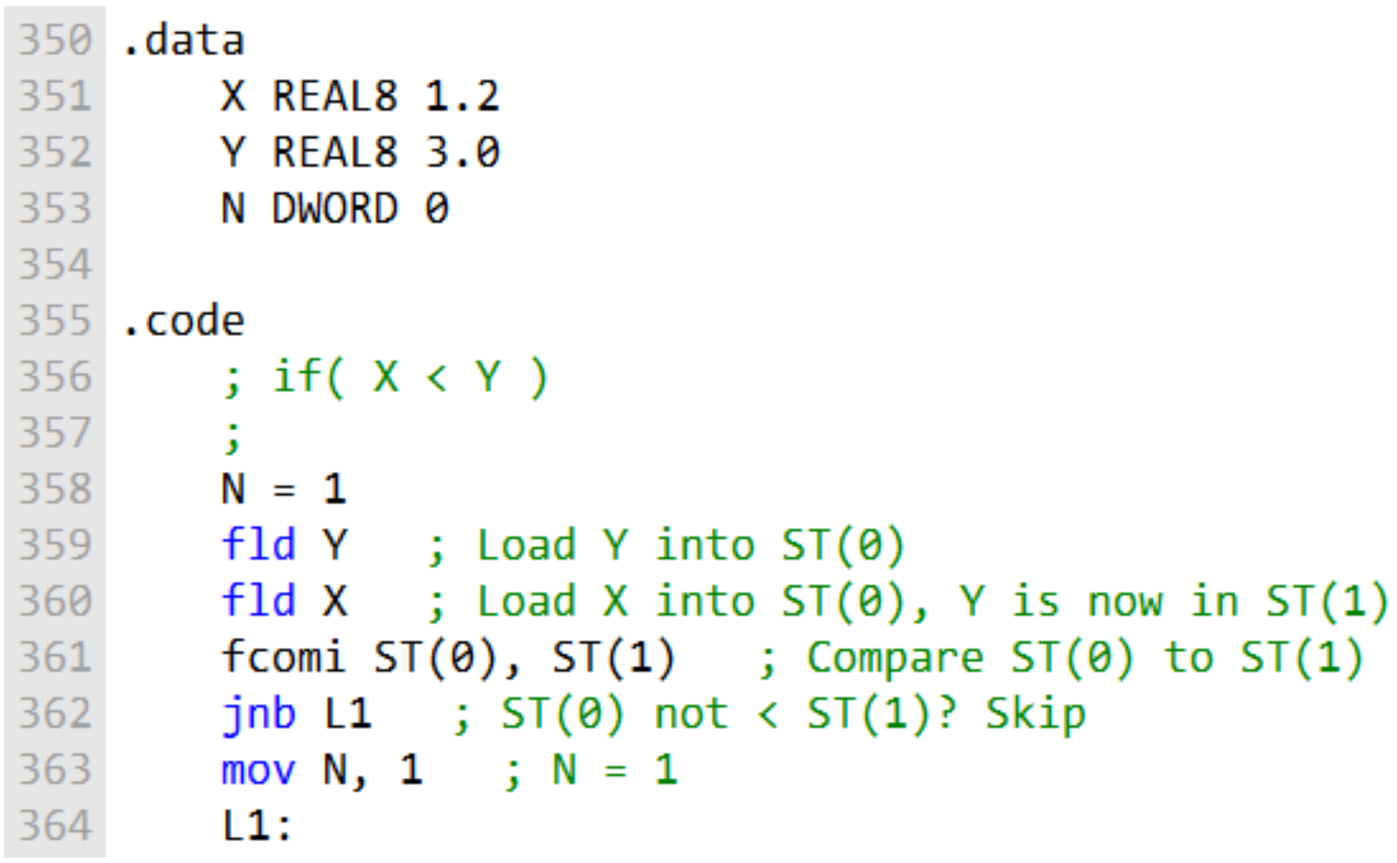
```
.data
    X REAL8 1.2
    Y REAL8 3.0
    N DWORD 0

.code
    ; if( X < Y )
    ;
    N = 1
    fld X    ; Load X into ST(0)
    fcomp Y    ; Compare ST(0) to Y
    fnstsw ax    ; Move status word into AX
    sahf    ; Copy AH into EFLAGS
    jnb L1    ; X not < Y? Skip
    mov N, 1    ; N = 1
    L1:
```

This code snippet demonstrates how the FPU condition codes are set and how conditional jumps are used to control program flow based on the comparison results.

