UMBC **CMSC 313 -- NASM Floating Point Documentation**

The NASM manual provides the following information about floating point:

2.2.6 Floating-Point Differences

NASM uses different names to refer to floating-point registers from MASM: where MASM would call them ST(0), ST(1) and so on, and a86 would call them simply 0, 1 and so on, NASM chooses to call them st0, st1 etc.

As of version 0.96, NASM now treats the instructions with `nowait' forms in the same way as MASM-compatible assemblers. The idiosyncratic treatment employed by 0.95 and earlier was based on a misunderstanding by the authors.

3.4.4 Floating-Point Constants

Floating-point constants are acceptable only as arguments to DD, DQ and DT. They are expressed in the traditional form: digits, then a period, then optionally more digits, then optionally an E followed by an exponent. The period is mandatory, so that NASM can distinguish between dd 1, which declares an integer constant, and dd 1.0 which declares a floating-point constant.

Some examples:

```
dd    1.2                    ; an easy one
dq    1.e10                  ; 10,000,000,000
dq    1.e+10                 ; synonymous with 1.e10
dq    1.e-10                 ; 0.000 000 000 1
dt    3.14159265358979323846 ; pi
```

NASM cannot do compile-time arithmetic on floating-point constants. This is because NASM is designed to be portable - although it always generates code to run on x86 processors, the assembler itself can run on any system with an ANSI C compiler. Therefore, the assembler cannot guarantee the presence of a floating-point unit capable of handling the Intel number formats, and so for NASM to be able to do floating arithmetic it would have to include its own complete set of floating-point routines, which would significantly increase the size of the assembler for very little benefit.

B.4.66 F2XM1: Calculate 2**X-1

```
F2XM1                          ; D9 F0              [8086,FPU]
```

F2XM1 raises 2 to the power of ST0, subtracts one, and stores the result back into ST0. The initial contents of ST0 must be a number in the range -1.0 to +1.0.

B.4.67 FABS: Floating-Point Absolute Value

```
FABS                           ; D9 E1              [8086,FPU]
```

FABS computes the absolute value of ST0,by clearing the sign bit, and stores the result back in ST0.

B.4.68 FADD, FADDP: Floating-Point Addition

```
FADD mem32                      ; D8 /0             [8086,FPU]
FADD mem64                      ; DC /0             [8086,FPU]

FADD fpureg                     ; D8 C0+r           [8086,FPU]
FADD ST0,fpureg                 ; D8 C0+r           [8086,FPU]

FADD TO fpureg                  ; DC C0+r           [8086,FPU]
FADD fpureg,ST0                 ; DC C0+r           [8086,FPU]

FADDP fpureg                    ; DE C0+r           [8086,FPU]
FADDP fpureg,ST0                ; DE C0+r           [8086,FPU]
```

- FADD, given one operand, adds the operand to ST0 and stores the result back in ST0. If the operand has the TO modifier, the result is stored in the register given rather than in ST0.
- FADDP performs the same function as FADD TO, but pops the register stack after storing the result.

The given two-operand forms are synonyms for the one-operand forms.

To add an integer value to ST0, use the c{FIADD} instruction (section B.4.80)

B.4.69 FBLD, FBSTP: BCD Floating-Point Load and Store

```
FBLD mem80                      ; DF /4             [8086,FPU]
FBSTP mem80                     ; DF /6             [8086,FPU]
```

FBLD loads an 80-bit (ten-byte) packed binary-coded decimal number from the given memory address, converts it to a real, and pushes it on the register stack. FBSTP stores the value of ST0, in packed BCD, at the given address and then pops the register stack.

B.4.70 FCHS: Floating-Point Change Sign

```
FCHS                            ; D9 E0             [8086,FPU]
```

FCHS negates the number in ST0, by inverting the sign bit: negative numbers become positive, and vice versa.

