# X86 Assembly/Floating Point



While integers are sufficient for some applications, it is often necessary to use the floating point coprocessor to manipulate numbers with fractional parts.

## x87 Coprocessor

The original x86 family members had a separate math coprocessor that handled floating point arithmetic. The original coprocessor was the 8087, and all FPUs since have been dubbed "x87" chips. Later variants integrated the floating point unit (FPU) into the microprocessor itself. Having the capability to manage floating point numbers means a few things:

1. The microprocessor must have space to store floating point numbers
2. The microprocessor must have instructions to manipulate floating point numbers

The FPU, even when it is integrated into an x86 chip, is still called the "x87" section. For instance, literature on the subject will frequently call the FPU Register Stack the "x87 Stack", and the FPU operations will frequently be called the "x87 instruction set".

## FPU Register Stack

The FPU has 8 registers, st0 to st7, formed into a stack. Numbers are pushed onto the stack from memory, and are popped off the stack back to memory. FPU instructions generally will pop the first two items off the stack, act on them, and push the answer back on to the top of the stack.

Floating point numbers may generally be either 32 bits long (C "float" type), or 64 bits long (C "double" type). However, in order to reduce round-off errors, the FPU stack registers are all 80 bits wide.

Most calling conventions return floating point values in the st0 register.

## Examples

The following program (using NASM syntax) calculates the square root of 123.45.

```
global _start

section .data
    val: dq 123.45  ;declare quad word (double precision)

section .bss
    res: resq 1     ;reserve 1 quad word for result

section .text
    _start:

    fld qword [val] ;load value into st0
    fsqrt           ;compute square root of st0 and store in st0
    fst qword [res] ;store st0 in result

    ;end program
```

Essentially, programs that use the FPU load values onto the stack with `FLD` and its variants, perform operations on these values, then store them into memory with one of the forms of `FST`. Because the x87 stack can only be accessed by FPU instructions – you cannot write `mov eax, st0` – it is necessary to store values to memory if you want to print them, for example.

A more complex example that evaluates the Law of Cosines:

```
;; c^2 = a^2 + b^2 - cos(C)*2*a*b
;; C is stored in ang

global _start

section .data
    a: dq 4.56    ;length of side a
    b: dq 7.89    ;length of side b
    ang: dq 1.5   ;opposite angle to side c (around 85.94 degrees)

section .bss
    c: resq 1     ;the result — length of side c

section .text
    _start:

    fld qword [a]    ;load a into st0
    fmul st0, st0    ;st0 = a * a = a^2

    fld qword [b]    ;load b into st1
    fmul st1, st1    ;st1 = b * b = b^2

    fadd st0, st1    ;st0 = a^2 + b^2

    fld qword [ang]  ;load angle into st0
    fcos             ;st0 = cos(ang)

    fmul qword [a]   ;st0 = cos(ang) * a
    fmul qword [b]   ;st0 = cos(ang) * a * b
    fadd st0, st0    ;st0 = cos(ang) * a * b + cos(ang) * a * b = 2(cos(ang) * a * b)

    fsubp st1, st0   ;st1 = st1 - st0 = (a^2 + b^2) - (2 * a * b * cos(ang))
                     ;and pop st0

    fsqrt            ;take square root of st0 = c

    fst qword [c]    ;store st0 in c — and we're done!

    ;end program
```

# Floating-Point Instruction Set

You may notice that some of the instructions below differ from another in name by just one letter: a **P** appended to the end. This suffix signifies that in addition to performing the normal operation, they also **P**op the x87 stack after execution is complete.

