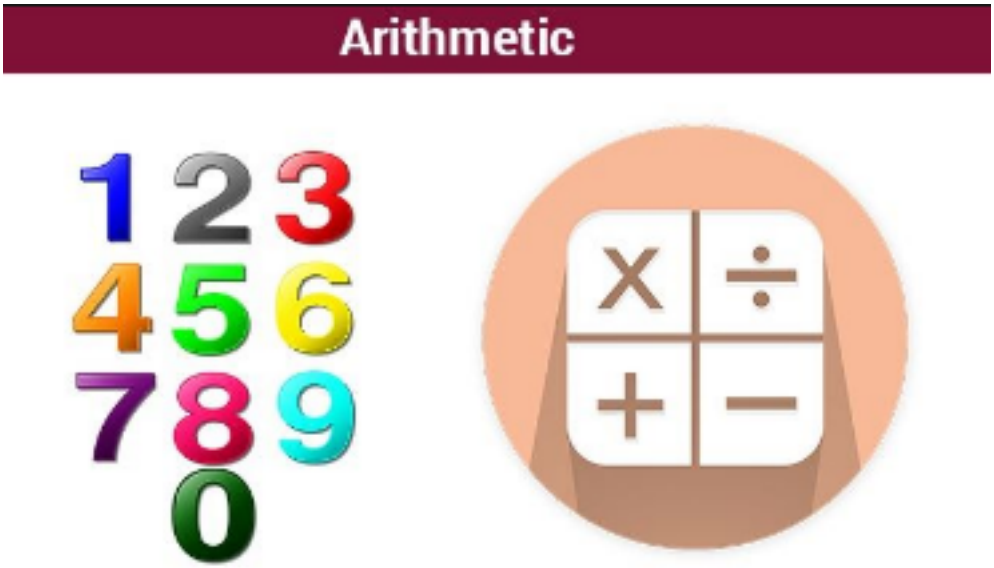
Floating-Point Instruction Set

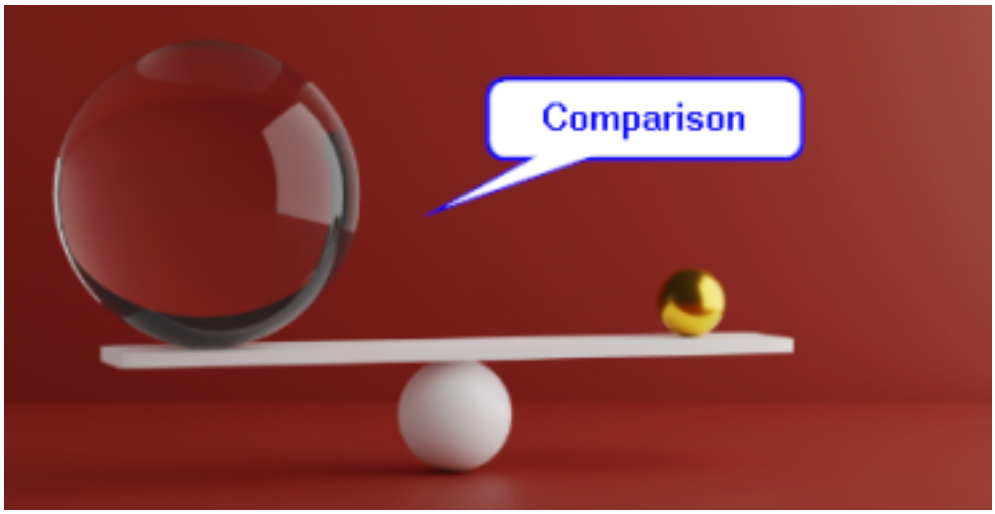
The floating-point instruction set is a set of instructions that the FPU uses to perform floating-point calculations. The instruction set is divided into the following categories:

Data transfer: These instructions move data between the FPU registers and memory.

Basic arithmetic: These instructions perform basic floating-point operations such as addition, subtraction, multiplication, and division.

