Real Arithmetic

Computer Organization and Assembly Languages
Yung-Yu Chuang
Fractional binary numbers

- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: \[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
Binary real numbers

- Binary real to decimal real

\[ 110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375 \]

- Decimal real to binary real

\[
\begin{align*}
0.5625 \times 2 & = 1.125 & \text{first bit} & = 1 \\
0.125 \times 2 & = 0.25 & \text{second bit} & = 0 \\
0.25 \times 2 & = 0.5 & \text{third bit} & = 0 \\
0.5 \times 2 & = 1.0 & \text{fourth bit} & = 1
\end{align*}
\]

\[ 4.5625 = 100.1001_2 \]
# Fractional binary numbers examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-3/4</td>
<td>$101.11_2$</td>
</tr>
<tr>
<td>2-7/8</td>
<td>$10.111_2$</td>
</tr>
<tr>
<td>63/64</td>
<td>$0.1111111_2$</td>
</tr>
<tr>
<td>1/3</td>
<td>$0.010101010101[01]..._2$</td>
</tr>
<tr>
<td>1/5</td>
<td>$0.001100110011[0011]..._2$</td>
</tr>
<tr>
<td>1/10</td>
<td>$0.0001100110011[0011]..._2$</td>
</tr>
</tbody>
</table>
Fixed-point numbers

<table>
<thead>
<tr>
<th>sign</th>
<th>integer part</th>
<th>fractional part</th>
</tr>
</thead>
</table>

radix point

0 000 0000 0000 0110 0110 0000 0000 0000 = 110.011

- only $2^{16}$ to $2^{-16}$
  - Not flexible, not adaptive to applications
- Fast computation, just integer operations.
  - It is often a good way to speed up in this way
  - If you know the working range beforehand.
IEEE floating point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs
- Driven by Numerical Concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make go fast
    - Numerical analysts predominated over hardware types in defining standard
IEEE floating point format

- IEEE defines two formats with different precisions: single and double

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

s  sign bit - 0 = positive, 1 = negative

\[ e = \text{biased exponent (8-bits)} = \text{true exponent} + 7F (127 \text{ decimal}) \]

The values 00 and FF have special meaning (see text).

f  fraction - the first 23-bits after the 1. in the significand.

\[ 23.85 = 10111.110110_2 = 1.0111110110 \times 2^4 \]

\[ e = 127 + 4 = 83h \]
IEEE floating point format

\[ e = 0 \text{ and } f = 0 \] denotes the number zero (which cannot be normalized). Note that there is a +0 and -0.

\[ e = 0 \text{ and } f \neq 0 \] denotes a denormalized number. These are discussed in the next section.

\[ e = \text{FF} \text{ and } f = 0 \] denotes infinity (\(\infty\)). There are both positive and negative infinities.

\[ e = \text{FF} \text{ and } f \neq 0 \] denotes an undefined result, known as NaN (Not a Number).

### Special Values

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IEEE double precision
Denormalized numbers

- Number smaller than $1.0 \times 2^{-126}$ can’t be presented by a single with normalized form. However, we can represent it with denormalized format.
- $1.0000\ldots00 \times 2^{-126}$ the least “normalized” number
- $0.1111\ldots11 \times 2^{-126}$ the largest “denormalized” number
- $1.001 \times 2^{-129} = 0.001001 \times 2^{-126}$

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Summary of Real Number Encodings

(3.14+1e20)−1e20=0
3.14+(1e20−1e20)=3.14
IA-32 floating point architecture

- Original 8086 only has integers. It is possible to simulate real arithmetic using software, but it is slow.
- 8087 floating-point processor (and 80287, 80387) was sold separately at early time.
- Since 80486, FPU (floating-point unit) was integrated into CPU.
FPU data types

- Three floating-point types
FPU data types

- Four integer types

- Word Integer
  - Sign
  - 15 14 0

- Doubleword Integer
  - Sign
  - 31 30 0

- Quadword Integer
  - Sign
  - 63 62 0

- Packed BCD Integers
  - X D17 D16 D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
  - 79 78 72 71
  - 4 Bits = 1 BCD Digit
FPU registers