### Original 8087 instructions

F2XM1, FABS, FADD, FADDP, FBLD, FBSTP, FCHS, FCLEX, FCOM, FCOMP, FCOMPP, FDECSTP, FDISI, FDIV, FDIVP, FDIVR, FDIVRP, FENI, FFREE, FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FILD, FIMUL, FINCSTP, FINIT, FIST, FISTP, FISUB, FISUBR, FLD, FLD1, FLDCW, FLDENV, FLDENVW, FLDL2E, FLDL2T, FLDLG2, FLDLN2, FLDPI, FLDZ, FMUL, FMULP, FNCLEX, FNDISI, FNENI, FNINIT, FNOP, FNSAVE, FNSAVEW, FNSTCW, FNSTENV, FNSTENVW, FNSTSW, FPATAN, FPREM, FPTAN, FRNDINT, FRSTOR, FRSTORW, FSAVE, FSAVEW, FSCALE, FSQRT, FST, FSTCW, FSTENV, FSTENVW, FSTP, FSTSW, FSUB, FSUBP, FSUBR, FSUBRP, FTST, FWAIT, FXAM, FXCH, FXTRACT, FYL2X, FYL2XP1

## Added in specific processors

### Added with 80287

FSETPM

### Added with 80387

FCOS, FLDENVD, FNSAVED, FNSTENVD, FPREM1, FRSTORD, FSAVED, FSIN, FSINCOS, FSTENVD, FUCOM, FUCOMP, FUCOMPP

### Added with Pentium Pro

FCMOVB, FCMOVBE, FCMOVE, FCMOVNB, FCMOVNBE, FCMOVNE, FCMOVNU, FCMOVU, FCOMI, FCOMIP, FUCOMI, FUCOMIP, FXRSTOR, FXSAVE

### Added with SSE

FXRSTOR, FXSAVE

These are also supported on later Pentium IIs which do not contain SSE support

### Added with SSE3

FISTTP (x87 to integer conversion with truncation regardless of status word)

## Undocumented instructions

FFREEP performs FFREE ST(i) and pop stack

# x86 Disassembly/Floating Point Numbers



## Floating Point Numbers

This page will talk about how **floating point** numbers are used in assembly language constructs. This page will not talk about new constructs, it will not explain what the FPU instructions do, how floating point numbers are stored or manipulated, or the differences in floating-point data representations. However, this page will demonstrate briefly how floating-point numbers are used in code and data structures that we have already considered.

The x86 architecture does not have any registers specifically for floating point numbers, but it does have a special stack for them. The floating point stack is built directly into the processor, and has access speeds similar to those of ordinary registers. Notice that the FPU stack is not the same as the regular system stack.

## Calling Conventions

With the addition of the floating-point stack, there is an entirely new dimension for passing parameters and returning values. We will examine our calling conventions here, and see how they are affected by the presence of floating-point numbers. These are the functions that we will be assembling, using both GCC, and cl.exe:

```
__cdecl double MyFunction1(double x, double y, float z)
{
        return (x + 1.0) * (y + 2.0) * (z + 3.0);
}

__fastcall double MyFunction2(double x, double y, float z)
{
        return (x + 1.0) * (y + 2.0) * (z + 3.0);
}

__stdcall double MyFunction3(double x, double y, float z)
{
        return (x + 1.0) * (y + 2.0) * (z + 3.0);
}
```

**cl.exe** doesn't use these directives, so to create these functions, 3 different files need to be created, compiled with the /Gd, /Gr, and /Gz options, respectively.

## CDECL

Here is the cl.exe assembly listing for MyFunction1:

```
PUBLIC  _MyFunction1
PUBLIC  __real@3ff0000000000000
PUBLIC  __real@4000000000000000
PUBLIC  __real@4008000000000000
EXTRN   __fltused:NEAR
;       COMDAT __real@3ff0000000000000
CONST   SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r     ; 1
CONST   ENDS
;       COMDAT __real@4000000000000000
CONST   SEGMENT
__real@4000000000000000 DQ 04000000000000000r     ; 2
CONST   ENDS
;       COMDAT __real@4008000000000000
CONST   SEGMENT
__real@4008000000000000 DQ 04008000000000000r     ; 3
CONST   ENDS
_TEXT   SEGMENT
_x$ = 8                                                 ; size = 8
_y$ = 16                                                ; size = 8
_z$ = 24                                                ; size = 4
_MyFunction1 PROC NEAR
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        fld     QWORD PTR _x$[ebp]
        fadd    QWORD PTR __real@3ff0000000000000
        fld     QWORD PTR _y$[ebp]
        fadd    QWORD PTR __real@4000000000000000
        fmulp   ST(1), ST(0)
        fld     DWORD PTR _z$[ebp]
        fadd    QWORD PTR __real@4008000000000000
        fmulp   ST(1), ST(0)
; Line 4
        pop     ebp
        ret     0
_MyFunction1 ENDP
_TEXT   ENDS
```