B.4.71 FCLEX, FNCLEX: Clear Floating-Point Exceptions

```
FCLEX                           ; 9B DB E2          [8086,FPU]
FNCLEX                          ; DB E2             [8086,FPU]
```

FCLEX clears any floating-point exceptions which may be pending. FNCLEX does the same thing but doesn't wait for previous floating-point operations (including the handling of pending exceptions) to finish first.

B.4.72 FCMOVcc: Floating-Point Conditional Move

```
FCMOVB fpureg               ; DA C0+r           [P6,FPU]
FCMOVB ST0,fpureg           ; DA C0+r           [P6,FPU]

FCMOVE fpureg               ; DA C8+r           [P6,FPU]
FCMOVE ST0,fpureg           ; DA C8+r           [P6,FPU]

FCMOVBE fpureg              ; DA D0+r           [P6,FPU]
FCMOVBE ST0,fpureg          ; DA D0+r           [P6,FPU]

FCMOVU fpureg               ; DA D8+r           [P6,FPU]
FCMOVU ST0,fpureg           ; DA D8+r           [P6,FPU]

FCMOVNB fpureg              ; DB C0+r           [P6,FPU]
FCMOVNB ST0,fpureg          ; DB C0+r           [P6,FPU]

FCMOVNE fpureg              ; DB C8+r           [P6,FPU]
FCMOVNE ST0,fpureg          ; DB C8+r           [P6,FPU]

FCMOVNBE fpureg             ; DB D0+r           [P6,FPU]
FCMOVNBE ST0,fpureg         ; DB D0+r           [P6,FPU]

FCMOVNU fpureg              ; DB D8+r           [P6,FPU]
FCMOVNU ST0,fpureg          ; DB D8+r           [P6,FPU]
```

The FCMOV instructions perform conditional move operations: each of them moves the contents of the given register into ST0 if its condition is satisfied, and does nothing if not.

The conditions are not the same as the standard condition codes used with conditional jump instructions. The conditions B, BE, NB, NBE, E and NE are exactly as normal, but none of the other standard ones are supported. Instead, the condition U and its counterpart NU are provided; the U condition is satisfied if the last two floating-point numbers compared were unordered, i.e. they were not equal but neither one could be said to be greater than the other, for example if they were NaNs. (The flag state which signals this is the setting of the parity flag: so the U condition is notionally equivalent to PE, and NU is equivalent to PO.)

The FCMOV conditions test the main processor's status flags, not the FPU status flags, so using FCMOV directly after FCOM will not work. Instead, you should either use FCOMI which writes directly to the main CPU flags word, or use FSTSW to extract the FPU flags.

Although the FCMOV instructions are flagged P6 above, they may not be supported by all Pentium Pro processors; the CPUID instruction (section B.4.34) will return a bit which indicates whether conditional moves are supported.

B.4.73 FCOM, FCOMP, FCOMPP, FCOMI, FCOMIP: Floating-Point Compare

```
FCOM mem32                      ; D8 /2             [8086,FPU]
FCOM mem64                      ; DC /2             [8086,FPU]
FCOM fpureg                     ; D8 D0+r           [8086,FPU]
FCOM ST0,fpureg                 ; D8 D0+r           [8086,FPU]

FCOMP mem32                     ; D8 /3             [8086,FPU]
FCOMP mem64                     ; DC /3             [8086,FPU]
FCOMP fpureg                    ; D8 D8+r           [8086,FPU]
FCOMP ST0,fpureg                ; D8 D8+r           [8086,FPU]

FCOMPP                          ; DE D9             [8086,FPU]

FCOMI fpureg                    ; DB F0+r           [P6,FPU]
FCOMI ST0,fpureg                ; DB F0+r           [P6,FPU]

FCOMIP fpureg                   ; DF F0+r           [P6,FPU]
FCOMIP ST0,fpureg               ; DF F0+r           [P6,FPU]
```

FCOM compares ST0 with the given operand, and sets the FPU flags accordingly. ST0 is treated as the left-hand side of the comparison, so that the carry flag is set (for a `less-than' result) if ST0 is less than the given operand.