- Data register
- Control register
- Status register
- Tag register
Data registers

- Load: push, TOP--
- Store: pop, TOP++
- Instructions access the stack using \( \text{ST}(i) \) relative to TOP
- If TOP=0 and push, TOP wraps to R7
- If TOP=7 and pop, TOP wraps to R0
- When overwriting occurs, generate an exception
- Real values are transferred to and from memory and stored in 10-byte temporary format. When storing, convert back to integer, long, real, long real.
Postfix expression

- \((5*6) - 4 \rightarrow 5 \ 6 \ * \ 4 \ -\)
Special-purpose registers

<table>
<thead>
<tr>
<th>15</th>
<th>0</th>
<th>47</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Register</td>
<td>Last Instruction Pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Register</td>
<td>Last Data (Operand) Pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag Register</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TAG Values
- 00 — Valid
- 01 — Zero
- 10 — Special: invalid (NaN, unsupported), infinity, or denormal
- 11 — Empty
Special-purpose registers

- Last data pointer stores the memory address of the operand for the last non-control instruction. Last instruction pointer stored the address of the last non-control instruction. Both are 48 bits, 32 for offset, 16 for segment selector.

```
11 10 11
```

```
10 8 7
```

x87 FPU Opcode Register
The instruction `FINIT` will initialize it to 037Fh.
Rounding

• FPU attempts to round an infinitely accurate result from a floating-point calculation
  - Round to nearest even: round toward to the closest one; if both are equally close, round to the even one
  - Round down: round toward to \(-\infty\)
  - Round up: round toward to \(+\infty\)
  - Truncate: round toward to zero

• Example
  - suppose 3 fractional bits can be stored, and a calculated value equals \(+1.0111\).
  - rounding up by adding \(0.0001\) produces \(1.100\)
  - rounding down by subtracting \(0.0001\) produces \(1.011\)
## Rounding

<table>
<thead>
<tr>
<th>method</th>
<th>original value</th>
<th>rounded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to nearest even</td>
<td>1.0111</td>
<td>1.100</td>
</tr>
<tr>
<td>Round down</td>
<td>1.0111</td>
<td>1.011</td>
</tr>
<tr>
<td>Round up</td>
<td>1.0111</td>
<td>1.100</td>
</tr>
<tr>
<td>Truncate</td>
<td>1.0111</td>
<td>1.011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>method</th>
<th>original value</th>
<th>rounded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to nearest even</td>
<td>-1.0111</td>
<td>-1.100</td>
</tr>
<tr>
<td>Round down</td>
<td>-1.0111</td>
<td>-1.100</td>
</tr>
<tr>
<td>Round up</td>
<td>-1.0111</td>
<td>-1.011</td>
</tr>
<tr>
<td>Truncate</td>
<td>-1.0111</td>
<td>-1.011</td>
</tr>
</tbody>
</table>
Floating-Point Exceptions

- Six types of exception conditions
  - #I: Invalid operation
  - #Z: Divide by zero
  - #D: Denormalized operand
  - #O: Numeric overflow
  - #U: Numeric underflow
  - #P: Inexact precision
  - detect before execution
  - detect after execution

- Each has a corresponding mask bit
  - if set when an exception occurs, the exception is handled automatically by FPU
  - if clear when an exception occurs, a software exception handler is invoked
Status register

C₃-C₀: condition bits after comparisons
FPU data types

.data
dbigVal REAL10 1.212342342234234243E+864
 code
fld bigVal

Table 17-11: Intrinsic Data Types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>QWORD</td>
<td>64-bit integer</td>
</tr>
<tr>
<td>TBYTE</td>
<td>80-bit (10-byte) integer</td>
</tr>
<tr>
<td>REAL4</td>
<td>32-bit (4-byte) IEEE short real</td>
</tr>
<tr>
<td>REAL8</td>
<td>64-bit (8-byte) IEEE long real</td>
</tr>
<tr>
<td>REAL10</td>
<td>80-bit (10-byte) IEEE extended real</td>
</tr>
</tbody>
</table>
FPU instruction set