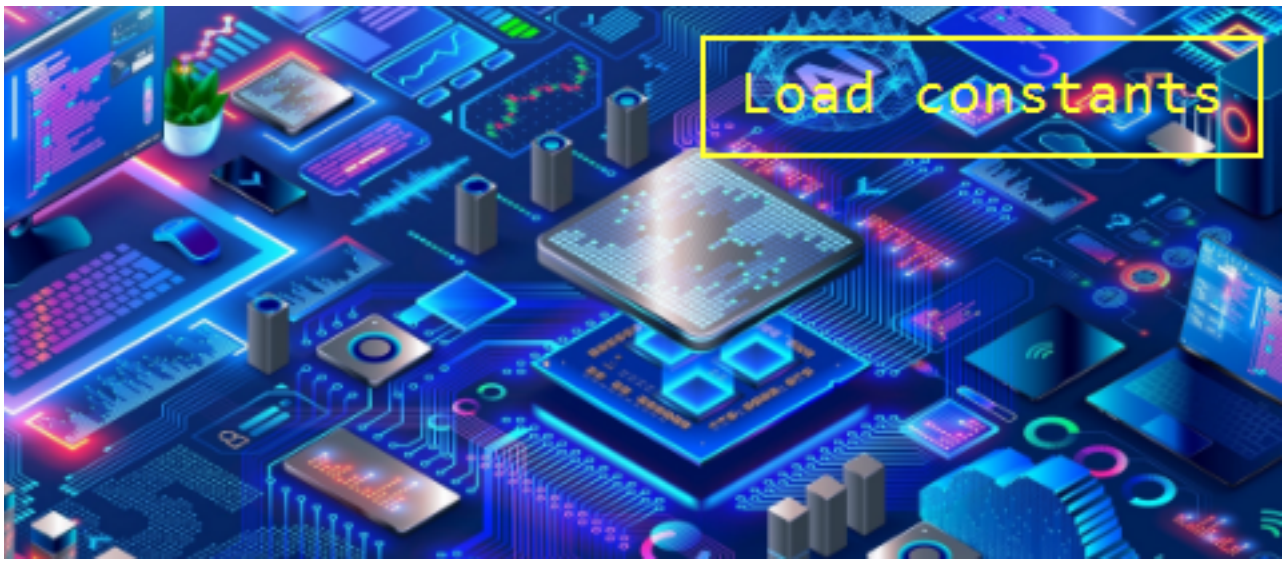


Comparison: These instructions compare two floating-point numbers and return a result indicating whether one number is greater than, equal to, or less than the other number.



Transcendental: These instructions perform transcendental functions such as sine, cosine, and tangent.

Load constants: These instructions load predefined constants into the FPU registers.



x87 FPU control: These instructions control the behavior of the FPU,

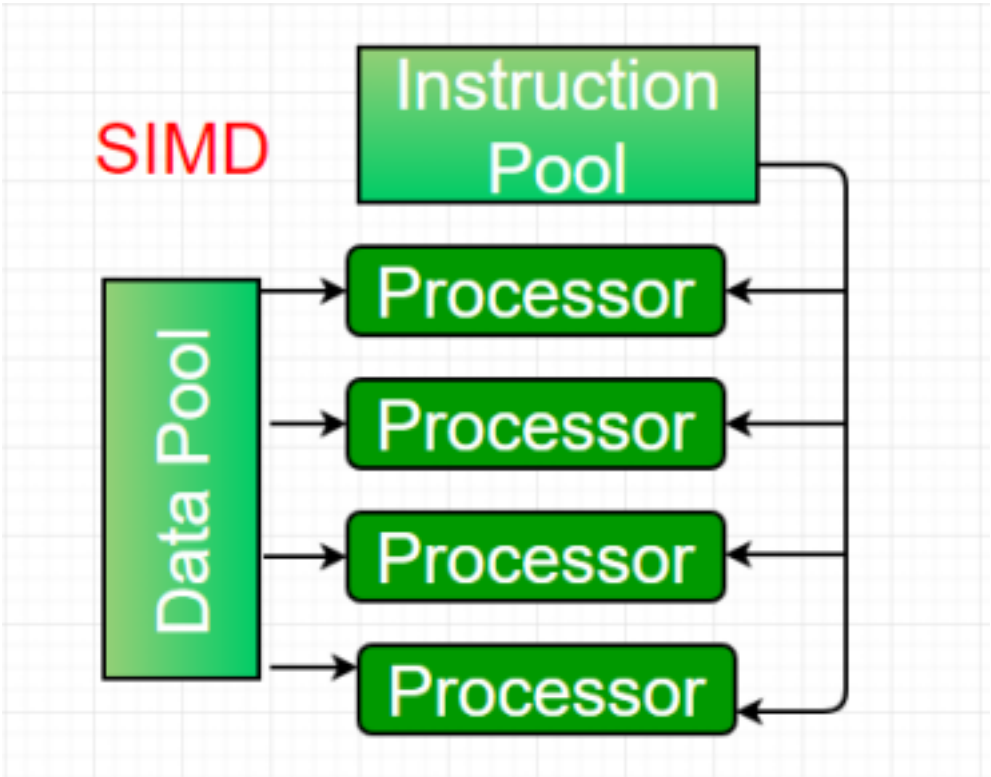
such as the rounding mode and the calculation precision.

```
const int mask = 0x0c00

asm(
     "        fldcw                  \n"
     "        fstcw                  \n"
     "        finit                  \n"
     "        fldl     %[mask]       \n"
     "        //& together           \n"
    //set bits 10 & 11

);
```

x87 FPU and SIMD state management: These instructions manage the state of the FPU, such as saving and restoring the FPU register stack.



Floating-point instruction names begin with the letter F.

The second letter of the instruction mnemonic indicates how a memory operand is to be interpreted: **B indicates a BCD operand, and I indicates a binary integer operand.** If neither is specified, the memory operand is assumed to be in real-number format.

memory operand is assumed to be in real-number format.

Floating-point instructions can have **zero operands, one operand, or two operands.**

If there are two operands, one must be a floating-point register. There are no immediate operands, but certain predefined constants can be loaded into the stack.

Integer operands must be loaded into the FPU from memory before they can be used in floating-point calculations. When storing floating-point values into integer memory operands, the values are automatically truncated or rounded into integers.

To initialize the FPU, you can use the **FINIT instruction.**

The FINIT instruction sets the FPU control word to **037Fh**, which masks all floating-point exceptions, sets rounding to nearest even, and sets the calculation precision to 64 bits.

It is recommended to **call FINIT at the beginning of your programs** so that you know the starting state of the FPU.