FCOMP does the same as FCOM, but pops the register stack afterwards. FCOMPP compares ST0 with ST1 and then pops the register stack twice.

FCOMI and FCOMIP work like the corresponding forms of FCOM and FCOMP, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.

The FCOM instructions differ from the FUCOM instructions (section B.4.108) only in the way they handle quiet NaNs: FUCOM will handle them silently and set the condition code flags to an `unordered' result, whereas FCOM will generate an exception.

B.4.74 FCOS: Cosine

```
FCOS                            ; D9 FF             [386,FPU]
```

FCOS computes the cosine of ST0 (in radians), and stores the result in ST0. The absolute value of ST0 must be less than 2**63.

See also FSINCOS (section B.4.100).

B.4.75 FDECSTP: Decrement Floating-Point Stack Pointer

```
FDECSTP                         ; D9 F6             [8086,FPU]
```

FDECSTP decrements the `top' field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the contents of ST7 had been pushed on the stack. See also FINCSTP (section B.4.85).

B.4.76 FxDISI, FxENI: Disable and Enable Floating-Point Interrupts

```
FDISI                   ; 9B DB E1          [8086,FPU]
FNDISI                  ; DB E1             [8086,FPU]


FENI                    ; 9B DB E0          [8086,FPU]
FNENI                   ; DB E0             [8086,FPU]
```

FDISI and FENI disable and enable floating-point interrupts. These instructions are only meaningful on original 8087 processors: the 287 and above treat them as no-operation instructions.

FNDISI and FNENI do the same thing as FDISI and FENI respectively, but without waiting for the floating-point processor to finish what it was doing first.

B.4.77 FDIV, FDIVP, FDIVR, FDIVRP: Floating-Point Division

```
FDIV mem32              ; D8 /6             [8086,FPU]
FDIV mem64              ; DC /6             [8086,FPU]


FDIV fpureg             ; D8 F0+r           [8086,FPU]
FDIV ST0,fpureg         ; D8 F0+r           [8086,FPU]


FDIV TO fpureg          ; DC F8+r           [8086,FPU]
FDIV fpureg,ST0         ; DC F8+r           [8086,FPU]


FDIVR mem32             ; D8 /0             [8086,FPU]
FDIVR mem64             ; DC /0             [8086,FPU]


FDIVR fpureg            ; D8 F8+r           [8086,FPU]
FDIVR ST0,fpureg        ; D8 F8+r           [8086,FPU]


FDIVR TO fpureg         ; DC F0+r           [8086,FPU]
FDIVR fpureg,ST0        ; DC F0+r           [8086,FPU]


FDIVP fpureg            ; DE F8+r           [8086,FPU]
FDIVP fpureg,ST0        ; DE F8+r           [8086,FPU]


FDIVRP fpureg           ; DE F0+r           [8086,FPU]
FDIVRP fpureg,ST0       ; DE F0+r           [8086,FPU]
```

- FDIV divides ST0 by the given operand and stores the result back in ST0, unless the TO qualifier is given, in which case it divides the given operand by ST0 and stores the result in the operand.
- FDIVR does the same thing, but does the division the other way up: so if TO is not given, it divides the given operand by ST0 and stores the result in ST0, whereas if TO is given it divides ST0 by its operand and stores the result in the operand.
- FDIVP operates like FDIV TO, but pops the register stack once it has finished.
- FDIVRP operates like FDIVR TO, but pops the register stack once it has finished.

For FP/Integer divisions, see FIDIV (section B.4.82).