- Instruction mnemonics begin with letter F
- Second letter identifies data type of memory operand
  - B = bcd
  - I = integer
  - no letter: floating point
- Examples
  - FBLD load binary coded decimal
  - FISTP store integer and pop stack
  - FMUL multiply floating-point operands
FPU instruction set

- **Fop** \{destination\}, \{source\}
- **Operands**
  - zero, one, or two
    - **fadd**
    - **fadd [a]**
    - **fadd st, st(1)**
  - no immediate operands
  - no general-purpose registers (EAX, EBX, ...) (FSTSW is the only exception which stores FPU status word to AX)
  - destination must be a stack register
  - integers must be loaded from memory onto the stack and converted to floating-point before being used in calculations
Classic stack (0-operand)

- ST(0) as source, ST(1) as destination. Result is stored at ST(1) and ST(0) is popped, leaving the result on the top. (with 0 operand, fadd=faddp)

```assembly
fld op1           ; op1 = 20.0
fld op2           ; op2 = 100.0
fadd
```

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th></th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0)</td>
<td>100.0</td>
<td>ST(0)</td>
<td>120.0</td>
</tr>
<tr>
<td>ST(1)</td>
<td>20.0</td>
<td>ST(1)</td>
<td></td>
</tr>
</tbody>
</table>
Memory operand (1-operand)

• ST(0) as the implied destination. The second operand is from memory.

FADD mySingle ; ST(0) = ST(0) + mySingle
FSUB mySingle  ; ST(0) = ST(0) - mySingle
FSUBR mySingle ; ST(0) = mySingle - ST(0)

FIADD myInteger ; ST(0) = ST(0) + myInteger
FISUB myInteger ; ST(0) = ST(0) - myInteger
FISUBR myInteger ; ST(0) = myInteger - ST(0)
Register operands (2-operand)

- Register: operands are FP data registers, one must be ST.

  \[
  \begin{align*}
  \text{FADD} & \quad \text{st, st(1)} \quad \text{; ST(0) = ST(0) + ST(1)} \\
  \text{FDIVR} & \quad \text{st, st(3)} \quad \text{; ST(0) = ST(3) / ST(0)} \\
  \text{FMUL} & \quad \text{st(2), st} \quad \text{; ST(2) = ST(2) * ST(0)}
  \end{align*}
  \]

- Register pop: the same as register with a ST pop afterwards.

  \[
  \text{FADDP} \quad \text{st(1), st}
  \]

**Before** | **Intermediate** | **After**
---|---|---
ST(0) | 200.0 | ST(0) | 200.0 | ST(0) | 232.0
ST(1) | 32.0 | ST(1) | 232.0 | ST(1) |
Example: evaluating an expression

\[(6.0 \times 2.0) + (4.5 \times 3.2)\]

INCLUDE Irvine32.inc

.data
array REAL4 6.0, 2.0, 4.5, 3.2
dotProduct REAL4 ?

.code
main PROC
  finit
  fld array ; push 6.0 onto the stack
  fmul array+4 ; ST(0) = 6.0 * 2.0
  fld array+8 ; push 4.5 onto the stack
  fmul array+12 ; ST(0) = 4.5 * 3.2
  fadd ; ST(0) = ST(0) + ST(1)
  fstp dotProduct ; pop stack into memory operand
exit
main ENDP
END main
(6.0 * 2.0) + (4.5 * 3.2)

fld array
fmul array+4
fld array+8
fmul array+12
fadd
fstp dotProduct
Load

FLD source  loads a floating point number from memory onto the top of the stack. The source may be a single, double or extended precision number or a coprocessor register.

FILD source  reads an integer from memory, converts it to floating point and stores the result on top of the stack. The source may be either a word, double word or quad word.

FLD1  stores a one on the top of the stack.

FLDZ  stores a zero on the top of the stack.