Here are some examples of floating-point instructions:

```
61 FLD [memory_address] ;Load a floating-point value from memory into the FPU stack.
62 FADD ST(0), ST(1) ;Add the top two values on the FPU stack and store the result at the top of the stack.
63 FSTP [memory_address] ;Store the top value on the FPU stack to memory.
```

FLOATING POINT DATATYPES

MASM supports the following floating-point data types:

- **QWORD:** 64-bit integer
- **TBYTE:** 80-bit (10-byte) integer
- **REAL4:** 32-bit (4-byte) IEEE short real
- **REAL8:** 64-bit (8-byte) IEEE long real
- **REAL10:** 80-bit (10-byte) IEEE extended real

When defining memory operands for FPU instructions, you must use one of these data types.

For example, to load a floating-point variable into the FPU stack, the variable must be defined as REAL4, REAL8, or REAL10.

Here is an example of how to use the REAL8 data type to define a floating-point variable and load it into the FPU stack:

```
; Define a double-precision floating-point variable named bigVal
.data
bigVal REAL8 1.212342342234234243E+864

; Start of the code section
.code
; Load the value of bigVal into the FPU stack
fld QWORD PTR [bigVal]

; Here, QWORD PTR is used to indicate that we are loading a double-precision (64-bit) value from memory.
; The brackets [] indicate that we are accessing the memory location pointed to by bigVal.

; The value of bigVal is now loaded onto the FPU stack.
```

This code will load the value stored in bigVal onto the FPU stack, making it ready for further floating-point operations.

The fld instruction loads the value of the variable bigVal into the FPU stack. The bigVal variable is defined as REAL8, so the fld instruction knows how to interpret the data in memory.

The floating-point data types supported by MASM are used in a wide variety of applications, including graphics, scientific computing, and financial modeling.

LOAD FLOATING POINT VALUES

This passage describes the FLD and FILD instructions.

The **FLD (load floating-point value)** instruction copies a floating-point operand to the top of the FPU stack (known as ST(0)).

The operand can be a 32-bit, 64-bit, or 80-bit memory operand (REAL4, REAL8, REAL10) or another FPU register.

The **FILD (load integer)** instruction converts a 16-, 32-, or 64-bit signed integer source operand to double-precision floating point and loads it into ST(0). The source operand's sign is **preserved.**

The following instructions load specialized constants on the stack.

They have no operands:

- **FLD1:** pushes 1.0 onto the register stack.
- **FLDL2T:** pushes $\log_2 10$ onto the register stack.
- **FLDL2E:** pushes $\log_2 e$ onto the register stack.
- **FLDPI:** pushes π onto the register stack.
- **FLDLG2:** pushes $\log_{10} 2$ onto the register stack.
- **FLDLN2:** pushes $\log_e 2$ onto the register stack.
- **FLDZ:** pushes 0.0 on the FPU stack.

Here are some examples of how to use the FLD and FILD instructions:

```
090 .data
091     dblOne    REAL8 234.56
092     dblTwo    REAL8 10.1
093     intVal    DWORD 100
094
095 .code
096     ; Load the value of dblOne onto the FPU stack.
097     fld     QWORD PTR [dblOne]
098
099     ; Load the value of dblTwo onto the FPU stack.
100     fld     QWORD PTR [dblTwo]
101
102     ; Load the integer value intVal onto the FPU stack, preserving its sign.
103     fild    DWORD PTR [intVal]
104
105     ; Load the mathematical constant p onto the FPU stack.
106     fldpi
107
108     ; Load the constant 1.0 onto the FPU stack.
109     fld1
```

In this section, you're defining data variables and loading them onto the FPU (Floating-Point Unit) stack:

dblOne and **dblTwo** are double-precision floating-point variables with values 234.56 and 10.1, respectively. These values are stored in 64 bits each.

bits each.

intVal is a 32-bit integer variable with a value of 100.

Now, let's explain the code part:

fld QWORD PTR [dblOne]: This instruction loads the value of dblOne (234.56) onto the FPU stack. QWORD PTR indicates that you're accessing a 64-bit value in memory. The fld instruction is used for loading floating-point values.

fld QWORD PTR [dblTwo]: This instruction loads the value of dblTwo (10.1) onto the FPU stack, similar to the previous instruction.

fild DWORD PTR [intVal]: This instruction is different. It loads the 32-bit integer value intVal (100) onto the FPU stack. The fild instruction is used to load integers and preserves the sign. It converts the integer into a floating-point format in the FPU.

fldpi: This instruction loads the mathematical constant π (pi) onto the FPU stack. It's a predefined constant. (we described it above)

fld1: This instruction loads the constant 1.0 onto the FPU stack. It's a predefined constant representing the floating-point value 1.0.

FST AND FSTP INSTRUCTIONS

Here is a concise explanation of the FST and FSTP instructions:

FST (Store Floating-Point Value):

• Copies a floating-point operand from the top of the FPU stack to memory OR
• Stores the value on top of the FPU stack (ST(0)) to memory.
• Supports memory operand types (REAL4, REAL8, REAL10) and can store to another FPU register.
• Does not pop the stack.

The FST instruction in x86 assembly language serves a dual purpose. It can be used to copy a floating-point operand from the top of the FPU stack (ST(0)) to memory. Additionally, it can store the value at the top of the FPU stack directly to memory.