B.4.78 FEMMS: Faster Enter/Exit of the MMX or floating-point state

```
FEMMS                   ; 0F 0E         [PENT,3DNOW]
```

FEMMS can be used in place of the EMMS instruction on processors which support the 3DNow! instruction set. Following execution of FEMMS, the state of the MMX/FP registers is undefined, and this allows a faster context switch between FP and MMX instructions. The FEMMS instruction can also be used before executing MMX instructions

B.4.79 FFREE: Flag Floating-Point Register as Unused

```
FFREE fpureg            ; DD C0+r           [8086,FPU]
FFREEP fpureg           ; DF C0+r           [286,FPU,UNDOC]
```

FFREE marks the given register as being empty.

FFREEP marks the given register as being empty, and then pops the register stack.

B.4.80 FIADD: Floating-Point/Integer Addition

```
FIADD mem16             ; DE /0             [8086,FPU]
FIADD mem32             ; DA /0             [8086,FPU]
```

FIADD adds the 16-bit or 32-bit integer stored in the given memory location to ST0, storing the result in ST0.

B.4.81 FICOM, FICOMP: Floating-Point/Integer Compare

```
FICOM mem16             ; DE /2             [8086,FPU]
FICOM mem32             ; DA /2             [8086,FPU]

FICOMP mem16            ; DE /3             [8086,FPU]
FICOMP mem32            ; DA /3             [8086,FPU]
```

FICOM compares ST0 with the 16-bit or 32-bit integer stored in the given memory location, and sets the FPU flags accordingly. FICOMP does the same, but pops the register stack afterwards.

B.4.82 FIDIV, FIDIVR: Floating-Point/Integer Division

```
FIDIV mem16             ; DE /6             [8086,FPU]
FIDIV mem32             ; DA /6             [8086,FPU]

FIDIVR mem16            ; DE /7             [8086,FPU]
FIDIVR mem32            ; DA /7             [8086,FPU]
```

FIDIV divides ST0 by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in ST0. FIDIVR does the division the other way up: it divides the integer by ST0, but still stores the result in ST0.

B.4.83 FILD, FIST, FISTP: Floating-Point/Integer Conversion

```
FILD mem16                  ; DF /0             [8086,FPU]
FILD mem32                  ; DB /0             [8086,FPU]
FILD mem64                  ; DF /5             [8086,FPU]

FIST mem16                  ; DF /2             [8086,FPU]
FIST mem32                  ; DB /2             [8086,FPU]

FISTP mem16                 ; DF /3             [8086,FPU]
FISTP mem32                 ; DB /3             [8086,FPU]
FISTP mem64                 ; DF /7             [8086,FPU]
```

FILD loads an integer out of a memory location, converts it to a real, and pushes it on the FPU register stack. FIST converts ST0 to an integer and stores that in memory; FISTP does the same as FIST, but pops the register stack afterwards.

B.4.84 FIMUL: Floating-Point/Integer Multiplication

```
FIMUL mem16                 ; DE /1             [8086,FPU]
FIMUL mem32                 ; DA /1             [8086,FPU]
```

FIMUL multiplies ST0 by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in ST0.

B.4.85 FINCSTP: Increment Floating-Point Stack Pointer

```
FINCSTP                     ; D9 F7             [8086,FPU]
```

FINCSTP increments the `top' field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the register stack had been popped; however, unlike the popping of the stack performed by many FPU instructions, it does not flag the new ST7 (previously ST0) as empty. See also FDECSTP (section B.4.75).

B.4.86 FINIT, FNINIT: Initialise Floating-Point Unit

```
FINIT                       ; 9B DB E3          [8086,FPU]
FNINIT                      ; DB E3             [8086,FPU]
```

FINIT initialises the FPU to its default state. It flags all registers as empty, without actually change their values, clears the top of stack pointer. FNINIT does the same, without first waiting for pending exceptions to clear.