Our first question is this: are the parameters passed on the stack, or on the floating-point register stack, or some place different entirely? Key to this question, and to this function is a knowledge of what **fld** and **fstp** do. fld (Floating-point Load) pushes a floating point value onto the FPU stack, while fstp (Floating-Point Store and Pop) moves a floating point value from ST0 to the specified location, and then pops the value from ST0 off the stack entirely. Remember that **double** values in cl.exe are treated as 8-byte storage locations (QWORD), while floats are only stored as 4-byte quantities (DWORD). It is also important to remember that floating point numbers are not stored in a human-readable form in memory, even if the reader has a solid knowledge of binary. Remember, these aren't integers. Unfortunately, the exact format of floating point numbers is well beyond the scope of this chapter.

x is offset +8, y is offset +16, and z is offset +24 from ebp. Therefore, z is pushed first, x is pushed last, and the parameters are passed right-to-left on the *regular stack* not the floating point stack. To understand how a value is returned however, we need to understand what **fmulp** does. fmulp is the "Floating-Point Multiply and Pop" instruction. It performs the instructions:

```
ST1 := ST1 * ST0
FPU POP ST0
```

This multiplies ST(1) and ST(0) and stores the result in ST(1). Then, ST(0) is marked empty and stack pointer is incremented. Thus, contents of ST(1) are on the top of the stack. So the top 2 values are multiplied together, and the result is stored on the top of the stack. Therefore, in our instruction above,

"fmulp ST(1), ST(0)", which is also the last instruction of the function, we can see that the last result is stored in ST0. Therefore, floating point parameters are passed on the regular stack, but floating point results are passed on the FPU stack.

One final note is that MyFunction2 cleans its own stack, as referenced by the **ret 20** command at the end of the listing. Because none of the parameters were passed in registers, this function appears to be exactly what we would expect an STDCALL function would look like: parameters passed on the stack from right-to-left, and the function cleans its own stack. We will see below that this is actually a correct assumption.

For comparison, here is the GCC listing:

```
LC1:
        .long   0
        .long   1073741824
        .align 8
LC2:
        .long   0
        .long   1074266112
.globl _MyFunction1
        .def    _MyFunction1;   .scl    2;      .type   32;     .endef
_MyFunction1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        fldl    8(%ebp)
        fstpl   -8(%ebp)
        fldl    16(%ebp)
        fstpl   -16(%ebp)
        fldl    -8(%ebp)
        fld1
        faddp   %st, %st(1)
        fldl    -16(%ebp)
        fldl    LC1
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        flds    24(%ebp)
        fldl    LC2
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        leave
        ret
        .align 8
```

This is a very difficult listing, so we will step through it (albeit quickly). 16 bytes of extra space is allocated on the stack. Then, using a combination of fldl and fstpl instructions, the first 2 parameters are moved from offsets +8 and +16, to offsets -8 and -16 from ebp. Seems like a waste of time, but remember, optimizations are off. **fld1** loads the floating point value 1.0 onto the FPU stack. **faddp** then adds the top of the stack (1.0), to the value in ST1 ([ebp - 8], originally [ebp + 8]).