FST supports various memory operand types such as REAL4, REAL8, and REAL10, and it can also be used to store values to another FPU register. Crucially, FST does not pop the stack, leaving the values on the stack intact for further use in calculations.

FSTP (Store Floating-Point Value and Pop):

- Copies the value in ST(0) to memory and pops ST(0) off the stack.
- Stores the value on top of the FPU stack (ST(0)) to memory and pops the stack.
- Supports the same memory operand types as FST.
- Physically removes values from the stack, changing the TOP pointer.

In contrast, the FSTP instruction not only copies the value in ST(0) to memory but also pops (removes) it from the FPU stack. This dual functionality makes it particularly useful for efficient stack management. Similar to FST, FSTP supports the same memory operand types and can store to other FPU registers.

By physically removing values from the stack, FSTP changes the TOP pointer, effectively "popping" the value from the stack, making it a preferred choice when you no longer need the value in the stack for subsequent operations.

Here's a code example illustrating the use of these instructions:

```
.data
    dblOne   REAL8 234.56
    dblTwo   REAL8 10.1
    dblThree REAL8 0.0
    dblFour  REAL8 0.0

.code
    ; Load values onto the FPU stack
    fld QWORD PTR [dblOne]  ; ST(0) = 234.56
    fld QWORD PTR [dblTwo]  ; ST(0) = 10.1, ST(1) = 234.56

    ; Store values from the stack to memory without popping
    fst QWORD PTR [dblThree]  ; Stores 10.1 in dblThree
    fst QWORD PTR [dblFour]   ; Stores 10.1 in dblFour

    ; Reset the values
    fld QWORD PTR [dblOne]  ; ST(0) = 234.56
    fld QWORD PTR [dblTwo]  ; ST(0) = 10.1, ST(1) = 234.56

    ; Store values from the stack to memory and pop
    fstp QWORD PTR [dblThree]  ; Stores 10.1 in dblThree and pops ST(0)
    fstp QWORD PTR [dblFour]   ; Stores 234.56 in dblFour and pops ST(0)
```

In this assembly code, you're working with the FPU (Floating-Point Unit) to load values onto the FPU stack and then store them in memory using the FST and FSTP instructions. Let's delve into the details without code boxes:

You begin by defining four data variables:

• **dblOne** and **dblTwo** are double-precision floating-point variables with values 234.56 and 10.1, respectively.

• **dblThree** and **dblFour** are initialized as 0.0 but will be used to store values from the FPU stack.

The fld instruction is used to load values onto the FPU stack. You load the value of dblOne (234.56) into ST(0) and then dblTwo (10.1) into ST(0), effectively pushing the previous value down in the stack (ST(1) = 234.56).

You then use the fst instruction to store the value at the top of the FPU stack (ST(0)) into memory without popping the value from the stack. You store this value in dblThree and dblFour, so both variables receive the value 10.1.

To reset the values for demonstration, you reload dblOne and dblTwo onto the FPU stack.

The fstp instruction is employed to store the value from the top of the FPU stack (ST(0)) into memory and simultaneously pop it off the stack.

This effectively removes the value from ST(0) and stores it in dblThree and dblFour. The first fstp stores 10.1 in dblThree, while the second stores 234.56 in dblFour.

The stack's top pointer (TOP) is incremented after each fstp, shifting the remaining values up.

In summary, this code illustrates how the FST and FSTP instructions allow you to move values between the FPU stack and memory.

FST stores the value in memory without altering the stack, while FSTP stores the value in memory and pops it from the stack.

These instructions are essential for efficient floating-point data handling in assembly programming.

FLOATING POINT ARITHEMETIC

The arithmetic instructions in x86 assembly allow you to perform basic arithmetic operations on floating-point numbers.

These instructions support the same memory operand types as **FLD (load)** and **FST (store)**, meaning operands can be **indirect**, **indexed**, **base-indexed**, and more. The key arithmetic instructions are as follows:

FCHS (Change Sign): This instruction reverses the sign of the floating-point value in ST(0), effectively changing it from positive to negative or vice versa.

FABS (Absolute Value): FABS clears the sign of the number in ST(0) to obtain its absolute value, effectively making it positive.

FADD (Add): FADD performs addition. It can add two values from the

FPU stack or add a value from memory to a value on the stack.

FSUB (Subtract): FSUB subtracts the source value from the destination value.

FSUBR (Reverse Subtract): FSUBR subtracts the destination value from the source value.

FMUL (Multiply): FMUL multiplies the source value by the destination value.

FDIV (Divide): FDIV divides the destination value by the source value.

FDIVR (Reverse Divide): FDIVR divides the source value by the destination value.

FCHS and FABS:

FCHS is used to change the sign of the value in ST(0), essentially negating it if it was positive or making it positive if it was negative.