B.4.87 FISUB: Floating-Point/Integer Subtraction

```
FISUB mem16                     ; DE /4             [8086,FPU]
FISUB mem32                     ; DA /4             [8086,FPU]

FISUBR mem16                    ; DE /5             [8086,FPU]
FISUBR mem32                    ; DA /5             [8086,FPU]
```

FISUB subtracts the 16-bit or 32-bit integer stored in the given memory location from ST0, and stores the result in ST0. FISUBR does the subtraction the other way round, i.e. it subtracts ST0 from the given integer, but still stores the result in ST0.

B.4.88 FLD: Floating-Point Load

```
FLD mem32                       ; D9 /0             [8086,FPU]
FLD mem64                       ; DD /0             [8086,FPU]
FLD mem80                       ; DB /5             [8086,FPU]
FLD fpureg                      ; D9 C0+r           [8086,FPU]
```

FLD loads a floating-point value out of the given register or memory location, and pushes it on the FPU register stack.

B.4.89 FLDxx: Floating-Point Load Constant

s

```
FLD1                            ; D9 E8             [8086,FPU]
FLDL2E                          ; D9 EA             [8086,FPU]
FLDL2T                          ; D9 E9             [8086,FPU]
FLDLG2                          ; D9 EC             [8086,FPU]
FLDLN2                          ; D9 ED             [8086,FPU]
FLDPI                           ; D9 EB             [8086,FPU]
FLDZ                            ; D9 EE             [8086,FPU]
```

These instructions push specific standard constants on the FPU register stack.

Instruction	Constant pushed
FLD1	1
FLDL2E	base-2 logarithm of e
FLDL2T	base-2 log of 10
FLDLG2	base-10 log of 2
FLDLN2	base-e log of 2
FLDPI	pi
FLDZ	zero

B.4.90 FLDCW: Load Floating-Point Control Word

```
FLDCW mem16                     ; D9 /5             [8086,FPU]
```

FLDCW loads a 16-bit value out of memory and stores it into the FPU control word (governing things like the rounding mode, the precision, and the exception masks). See also FSTCW (section B.4.103). If exceptions are enabled and you don't want to generate one, use FCLEX or FNCLEX (section B.4.71) before loading the new control word.

B.4.91 FLDENV: Load Floating-Point Environment

```
FLDENV mem                      ; D9 /4             [8086,FPU]
```

FLDENV loads the FPU operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) from memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also FSTENV (section B.4.104).

B.4.92 FMUL, FMULP: Floating-Point Multiply

```
FMUL mem32                      ; D8 /1             [8086,FPU]
FMUL mem64                      ; DC /1             [8086,FPU]

FMUL fpureg                     ; D8 C8+r           [8086,FPU]
FMUL ST0,fpureg                 ; D8 C8+r           [8086,FPU]

FMUL TO fpureg                  ; DC C8+r           [8086,FPU]
FMUL fpureg,ST0                 ; DC C8+r           [8086,FPU]

FMULP fpureg                    ; DE C8+r           [8086,FPU]
FMULP fpureg,ST0                ; DE C8+r           [8086,FPU]
```

FMUL multiplies ST0 by the given operand, and stores the result in ST0, unless the TO qualifier is used in which case it stores the result in the operand. FMULP performs the same operation as FMUL TO, and then pops the register stack.

B.4.93 FNOP: Floating-Point No Operation

```
FNOP                            ; D9 D0             [8086,FPU]
```

FNOP does nothing.

B.4.94 FPATAN, FPTAN: Arctangent and Tangent

```
FPATAN                          ; D9 F3             [8086,FPU]
FPTAN                           ; D9 F2             [8086,FPU]
```

FPATAN computes the arctangent, in radians, of the result of dividing ST1 by ST0, stores the result in ST1, and pops the register stack. It works like the C atan2 function, in that changing the sign of both ST0 and ST1 changes the output value by pi (so it performs true rectangular-to-polar coordinate conversion, with ST1 being the Y coordinate and ST0 being the X coordinate, not merely an arctangent).

FPTAN computes the tangent of the value in ST0 (in radians), and stores the result back into ST0.