An improvement to consider is the usage of the FCOMI instruction, available on newer processors like the Intel P6 family (e.g., Pentium Pro and Pentium II).

FCOMI is a more efficient instruction that directly compares two values on the FPU stack and sets condition flags. It's designed for improved performance in comparison operations compared to FCOM.

The FCOMI instruction can perform floating-point comparisons and set the Zero, Parity, and Carry flags directly. Here's the same comparison using FCOMI:



```
.data
    X REAL8 1.2
    Y REAL8 3.0
    N DWORD 0

.code
    ; if( X < Y )
    ;
    N = 1
    fld Y    ; Load Y into ST(0)
    fld X    ; Load X into ST(0), Y is now in ST(1)
    fcomi ST(0), ST(1)   ; Compare ST(0) to ST(1)
    jnb L1    ; ST(0) not < ST(1)? Skip
    mov N, 1    ; N = 1
    L1:
```

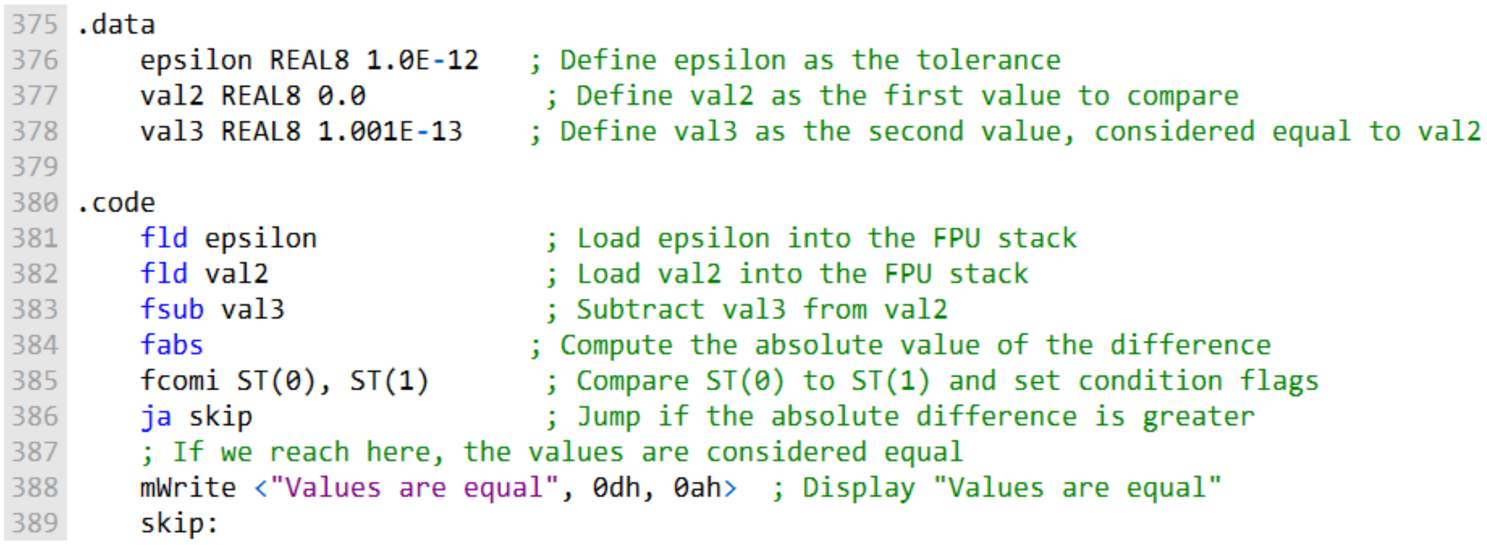
COMPARING FOR EQUALITY

The passage you sent describes how to compare floating-point values for equality in assembly language.

The proper way to compare floating-point values for equality is to take the **absolute value of their difference, |x - y|**, and compare it to a small user-defined value called **epsilon.**

This is because floating-point numbers are represented approximately in computers, and even small rounding errors can accumulate over time.