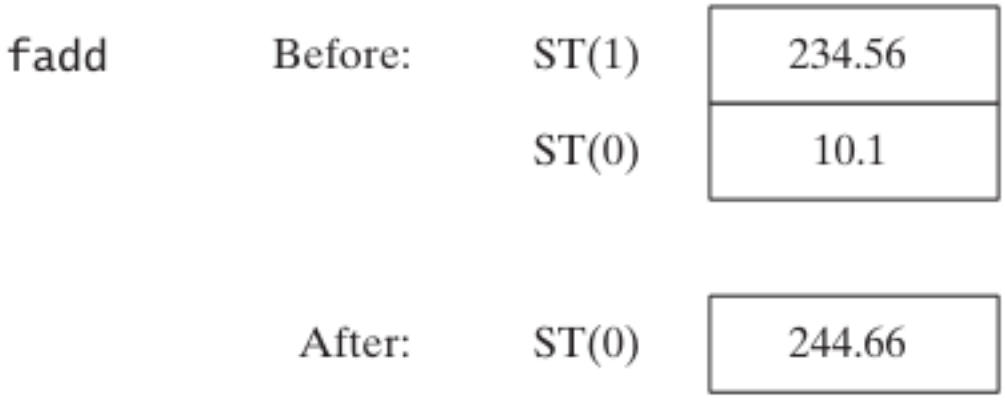
FABS, on the other hand, computes the absolute value by clearing the sign of the value in ST(0).

FADD (Add)

FADD can be used in different formats:

With no operands

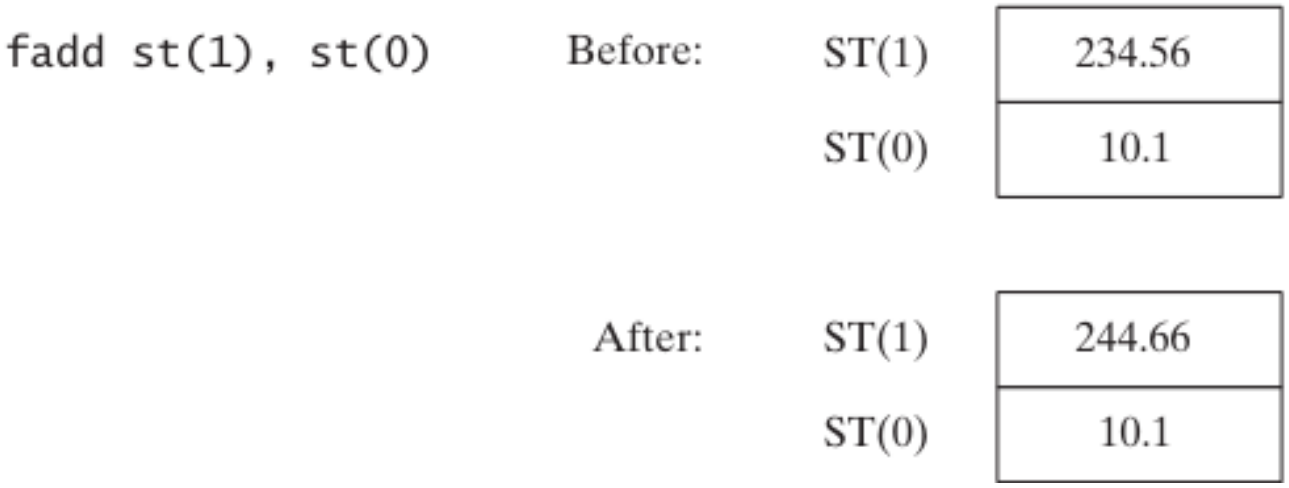
It adds ST(0) to ST(1), storing the result in ST(1). It then pops ST(0), leaving the result on top of the stack.



With a memory operand (REAL4 or REAL8)

It adds the value in memory to the value in ST(0).

With a register number (i), it adds the value in ST(i) to ST(0).



The format **"FADD ST(i), ST(0)"** adds ST(0) to ST(i), effectively **swapping their positions.**

Memory Operand with FADD

When FADD is used with a memory operand, it adds the value stored in memory to ST(0), the top of the FPU stack.

For instance, **"fadd mySingle"** adds the value stored in the memory location "mySingle" to ST(0), effectively increasing its value.

```
.data
mySingle REAL4 5.5          ; Define a single-precision floating-point value in memory
myDouble REAL8 10.1         ; Define a double-precision floating-point value in memory
result  REAL8 0.0           ; Define a memory location to store the result

.code
main:
    fld     myDouble        ; Load myDouble onto the FPU stack
    fadd    mySingle        ; Add mySingle to ST(0)

    fstp    result          ; Store the result in the "result" memory location

    ; Exit the program (endless loop to prevent immediate termination)
    mov     eax, 1          ; Specify the exit system call
    int     0x80            ; Call the kernel

    ; Rest of the program here (not shown in this example)
```

In this example, we have defined two floating-point values, mySingle and myDouble, in memory.

We load myDouble onto the FPU stack using the fld instruction, and then we use fadd to add the value of mySingle to the value in ST(0).

Finally, we store the result in the memory location named result using fstp. The program then exits.

Please note that this code is provided for educational purposes and assumes a Linux environment.