FLDPI  stores $\pi$

FLDL2T  stores $\log_2(10)$

FLDL2E  stores $\log_2(e)$

FLDLG2  stores $\log_{10}(2)$

FLDLN2  stores $\ln(2)$
load

.data
array REAL8 10 DUP(?)
.code
fld array ; direct
fld [array+16] ; direct-offset
fld REAL8 PTR[esi] ; indirect
fld array[esi] ; indexed
fld array[esi*8] ; indexed, scaled
fld REAL8 PTR[ebx+esi]; base-index
fld array[ebx+esi] ; base-index-displacement
Store

**FST dest** stores the top of the stack (ST0) into memory. The *destination* may either be a single or double precision number or a coprocessor register.

**FSTP dest** stores the top of the stack into memory just as FST; however, after the number is stored, its value is popped from the stack. The *destination* may either a single, double or extended precision number or a coprocessor register.

**FIST dest** stores the value of the top of the stack converted to an integer into memory. The *destination* may either a word or a double word. The stack itself is unchanged. How the floating point number is converted to an integer depends on some bits in the coprocessor’s *control word*. This is a special (non-floating point) word register that controls how the coprocessor works. By default, the control word is initialized so that it rounds to the nearest integer when it converts to integer. However, the **FSTCW** (Store Control Word) and **FLDCW** (Load Control Word) instructions can be used to change this behavior.

**FISTP dest** Same as **FIST** except for two things. The top of the stack is popped and the *destination* may also be a quad word.
Store

```
fst  dblOne ; 200.0
fst  dblTwo ; 200.0
fstp dblThree ; 200.0
fstp dblFour  ;  32.0
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0)</td>
<td>200.0</td>
</tr>
<tr>
<td>ST(1)</td>
<td>32.0</td>
</tr>
</tbody>
</table>
### Arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCHS</td>
<td>Change sign</td>
</tr>
<tr>
<td>FADD</td>
<td>Add source to destination</td>
</tr>
<tr>
<td>FSUB</td>
<td>Subtract source from destination</td>
</tr>
<tr>
<td>FSUBR</td>
<td>Subtract destination from source</td>
</tr>
<tr>
<td>FMUL</td>
<td>Multiply source by destination</td>
</tr>
<tr>
<td>FDIV</td>
<td>Divide destination by source</td>
</tr>
<tr>
<td>FDIVR</td>
<td>Divide source by destination</td>
</tr>
</tbody>
</table>

FCHS ; change sign of ST
FABS ; ST=|ST|
Floating-Point add

- **FADD**
  - adds source to destination
  - No-operand version pops the FPU stack after addition

- **Examples:**

  fadd st(1), st(0)  
  Before:  
  | ST(1) | 234.56 |
  | ST(0) | 10.1   |

  After:  
  | ST(1) | 244.66 |
  | ST(0) | 10.1   |
Floating-Point subtract

- **FSUB**
  - subtracts source from destination.
  - No-operand version pops the FPU stack after subtracting

- Example:
  
  ```assembly
  fsub mySingle ; ST -= mySingle
  fsub array[edi*8] ; ST -= array[edi*8]
  ```
Floating-point multiply/divide

- **FMUL**
  - Multiplies source by destination, stores product in destination

- **FDIV**
  - Divides destination by source, then pops the stack

FMUL\(^6\)
- FMUL m32fp
- FMUL m64fp
- FMUL ST(0), ST(i)
- FMUL ST(i), ST(0)

FDIV\(^7\)
- FDIV m32fp
- FDIV m64fp
- FDIV ST(0), ST(i)
- FDIV ST(i), ST(0)
**Miscellaneous instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCHS</td>
<td>ST0 = - ST0 Changes the sign of ST0</td>
</tr>
<tr>
<td>FABS</td>
<td>ST0 =</td>
</tr>
<tr>
<td>FSQRT</td>
<td>ST0 = √ST0 Takes the square root of ST0</td>
</tr>
<tr>
<td>FSSCALE</td>
<td>ST0 = ST0 × 2[^ST1^] multiples ST0 by a power of 2 quickly. ST1 is not removed from the coprocessor stack.</td>
</tr>
</tbody>
</table>

**.data**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>REAL4</td>
<td>2.75</td>
</tr>
<tr>
<td>five</td>
<td>REAL4</td>
<td>5.2</td>
</tr>
</tbody>
</table>

**.code**