## FASTCALL

Here is the cl.exe listing for MyFunction2:

```
PUBLIC  @MyFunction2@20
PUBLIC  __real@3ff0000000000000
PUBLIC  __real@4000000000000000
PUBLIC  __real@4008000000000000
EXTRN   __fltused:NEAR
;       COMDAT __real@3ff0000000000000
CONST   SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r     ; 1
CONST   ENDS
;       COMDAT __real@4000000000000000
CONST   SEGMENT
__real@4000000000000000 DQ 04000000000000000r     ; 2
CONST   ENDS
;       COMDAT __real@4008000000000000
CONST   SEGMENT
__real@4008000000000000 DQ 04008000000000000r     ; 3
CONST   ENDS
_TEXT   SEGMENT
_x$ = 8                                                 ; size = 8
_y$ = 16                                                ; size = 8
_z$ = 24                                                ; size = 4
@MyFunction2@20 PROC NEAR
; Line 7
        push    ebp
        mov     ebp, esp
; Line 8
        fld     QWORD PTR _x$[ebp]
        fadd    QWORD PTR __real@3ff0000000000000
        fld     QWORD PTR _y$[ebp]
        fadd    QWORD PTR __real@4000000000000000
        fmulp   ST(1), ST(0)
        fld     DWORD PTR _z$[ebp]
        fadd    QWORD PTR __real@4008000000000000
        fmulp   ST(1), ST(0)
; Line 9
        pop     ebp
        ret     20                                      ; 00000014H
@MyFunction2@20 ENDP
_TEXT   ENDS
```

We can see that this function is taking 20 bytes worth of parameters, because of the @20 decoration at the end of the function name. This makes sense, because the function is taking two **double** parameters (8 bytes each), and one **float** parameter (4 bytes each). This is a grand total of 20 bytes. We can notice at a first glance, without having to actually analyze or understand any of the code, that there is only one register being accessed here: **ebp**. This seems strange, considering that FASTCALL passes its regular 32-bit arguments in registers. However, that is not the case here: all the floating-point parameters (even z, which is a 32-bit float) are passed on the stack. We know this, because by looking at the code, there is no other place where the parameters could be coming from.

Notice also that **fmulp** is the last instruction performed again, as it was in the CDECL example. We can infer then, without investigating too deeply, that the result is passed at the top of the floating-point stack.

Notice also that x (offset [ebp + 8]), y (offset [ebp + 16]) and z (offset [ebp + 24]) are pushed in reverse order: z is first, x is last. This means that floating point parameters are passed in right-to-left order, on the stack. This is exactly the same as CDECL code, although only because we are using floating-point values.

Here is the GCC assembly listing for MyFunction2:

```
        .align 8
LC5:
        .long   0
        .long   1073741824
        .align 8
LC6:
        .long   0
        .long   1074266112
.globl @MyFunction2@20
        .def    @MyFunction2@20; .scl    2;      .type   32;     .endef
@MyFunction2@20:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        fldl    8(%ebp)
        fstpl   -8(%ebp)
        fldl    16(%ebp)
        fstpl   -16(%ebp)
        fldl    -8(%ebp)
        fld1
        faddp   %st, %st(1)
        fldl    -16(%ebp)
        fldl    LC5
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        flds    24(%ebp)
        fldl    LC6
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        leave
        ret     $20
```

This is a tricky piece of code, but luckily we don't need to read it very close to find what we are looking for. First off, notice that no other registers are accessed besides **ebp**. Again, GCC passes all floating point values (even the 32-bit float, z) on the stack. Also, the floating point result value is passed on the top of the floating point stack.

We can see again that GCC is doing something strange at the beginning, taking the values on the stack from [ebp + 8] and [ebp + 16], and moving them to locations [ebp - 8] and [ebp - 16], respectively. Immediately after being moved, these values are loaded onto the floating point stack and arithmetic is performed. z isn't loaded till later, and isn't ever moved to [ebp - 24], despite the pattern.