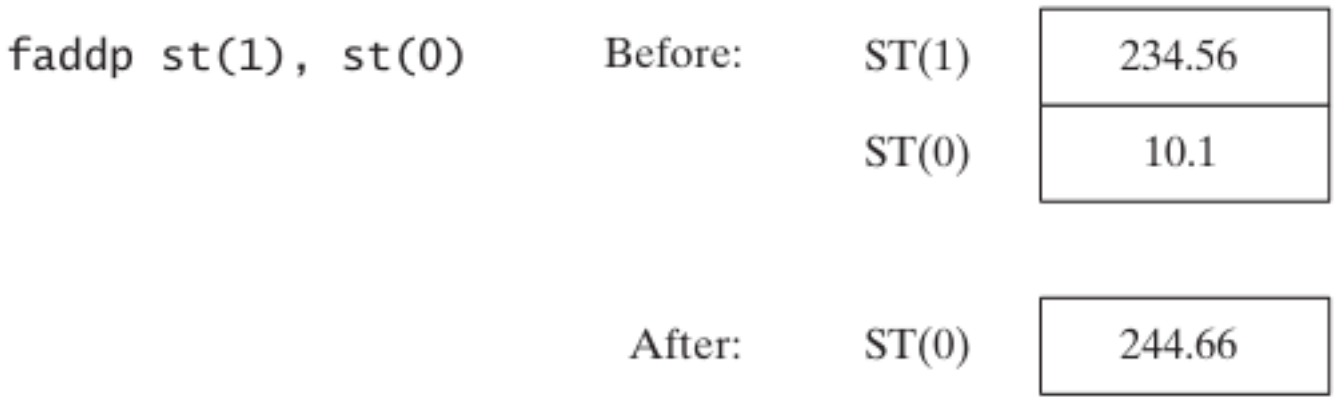
The system call (int 0x80) is specific to Linux, and you may need to adjust it for other operating systems. Additionally, make sure to use the appropriate assembly syntax for your assembler and platform.

FADDP (Add with Pop)

FADDP, the "**add with pop**" instruction, performs an addition operation and then pops the value in ST(0) from the stack.

This is particularly useful for efficiently managing the stack.

The syntax for FADDP is "**FADDP ST(1), ST(0)**", indicating that the result of the addition is stored in ST(1), and ST(0) is removed from the stack.



FIADD (Add Integer)

The FIADD instruction is used to add an integer value to ST(0) after converting the source operand to double-extended precision floating-point format.

It **supports two operand types: m16int** and **m32int**, which refer to 16-bit and 32-bit integers, respectively.

For example, **"fiadd myInteger"** adds the value stored in the memory location **"myInteger"** to ST(0) after converting it to a floating-point format.

```
.data
myInteger DWORD 1        ; Define a 32-bit integer value in memory
result REAL8 0.0         ; Define a memory location to store the result

.code
main:
    finit                ; Initialize the FPU
    fild    myInteger    ; Load myInteger onto the FPU stack and convert it to a floating-point value
    fadd                 ; Add ST(0) to ST(1), effectively adding myInteger to ST(0)

    fstp    result       ; Store the result in the "result" memory location

    ; Exit the program (endless loop to prevent immediate termination)
    mov     eax, 1       ; Specify the exit system call
    int     0x80         ; Call the kernel

    ; Rest of the program here (not shown in this example)
```

In this code:

- We define an integer value, myInteger, in memory and a memory location named result to store the result.

- We use the finit instruction to initialize the FPU.

• We load the integer value myInteger onto the FPU stack and convert it to a floating-point value using fild.

• Then, we use fadd to add the value in ST(0) to ST(1), which effectively adds the value of myInteger to the value in ST(0).

• Finally, we store the result in the result memory location using fstp.

The system call (int 0x80) is specific to Linux, and you may need to adjust it for Windows or other operating systems. Make sure to use the appropriate assembly syntax for your platform.

FSUB (Floating-Point Subtract):

The FSUB instruction performs subtraction in the FPU. It subtracts a source operand from a destination operand, storing the difference in the destination operand.

The destination is always an FPU register (ST(0)), and the source can be either an FPU register or memory. It supports the same memory operand types as FADD.

The operation of FSUB is similar to that of FADD, except it performs subtraction.

Without Operands

For example, if used **without operands**, it subtracts ST(0) from ST(1), temporarily storing the result in ST(1). ST(0) is then popped from the stack, leaving the result on top.

Examples:

• fsub mySingle: This instruction subtracts the value in mySingle from ST(0), effectively performing ST(0) -= mySingle.

• fsub array[edi*8]: Here, the value stored in array[edi*8] is subtracted from ST(0). This operation doesn't pop the stack, leaving the result in ST(0).

FSUBP (Floating-Point Subtract with Pop)

The FSUBP instruction is similar to FSUB but with an additional step.

It performs the subtraction and then pops (removes) ST(0) from the stack.

This means that after executing FSUBP, the result is left in ST(0), and the stack pointer is adjusted accordingly.

```
.data
mySingle REAL8 5.0           ; Example value for mySingle
array REAL8 10.0              ; Example value for array[edi*8]

.code
; Subtract mySingle from ST(0) using FSUB
fld QWORD PTR [mySingle]     ; Load mySingle onto the FPU stack
fsub                         ; Subtract mySingle from ST(0)
; The result is now in ST(0)

; Subtract array[edi*8] from ST(0) using FSUB
fld QWORD PTR [array+edi*8]  ; Load array[edi*8] onto the FPU stack
fsub                         ; Subtract array[edi*8] from ST(0)
; The result is now in ST(0)

; Now, let's use FSUBP to subtract and pop ST(0)

; Subtract 5.0 from ST(0) and pop the stack
fld QWORD PTR [mySingle]     ; Load mySingle onto the FPU stack
fsubp                        ; Subtract mySingle from ST(0) and pop ST(0)
; The result is left in ST(0), and ST(0) is removed from the stack
```

Example:

fsubp ST(1), ST(0): This instruction subtracts ST(0) from ST(1) and leaves the result in ST(0) while removing the old ST(0) from the stack.