```
fld      five ; ST0=5.2
fld      x     ; ST0=2.75, ST1=5.2
fscale   ; ST0=2.75×32=88
               ; ST1=5.2
```
Example: compute distance

; compute \( D = \sqrt{x^2+y^2} \)

fld x ; load x
fld st(0) ; duplicate x
fmul ; \( x^2 \)

fld y ; load y
fld st(0) ; duplicate y
fmul ; \( y^2 \)

fadd ; \( x^2+y^2 \)
fsqrt
fst D
Example: expression

; expression: valD = -valA + (valB * valC).
.data
valA REAL8 1.5
valB REAL8 2.5
valC REAL8 3.0
valD REAL8 ? \quad ; \text{will be } +6.0
.code
fld valA \quad ; \text{ST}(0) = \text{valA}
fchs \quad ; \text{change sign of ST}(0)
fld valB \quad ; \text{load valB into ST}(0)
fmul valC \quad ; \text{ST}(0) *= \text{valC}
fadd \quad ; \text{ST}(0) += \text{ST}(1)
fstp valD \quad ; \text{store ST}(0) to valD
Example: array sum

.data
N = 20
array REAL8 N DUP(1.0)
sum REAL8 0.0
.code
    mov ecx, N
    mov esi, OFFSET array
    fldz ; ST0 = 0
lp:    fadd REAL8 PTR [esi]; ST0 += *(esi)
    add esi, 8 ; move to next double
    loop lp
    fstp sum ; store result
Comparisons

**FCOM** `src` compares ST0 and `src`. The `src` can be a coprocessor register or a float or double in memory.

**FCOMP** `src` compares ST0 and `src`, then pops stack. The `src` can be a coprocessor register or a float or double in memory.

**FCOMPP**
compares ST0 and ST1, then pops stack twice.

**FICOM** `src`
compares ST0 and (float) `src`. The `src` can be a word or dword integer in memory.

**FICOMP** `src`
compares ST0 and (float) `src`, then pops stack. The `src` can be a word or dword integer in memory.

**FTST**
compares ST0 and 0.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code Bits</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C3</td>
<td>C2</td>
</tr>
<tr>
<td>fcom, fcomp, fcompp, ficom, ficomp</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X = Don’t care</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparisons

• The above instructions change FPU’s status register of FPU and the following instructions are used to transfer them to CPU.

  FSTSW dest  Stores the coprocessor status word into either a word in memory or the AX register.
  SAHF  Stores the AH register into the FLAGS register.
  LAHF  Loads the AH register with the bits of the FLAGS register.

• **SAHF** copies $C_0$ into carry, $C_2$ into parity and $C_3$ to zero. Since the sign and overflow flags are not set, use conditional jumps for unsigned integers ($ja$, $jae$, $jb$, $jbe$, $je$, $jz$).
Comparisons

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Status Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>CF</td>
</tr>
<tr>
<td>C1</td>
<td>(nonc)</td>
</tr>
<tr>
<td>C2</td>
<td>PF</td>
</tr>
<tr>
<td>C3</td>
<td>ZF</td>
</tr>
</tbody>
</table>

15 x87 FPU Status Word

0

FSTSW AX Instruction

15 AX Register

0

SAHF Instruction

31 EFLAGS Register

7

0
Branching after FCOM

• Required steps:
  1. Use the **FSTSW** instruction to move the FPU status word into **AX**.
  2. Use the **SAHF** instruction to copy AH into the **EFLAGS** register.
  3. Use **JA**, **JB**, etc to do the branching.

• Pentium Pro supports two new comparison instructions that directly modify CPU’s **FLAGS**.

  \[
  \texttt{FCOMI ST(0), src} \quad ; \quad \text{src=STn} \\
  \texttt{FCOMIP ST(0), src}
  \]

Example

  \[
  \texttt{fcomi ST(0), ST(1)} \\
  \texttt{jnb Label1}
  \]
Example: comparison

.data
x REAL8  1.0
y REAL8  2.0

.code
    ; if (x>y) return 1 else return 0
    fld x ; ST0 = x
    fcomp y ; compare ST0 and y
    fstsw ax ; move C bits into FLAGS
    sahf
    jna else_part ; if x not above y, ...
then_part:
    mov eax, 1
    jmp end_if
else_part:
    mov eax, 0
end_if:
Example: comparison

.data
x REAL8 1.0
y REAL8 2.0

.code
; if (x>y) return 1 else return 0
fld y ; ST0 = y
fld x ; ST0 = x ST1 = y
fcomi ST(0), ST(1)

jna else_part ; if x not above y, ...
then_part:
    mov eax, 1
jmp end_if
else_part:
    mov eax, 0
end_if:
Comparing for equality

- Not to compare floating-point values directly because of precision limit. For example, \( \sqrt{2.0} \times \sqrt{2.0} \neq 2.0 \)

<table>
<thead>
<tr>
<th>instruction</th>
<th>FPU stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>fld two</td>
<td>ST(0): +2.0000000E+000</td>
</tr>
<tr>
<td>fsqrt</td>
<td>ST(0): +1.4142135+000</td>
</tr>
<tr>
<td>fmul ST(0), ST(0)</td>
<td>ST(0): +2.0000000E+000</td>
</tr>
<tr>
<td>fsub two</td>
<td>ST(0): +4.4408921E-016</td>
</tr>
</tbody>
</table>
Comparing for equality

- Calculate the absolute value of the difference between two floating-point values