LC5 and LC6 are constant values, that most likely represent floating point values (because the numbers themselves, 1073741824 and 1074266112 don't make any sense in the context of our example functions. Notice though that both LC5 and LC6 contain two **.long** data items, for a total of 8 bytes of storage? They are therefore most definitely **double** values.

## STDCALL

Here is the cl.exe listing for MyFunction3:

```
PUBLIC  _MyFunction3@20
PUBLIC  __real@3ff0000000000000
PUBLIC  __real@4000000000000000
PUBLIC  __real@4008000000000000
EXTRN   __fltused:NEAR
;       COMDAT __real@3ff0000000000000
CONST   SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r     ; 1
CONST   ENDS
;       COMDAT __real@4000000000000000
CONST   SEGMENT
__real@4000000000000000 DQ 04000000000000000r     ; 2
CONST   ENDS
;       COMDAT __real@4008000000000000
CONST   SEGMENT
__real@4008000000000000 DQ 04008000000000000r     ; 3
CONST   ENDS
_TEXT   SEGMENT
_x$ = 8                                           ; size = 8
_y$ = 16                                                  ; size = 8
_z$ = 24                                                  ; size = 4
_MyFunction3@20 PROC NEAR
; Line 12
        push    ebp
        mov     ebp, esp
; Line 13
        fld     QWORD PTR _x$[ebp]
        fadd    QWORD PTR __real@3ff0000000000000
        fld     QWORD PTR _y$[ebp]
        fadd    QWORD PTR __real@4000000000000000
        fmulp   ST(1), ST(0)
        fld     DWORD PTR _z$[ebp]
        fadd    QWORD PTR __real@4008000000000000
        fmulp   ST(1), ST(0)
; Line 14
        pop     ebp
        ret     20                                        ; 00000014H
_MyFunction3@20 ENDP
_TEXT   ENDS
END
```

x is the highest on the stack, and z is the lowest, therefore these parameters are passed from right-to-left. We can tell this because x has the smallest offset (offset [ebp + 8]), while z has the largest offset (offset [ebp + 24]). We see also from the final fmulp instruction that the return value is passed on the FPU stack. This function also cleans the stack itself, as noticed by the call '*ret 20*. It is cleaning exactly 20 bytes off the stack which is, incidentally, the total amount that we passed to begin with. We can also notice that the implementation of this function looks exactly like the FASTCALL version of this function. This is true because FASTCALL only passes DWORD-sized parameters in registers, and floating point numbers do not qualify. This means that our assumption above was correct.

Here is the GCC listing for MyFunction3:

```
        .align 8
LC9:
        .long   0
        .long   1073741824
        .align 8
LC10:
        .long   0
        .long   1074266112
.globl @MyFunction3@20
        .def    @MyFunction3@20; .scl    2;      .type   32;     .endef
@MyFunction3@20:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        fldl    8(%ebp)
        fstpl   -8(%ebp)
        fldl    16(%ebp)
        fstpl   -16(%ebp)
        fldl    -8(%ebp)
        fld1
        faddp   %st, %st(1)
        fldl    -16(%ebp)
        fldl    LC9
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        flds    24(%ebp)
        fldl    LC10
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        leave
        ret     $20
```

Here we can also see, after all the opening nonsense, that [ebp - 8] (originally [ebp + 8]) is value x, and that [ebp - 24] (originally [ebp - 24]) is value z. These parameters are therefore passed right-to-left. Also, we can deduce from the final fmulp instruction that the result is passed in ST0. Again, the STDCALL function cleans its own stack, as we would expect.

## Conclusions

Floating point values are passed as parameters on the stack, and are passed on the FPU stack as results. Floating point values do not get put into the general-purpose integer registers (eax, ebx, etc...), so FASTCALL functions that only have floating point parameters collapse into STDCALL functions instead. **double** values are 8-bytes wide, and therefore will take up 8-bytes on the stack. **float** values however, are only 4-bytes wide.