FISUB (Floating-Point Subtract Integer):

The FISUB instruction is used to subtract an integer from a floating-point value in ST(0).

It first converts the source operand (an integer) to double-extended precision floating-point format before performing the subtraction.

precision floating-point format before performing the subtraction.

This allows you to perform arithmetic operations involving integers and floating-point numbers in the FPU.

```
.data
myInteger DWORD 3      ; Example integer value for myInteger

.code
; Subtract an integer from ST(0) using FISUB
fld QWORD PTR [myInteger]   ; Load myInteger as a 64-bit integer onto the FPU stack
fisub                       ; Subtract myInteger from ST(0)
; The result is now in ST(0)

; Use FSUBP to subtract ST(0) from ST(1) and pop ST(0)
fsubp ST(1), ST(0)
; After executing this, the result is in ST(0), and the old ST(0) has been removed from the stack
```

These code example show how to use FISUB to subtract an integer from an FPU register and FSUBP to subtract and pop the stack, as well as the explanation you provided for fsubp ST(1), ST(0).

Examples:

fisub m16int: This instruction subtracts the 16-bit integer from ST(0) after converting it to a floating-point format.

fisub m32int: Similarly, this instruction subtracts the 32-bit integer from ST(0) after converting it to a floating-point format.

```
.data
    my16BitInt WORD 5    ; Example 16-bit integer value
    my32BitInt DWORD 10  ; Example 32-bit integer value

.code
    ; Subtract a 16-bit integer from ST(0) using FISUB
    fld QWORD PTR [my16BitInt]  ; Load my16BitInt as a 16-bit integer onto the FPU stack
    fisub                       ; Subtract my16BitInt from ST(0)
    ; The result is now in ST(0)

    ; Reset ST(0) to another value
    fld QWORD PTR [my32BitInt]  ; Load my32BitInt as a 32-bit integer onto the FPU stack

    ; Subtract a 32-bit integer from ST(0) using FISUB
    fisub                       ; Subtract my32BitInt from ST(0)
    ; The result is now in ST(0)
```

These code examples demonstrate how to use FISUB to subtract 16-bit and 32-bit integers from an FPU register after converting them to

floating-point format.

This provides flexibility for performing arithmetic operations with different data types and operands in floating-point arithmetic.

FMUL (floating-point multiply)

The FMUL (floating-point multiply) instruction is used to multiply a source operand by a destination operand, and the result is stored in the destination operand, which is always an FPU register.

The source can be either an FPU register or a memory operand, similar to FADD and FSUB:

FMUL:

FMUL can be used **with no operands**, in which case it multiplies ST(0) by ST(1), and the result is temporarily stored in ST(1). ST(0) is then popped from the stack, leaving the product in ST(0).

When used **with a memory operand**, FMUL multiplies ST(0) by the value stored in the memory operand.

FMULP (floating-point multiply with pop) is similar to FMUL, but it pops ST(0) from the stack after performing the multiplication.

This means that after executing FMULP, the result is left in ST(0), and the stack pointer is adjusted accordingly.

FIMUL (floating-point integer multiply) is identical to FIADD, except it performs multiplication instead of addition.

It converts the source operand (an integer) to double-extended precision floating-point format before performing the multiplication.

Here's an example in assembly code:

```
.data
    mySingle REAL4 3.0          ; Example single-precision floating-point value
    my16BitInt WORD 2           ; Example 16-bit integer
    my32BitInt DWORD 4          ; Example 32-bit integer
    result REAL8 0.0            ; Storage for results

.code
    ; Multiply ST(0) by a single-precision floating-point value
    fld QWORD PTR [mySingle]   ; Load mySingle as a single-precision value onto the FPU stack
    fmul                       ; Multiply ST(0) by mySingle
    fstp QWORD PTR [result]  ; Store the result in result

    ; Reset ST(0) to another value
    fld QWORD PTR [my32BitInt]  ; Load my32BitInt as a 32-bit integer onto the FPU stack

    ; Multiply ST(0) by another value
    fld QWORD PTR [mySingle]   ; Load another single-precision value onto the FPU stack
    fmul                       ; Multiply ST(0) by the new single-precision value
    fstp QWORD PTR [result]  ; Store the result in result

    ; Reset ST(0) to another value
    fld QWORD PTR [my16BitInt]  ; Load my16BitInt as a 16-bit integer onto the FPU stack

    ; Multiply ST(0) by another value
    fld QWORD PTR [mySingle]   ; Load another single-precision value onto the FPU stack
    fimul WORD PTR [esp]     ; Multiply ST(0) by the 16-bit integer
    fstp QWORD PTR [result]  ; Store the result in result
```

Let's explain the provided code:

Variable Initialization:

In the .data section, we define several variables. mySingle is set to a single-precision floating-point value (3.0), my16BitInt to a 16-bit integer (2), my32BitInt to a 32-bit integer (4), and result is initialized to store the results of the various multiplications.

Single-Precision Floating-Point Multiplication:

The first part of the code demonstrates single-precision floating-point multiplication. We load the value stored in mySingle onto the FPU stack using fld.