The following code in assembly language compares two floating-point values, val2 and val3, for equality using a tolerance of epsilon:



```
.data
    epsilon REAL8 1.0E-12   ; Define epsilon as the tolerance
    val2 REAL8 0.0           ; Define val2 as the first value to compare
    val3 REAL8 1.001E-13    ; Define val3 as the second value, considered equal to val2

.code
    fld epsilon              ; Load epsilon into the FPU stack
    fld val2                 ; Load val2 into the FPU stack
    fsub val3                ; Subtract val3 from val2
    fabs                    ; Compute the absolute value of the difference
    fcomi ST(0), ST(1)       ; Compare ST(0) to ST(1) and set condition flags
    ja skip                  ; Jump if the absolute difference is greater
    ; If we reach here, the values are considered equal
    mWrite <"Values are equal", 0dh, 0ah>  ; Display "Values are equal"
    skip:
```

This assembly code is designed to compare two floating-point values, val2 and val3, for equality while considering a tolerance level defined as epsilon.

The purpose of this code is to address a common challenge when working with floating-point numbers, which is that due to precision limitations, exact equality comparisons may not yield the expected results.

Therefore, it's a common practice to compare values within a certain tolerance range instead.

Here's a step-by-step explanation of the code:

Data Definitions: In the .data section, we define three variables:

- **epsilon:** This is set to 1.0E-12, representing the tolerance level for considering two values as equal.
- **val2:** This variable holds the first value for comparison, initialized with 0.0.
- **val3:** This variable holds the second value, which is considered equal to val2 and is set to 1.001E-13.

Code Section: In the .code section, the actual comparison is performed.

- **fld epsilon:** The fld instruction loads the value of epsilon onto the FPU stack. This value represents the maximum allowable difference between two values for them to be considered equal.

- **fld val2:** The next instruction loads the value of val2 onto the FPU stack. This sets up the first value for comparison.

- **fsub val3:** Here, we subtract the value of val3 from the value of val2. This computes the difference between the two values.

- fabs: The fabs instruction computes the absolute value of the difference obtained in the previous step. This ensures that we're only looking at the magnitude of the difference, regardless of its sign.

- **fcomi ST(0), ST(1):** The fcomi instruction performs the actual comparison between the top of the FPU stack (ST(0)) and the next value (ST(1)). It sets certain condition flags based on the result of this comparison. Specifically, it sets the Carry Flag (CF) and Zero Flag (ZF) based on the relationship between the two values.

- **ja skip:** The ja instruction is a conditional jump that stands for "jump if above." It checks the CF and ZF flags to determine whether the absolute difference (ST(0)) is greater than the second value (ST(1)), considering the defined tolerance. If the jump condition is met (i.e., if CF=1 and ZF=0), it means the absolute difference is greater than epsilon, and we jump to the skip label.

- **mWrite <"Values are equal", 0dh, 0ah>:** If the ja condition is not met, it implies that the absolute difference is within the tolerance range defined by epsilon, and the values are considered equal.

- In this case, the code proceeds to display the message "Values are equal." The mWrite instruction is not a standard x86 assembly instruction but is used here to represent a placeholder for a function that would display the message.

- The 0dh and 0ah are carriage return and line feed characters for formatting.

- **skip::** This label is where the code jumps to if the absolute difference is greater than epsilon. It serves as the point to continue execution after the comparison.

In summary, this code demonstrates how to compare floating-point values with a tolerance level (epsilon) to determine if they are equal within a certain range.

It accounts for the precision limitations of floating-point arithmetic and is a common practice when dealing with real-world numerical computations.

The code works summary:

• Load the tolerance value, epsilon, onto the FPU stack.
• Load the first value, val2, onto the FPU stack.
• Subtract the second value, val3, from val2.
• Take the absolute value of the difference.
• Compare the absolute value of the difference to the tolerance value.
• If the absolute value of the difference is greater than the tolerance value, jump to the skip label.
• Otherwise, display the message "Values are equal".
• This code will display the message "Values are equal" if the difference between val2 and val3 is less than or equal to the tolerance value, epsilon. Otherwise, the program will skip over the mWrite instruction and continue execution.

It is important to note that the choice of an appropriate tolerance value depends on the specific application.

For example, in a graphics application, a tolerance value of **$1.0E^{-6}$** m-ight be sufficient.

However, in a financial modeling application, a tolerance value of **$1.0E^{-12}$** or less might be required.