The absolute value of ST0 must be less than 2**63.

B.4.95 FPREM, FPREM1: Floating-Point Partial Remainder

```
FPREM                   ; D9 F8             [8086,FPU]
FPREM1                  ; D9 F5             [386,FPU]
```

These instructions both produce the remainder obtained by dividing ST0 by ST1. This is calculated, notionally, by dividing ST0 by ST1, rounding the result to an integer, multiplying by ST1 again, and computing the value which would need to be added back on to the result to get back to the original value in ST0.

The two instructions differ in the way the notional round-to-integer operation is performed. FPREM does it by rounding towards zero, so that the remainder it returns always has the same sign as the original value in ST0; FPREM1 does it by rounding to the nearest integer, so that the remainder always has at most half the magnitude of ST1.

Both instructions calculate partial remainders, meaning that they may not manage to provide the final result, but might leave intermediate results in ST0 instead. If this happens, they will set the C2 flag in the FPU status word; therefore, to calculate a remainder, you should repeatedly execute FPREM or FPREM1 until C2 becomes clear.

B.4.96 FRNDINT: Floating-Point Round to Integer

```
FRNDINT                 ; D9 FC             [8086,FPU]
```

FRNDINT rounds the contents of ST0 to an integer, according to the current rounding mode set in the FPU control word, and stores the result back in ST0.

B.4.97 FSAVE, FRSTOR: Save/Restore Floating-Point State

```
FSAVE mem               ; 9B DD /6          [8086,FPU]
FNSAVE mem              ; DD /6             [8086,FPU]

FRSTOR mem              ; DD /4             [8086,FPU]
```

FSAVE saves the entire floating-point unit state, including all the information saved by FSTENV (section B.4.104) plus the contents of all the registers, to a 94 or 108 byte area of memory (depending on the CPU mode). FRSTOR restores the floating-point state from the same area of memory.

FNSAVE does the same as FSAVE, without first waiting for pending floating-point exceptions to clear.

B.4.98 FSCALE: Scale Floating-Point Value by Power of Two

```
FSCALE                  ; D9 FD             [8086,FPU]
```

FSCALE scales a number by a power of two: it rounds ST1 towards zero to obtain an integer, then multiplies ST0 by two to the power of that integer, and stores the result in ST0.

B.4.99 FSETPM: Set Protected Mode

```
FSETPM                  ; DB E4             [286,FPU]
```

This instruction initialises protected mode on the 287 floating-point coprocessor. It is only meaningful on that processor: the 387 and above treat the instruction as a no-operation.

B.4.100 FSIN, FSINCOS: Sine and Cosine

```
FSIN                            ; D9 FE                 [386,FPU]
FSINCOS                         ; D9 FB                 [386,FPU]
```

FSIN calculates the sine of ST0 (in radians) and stores the result in ST0. FSINCOS does the same, but then pushes the cosine of the same value on the register stack, so that the sine ends up in ST1 and the cosine in ST0. FSINCOS is faster than executing FSIN and FCOS (see section B.4.74) in succession.

The absolute value of ST0 must be less than 2**63.

B.4.101 FSQRT: Floating-Point Square Root

```
FSQRT                           ; D9 FA                 [8086,FPU]
```

FSQRT calculates the square root of ST0 and stores the result in ST0.

B.4.102 FST, FSTP: Floating-Point Store

```
FST mem32                       ; D9 /2                 [8086,FPU]
FST mem64                       ; DD /2                 [8086,FPU]
FST fpureg                      ; DD D0+r               [8086,FPU]

FSTP mem32                      ; D9 /3                 [8086,FPU]
FSTP mem64                      ; DD /3                 [8086,FPU]
FSTP mem80                      ; DB /7                 [8086,FPU]
FSTP fpureg                     ; DD D8+r               [8086,FPU]
```

FST stores the value in ST0 into the given memory location or other FPU register. FSTP does the same, but then pops the register stack.