This places the single-precision value in the top of the FPU stack, which is ST(0). We then use the fmul instruction to multiply the value in ST(0) (the value from mySingle) by itself.

The result remains in ST(0). Finally, the fstp instruction is used to store the result in the result variable.

Resetting the FPU Stack:

We reset the FPU stack to a different value (my32BitInt) using another fld instruction. This prepares the FPU stack for the next multiplication operation.

Single-Precision Floating-Point Multiplication (Again):

Similar to the first part, we load a single-precision value onto the FPU stack, this time using fld to place a new value into ST(0).

We then use the fmul instruction to multiply the value in ST(0) by the new single-precision value. The result remains in ST(0).

Once again, the fstp instruction is used to store the result in the result variable.

Resetting the FPU Stack Again:

We reset the FPU stack to a different value (my16BitInt) using yet another fld instruction.

This prepares the FPU stack for the next multiplication operation.

Integer Multiplication with FIMUL:

In this part, we load a 16-bit integer value onto the FPU stack using fld. We then load another single-precision value onto the FPU stack.

However, instead of using fmul as in the previous scenarios, we use the fimul instruction with WORD PTR [esp].

This performs integer multiplication between the 16-bit integer in ST(0) and the single-precision value, and the result remains in ST(0).

Finally, the fstp instruction is used to store the result in the result variable.

Completion and Result Storage:

The code concludes with the result of the integer multiplication stored in the result variable.

This code showcases various scenarios of floating-point and integer multiplication operations in the x86 FPU, demonstrating how to load values, perform multiplications, and store the results for different data types and operand combinations.

FDIV (Floating-Point Division):

The FDIV instruction is used to perform division operations within the x86 FPU.

It divides the value in a destination operand by the value in a source operand, with the result stored in the destination operand.

The destination operand must be an FPU register, while the source operand can be either an FPU register or a memory location.

The syntax is similar to other FPU arithmetic instructions, supporting both memory and register operands.

For example, **fdiv ST(0), ST(i)** divides ST(i) by ST(0), and **fdiv m64fp** divides ST(0) by the value in memory specified by m64fp.

Handling special cases is essential when dealing with division.

For instance, division by zero results in a divide-by-zero exception, so it's crucial to ensure that the divisor is not zero.

Special situations arise when **dividing by infinity**, **zero, or NaN (Not-a-Number)**, each governed by specific rules as detailed in the Intel Instruction Set Reference manual.

FIDIV (Floating-Point Divide Integer):

The FIDIV instruction is used to perform integer division within the x86 FPU.

Before the division operation, it converts the source operand (an integer) to double-extended precision floating-point format, ensuring compatibility with the FPU's floating-point arithmetic.

Syntax examples include **fidiv m16int** for dividing a 16-bit integer

and **fidiv m32int** for dividing a 32-bit integer.

The FIDIV instruction is particularly useful when you need to perform arithmetic operations involving integers and floating-point numbers in the FPU.

Now, let's provide a code that showcases various division scenarios, including division by different data types and operand combinations.

```
.data
    dblOne REAL8 1234.56
    dblTwo REAL8 10.0
    intDivisor WORD 2
    quotient REAL8 ?

.code
    ; Single-Precision Floating-Point Division
    fld dblOne          ; Load dblOne onto the FPU stack
    fdiv dblTwo         ; Divide ST(0) by dblTwo
    fstp quotient       ; Store the result in quotient

    ; Reset the FPU stack
    fld dblOne          ; Load dblOne again

    ; Double-Precision Floating-Point Division
    fdiv dblTwo         ; Divide ST(0) by dblTwo
    fstp quotient       ; Store the result in quotient

    ; Integer Division Using FIDIV
    fld intDivisor      ; Load the integer divisor onto the FPU stack
    fidiv intDivisor    ; Divide ST(0) by intDivisor
    fstp quotient       ; Store the result in quotient
```

Let's describe the provided assembly code in paragraphs to understand its functionality.

In the provided assembly code, we are performing various division operations using the x86 Floating-Point Unit (FPU).

These operations involve both floating-point and integer division. The code showcases different scenarios of division and stores the results in the quotient variable.

The code begins by defining some data elements in the .data section.

It sets up two floating-point variables, dblOne and dblTwo, with respective values.

Additionally, it defines an integer variable, intDivisor, and an empty variable named quotient, which will hold the results of the division operations.

The .code section is where the actual division operations are performed. First, it loads the value of dblOne onto the FPU stack using the fld (load) instruction.

This value represents a double-precision floating-point number.

Next, it performs single-precision floating-point division by dividing ST(0) (top of the stack) by the value of dblTwo using the fdiv (division) instruction.

The result is then stored in the quotient variable using fstp (store and pop).

The code resets the FPU stack by loading dblOne again. This time, it performs double-precision floating-point division by dividing ST(0) by dblTwo.

This demonstrates division with larger precision.

Lastly, the code performs integer division using the FIDIV (integer divide) instruction.

It loads the value of intDivisor onto the FPU stack and divides ST(0) by this integer value. The result is then stored in the quotient variable.

In summary, the code illustrates various division scenarios using the x86 FPU.

It covers single-precision and double-precision floating-point division as well as integer division, providing a comprehensive example of division operations within the FPU.

The results are stored in the quotient variable, making it a versatile demonstration of division in assembly language.