```
.data
epsilon REAL8 1.0E-12  ; difference value
val2  REAL8 0.0  ; value to compare
val3  REAL8 1.001E-13  ; considered equal to val2

.code
; if( val2 == val3 ), display "Values are equal".
    fld epsilon
    fld val2
    fsub val3
    fabs
    fcomi ST(0),ST(1)
    ja skip
    mWrite <!"Values are equal"!,0dh,0ah>
skip:
```
Example: quadratic formula

\[ ax^2 + bx + c = 0 \quad x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

```
fld  MinusFour  ; stack -4
fld  a          ; stack: a, -4
fld  c          ; stack: c, a, -4
fmulp st1, st0  ; stack: a*c, -4
fmulp st1, st0  ; stack: -4*a*c
fld  b          ; stack: b, b, -4*a*c
fld  b          ; stack: b*b, -4*a*c
fmulp st1, st0  ; stack: b*b - 4*a*c
faddp st1, st0
```
Example: quadratic formula

```plaintext
ftst  ; test with 0
fstsw ax
sahf
jb  no_real_solutions ; if disc < 0, no solutions
f.sqrt  ; stack: sqrt(b*b - 4*a*c)
fstp disc  ; store and pop stack
fld1  ; stack: 1.0
fld a  ; stack: a, 1.0
f.scale  ; stack: a * 2^(1.0) = 2*a, 1
fdivp st1,st0  ; stack: 1/(2*a)
fst one_over_2a  ; stack: 1/(2*a)
```

\[ x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]
Example: quadratic formula

fld b ; stack: b, 1/(2*a)
fld disc ; stack: disc, b, 1/(2*a)
fsubrp st1, st0 ; stack: disc - b, 1/(2*a)
fmulp st1, st0 ; stack: (-b + disc)/(2*a)
fstp root1 ; store in *root1
fld b ; stack: b
fld disc ; stack: disc, b
fchs ; stack: -disc, b
fsubrp st1, st0 ; stack: -disc - b
fmul one_over_2a ; stack: (-b - disc)/(2*a)
fstp root2 ; store in *root2

\[ x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]
Other instructions

- **F2XM1**: \( ST = 2^{ST(0)} - 1 \); \( ST \) in \([-1, 1]\)
- **FYL2X**: \( ST = ST(1) \cdot \log_2(ST(0)) \)
- **FYL2XP1**: \( ST = ST(1) \cdot \log_2(ST(0) + 1) \)
- **FPTAN**: \( ST(0) = 1; ST(1) = \tan(ST) \)
- **FPATAN**: \( ST = \arctan(ST(1)/ST(0)) \)
- **FSIN**: \( ST = \sin(ST) \) in radius
- **FCOS**: \( ST = \sin(ST) \) in radius
- **FSINCOS**: \( ST(0) = \cos(ST); ST(1) = \sin(ST) \)