B.4.103 FSTCW: Store Floating-Point Control Word

```
FSTCW mem16                     ; 9B D9 /7              [8086,FPU]
FNSTCW mem16                    ; D9 /7                 [8086,FPU]
```

FSTCW stores the FPU control word (governing things like the rounding mode, the precision, and the exception masks) into a 2-byte memory area. See also FLDCW (section B.4.90).

FNSTCW does the same thing as FSTCW, without first waiting for pending floating-point exceptions to clear.

B.4.104 FSTENV: Store Floating-Point Environment

```
FSTENV mem                      ; 9B D9 /6              [8086,FPU]
FNSTENV mem                     ; D9 /6                 [8086,FPU]
```

FSTENV stores the FPU operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) into memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also FLDENV (section B.4.91).

FNSTENV does the same thing as FSTENV, without first waiting for pending floating-point exceptions to clear.

B.4.105 FSTSW: Store Floating-Point Status Word

```
FSTSW mem16                 ; 9B DD /7          [8086,FPU]
FSTSW AX                    ; 9B DF E0          [286,FPU]


FNSTSW mem16                ; DD /7             [8086,FPU]
FNSTSW AX                   ; DF E0             [286,FPU]
```

FSTSW stores the FPU status word into AX or into a 2-byte memory area.

FNSTSW does the same thing as FSTSW, without first waiting for pending floating-point exceptions to clear.

B.4.106 FSUB, FSUBP, FSUBR, FSUBRP: Floating-Point Subtract

```
FSUB mem32                  ; D8 /4             [8086,FPU]
FSUB mem64                  ; DC /4             [8086,FPU]


FSUB fpureg                 ; D8 E0+r           [8086,FPU]
FSUB ST0,fpureg             ; D8 E0+r           [8086,FPU]


FSUB TO fpureg              ; DC E8+r           [8086,FPU]
FSUB fpureg,ST0             ; DC E8+r           [8086,FPU]


FSUBR mem32                 ; D8 /5             [8086,FPU]
FSUBR mem64                 ; DC /5             [8086,FPU]


FSUBR fpureg                ; D8 E8+r           [8086,FPU]
FSUBR ST0,fpureg            ; D8 E8+r           [8086,FPU]


FSUBR TO fpureg             ; DC E0+r           [8086,FPU]
FSUBR fpureg,ST0            ; DC E0+r           [8086,FPU]


FSUBP fpureg                ; DE E8+r           [8086,FPU]
FSUBP fpureg,ST0            ; DE E8+r           [8086,FPU]


FSUBRP fpureg               ; DE E0+r           [8086,FPU]
FSUBRP fpureg,ST0           ; DE E0+r           [8086,FPU]
```

- FSUB subtracts the given operand from ST0 and stores the result back in ST0, unless the TO

qualifier is given, in which case it subtracts ST0 from the given operand and stores the result in the operand.
- FSUBR does the same thing, but does the subtraction the other way up: so if TO is not given, it subtracts ST0 from the given operand and stores the result in ST0, whereas if TO is given it subtracts its operand from ST0 and stores the result in the operand.
- FSUBP operates like FSUB TO, but pops the register stack once it has finished.
- FSUBRP operates like FSUBR TO, but pops the register stack once it has finished.

B.4.107 FTST: Test ST0 Against Zero

```
FTST                            ; D9 E4             [8086,FPU]
```

FTST compares ST0 with zero and sets the FPU flags accordingly. ST0 is treated as the left-hand side of the comparison, so that a `less-than' result is generated if ST0 is negative.

B.4.108 FUCOMxx: Floating-Point Unordered Compare

```
FUCOM fpureg                    ; DD E0+r           [386,FPU]
FUCOM ST0,fpureg                ; DD E0+r           [386,FPU]

FUCOMP fpureg                   ; DD E8+r           [386,FPU]
FUCOMP ST0,fpureg               ; DD E8+r           [386,FPU]

FUCOMPP                         ; DA E9             [386,FPU]

FUCOMI fpureg                   ; DB E8+r           [P6,FPU]
FUCOMI ST0,fpureg               ; DB E8+r           [P6,FPU]

FUCOMIP fpureg                  ; DF E8+r           [P6,FPU]
FUCOMIP ST0,fpureg              ; DF E8+r           [P6,FPU]
```

- FUCOM compares ST0 with the given operand, and sets the FPU flags accordingly. ST0 is treated as the left-hand side of the comparison, so that the carry flag is set (for a `less-than' result) if ST0 is less than the given operand.
- FUCOMP does the same as FUCOM, but pops the register stack afterwards. FUCOMPP compares ST0 with ST1 and then pops the register stack twice.
- FUCOMI and FUCOMIP work like the corresponding forms of FUCOM and FUCOMP, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.

The FUCOM instructions differ from the FCOM instructions (section B.4.73) only in the way they handle quiet NaNs: FUCOM will handle them silently and set the condition code flags to an `unordered' result, whereas FCOM will generate an exception.

B.4.109 FXAM: Examine Class of Value in ST0

```
FXAM                            ; D9 E5             [8086,FPU]
```

FXAM sets the FPU flags C3, C2 and C0 depending on the type of value stored in ST0:

Register contents	Flags
Unsupported format	000
NaN	001
Finite number	010
Infinity	011
Zero	100
Empty register	101
Denormal	110

Additionally, the C1 flag is set to the sign of the number.

B.4.110 FXCH: Floating-Point Exchange

```
FXCH                        ; D9 C9             [8086,FPU]
FXCH fpureg                 ; D9 C8+r           [8086,FPU]
FXCH fpureg,ST0             ; D9 C8+r           [8086,FPU]
FXCH ST0,fpureg             ; D9 C8+r           [8086,FPU]
```

FXCH exchanges ST0 with a given FPU register. The no-operand form exchanges ST0 with ST1.

B.4.111 FXRSTOR: Restore FP, MMX and SSE State

```
FXRSTOR memory              ; 0F AE /1            [P6,SSE,FPU]
```

The FXRSTOR instruction reloads the FPU, MMX and SSE state (environment and registers), from the 512 byte memory area defined by the source operand. This data should have been written by a previous FXSAVE.

B.4.112 FXSAVE: Store FP, MMX and SSE State

```
FXSAVE memory               ; 0F AE /0        [P6,SSE,FPU]
```

FXSAVEThe FXSAVE instruction writes the current FPU, MMX and SSE technology states (environment and registers), to the 512 byte memory area defined by the destination operand. It does this without checking for pending unmasked floating-point exceptions (similar to the operation of FNSAVE).

Unlike the FSAVE/FNSAVE instructions, the processor retains the contents of the FPU, MMX and SSE state in the processor after the state has been saved. This instruction has been optimised to maximize floating-point save performance.

B.4.113 FXTRACT: Extract Exponent and Significand

```
FXTRACT                     ; D9 F4             [8086,FPU]
```

FXTRACT separates the number in ST0 into its exponent and significand (mantissa), stores the exponent back into ST0, and then pushes the significand on the register stack (so that the significand ends up in ST0, and the exponent in ST1).

B.4.114 FYL2X, FYL2XP1: Compute Y times Log2(X) or Log2(X+1)

```
FYL2X                           ; D9 F1                 [8086,FPU]
FYL2XP1                         ; D9 F9                 [8086,FPU]
```

FYL2X multiplies ST1 by the base-2 logarithm of ST0, stores the result in ST1, and pops the register stack (so that the result ends up in ST0). ST0 must be non-zero and positive.

FYL2XP1 works the same way, but replacing the base-2 log of ST0 with that of ST0 plus one. This time, ST0 must have magnitude no greater than 1 minus half the square root of two.
