Real Arithmetic

Computer Organization and Assembly Languages

Yung-Yu Chuang

2005/12/22

Announcement

- Grade for homework #3 is online
- You are encouraged to work in pairs for your final project
- You will have to schedule a demo session with your TA for your final project
- The final project will be due on the week after final week

Binary real numbers

- Binary real to decimal real

$$110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375$$

- Decimal real to binary real

$$\begin{aligned} 0.5625 \times 2 &= 1.125 & \text{first bit} &= 1 \\ 0.125 \times 2 &= 0.25 & \text{second bit} &= 0 \\ 0.25 \times 2 &= 0.5 & \text{third bit} &= 0 \\ 0.5 \times 2 &= 1.0 & \text{fourth bit} &= 1 \end{aligned}$$

$4.5625 = 100.1001_2$

IEEE floating point format

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

31	30 … 23	22 … 0
s	e	f

s sign bit - 0 = positive, 1 = negative

e biased exponent (8-bits) = true exponent + 7F (127 decimal). The values 00 and FF have special meaning (see text).

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

$23.85 = 10111.11\overline{0110}_2 = 1.011111\overline{0110} \text{x} 2^4$

$e = 127+4=83h$

0 100 0001 1 011 1110 1100 1100 1100 1100

IEEE floating point format

63	62 … 52	51 … 0
s	e	f

IEEE double precision

$e = 0$ and $f = 0$	denotes the number zero (which can not be normalized) Note that there is a +0 and -0.
$e = 0$ and $f \neq 0$	denotes a *denormalized number*. These are discussed in the next section.
$e = \mathrm{FF}$ and $f = 0$	denotes infinity (∞). There are both positive and negative infinities.
$e = \mathrm{FF}$ and $f \neq 0$	denotes an undefined result, known as *NaN* (Not a Number).

special values

Denormalized numbers

- Number smaller than 1.0×2^{-126} can't be presented by a single with normalized form. However, we can represent it with denormalized format.
- $1.001 \times 2^{-129} = 0.01001 \times 2^{-127}$

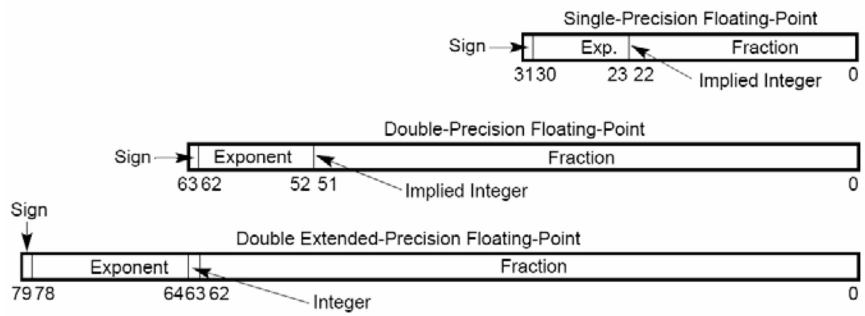
$\underline{0}$ 000 0000 0 $\underline{001\ 0010\ 0000\ 0000\ 0000\ 0000}$

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 was sold separately at early time. Later, FPU (floating-point unit) was integrated into CPU.

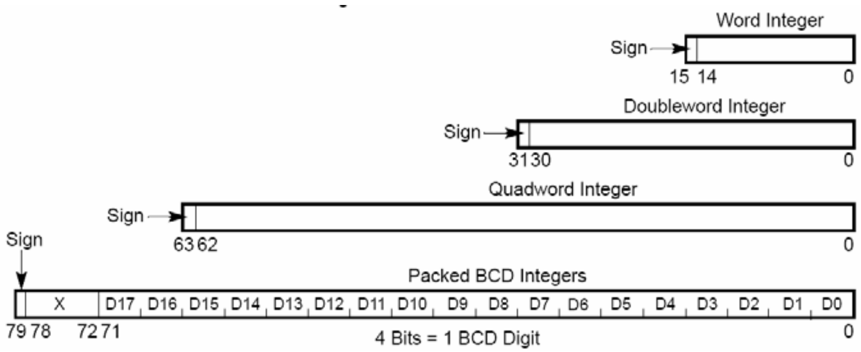
FPU data types

- Three floating-point types



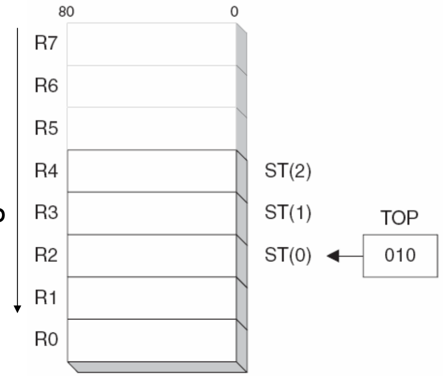
FPU data types

- Four integer types



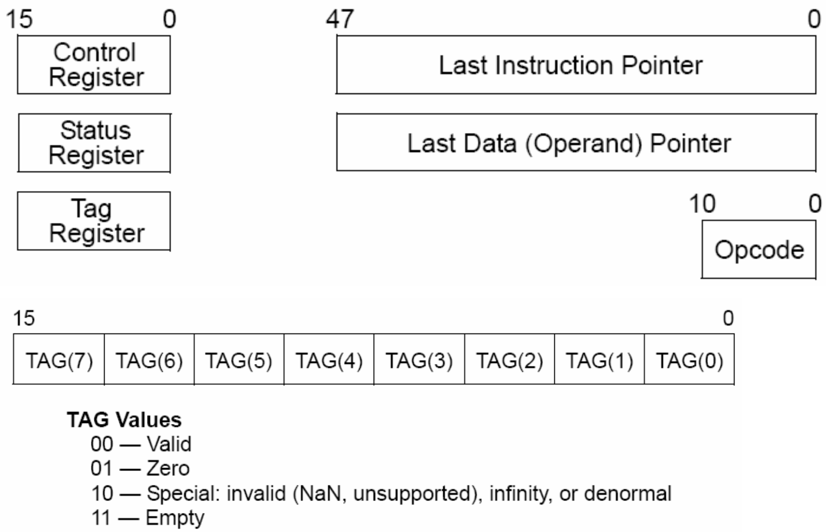
Data registers

- Load: push, TOP--
- Store: pop, TOP++
- Instructions access the stack using ST(i) relative to TOP
- If TOP=0 and push, TOP wraps to R7
- If TOP=7 and pop, result in an exception



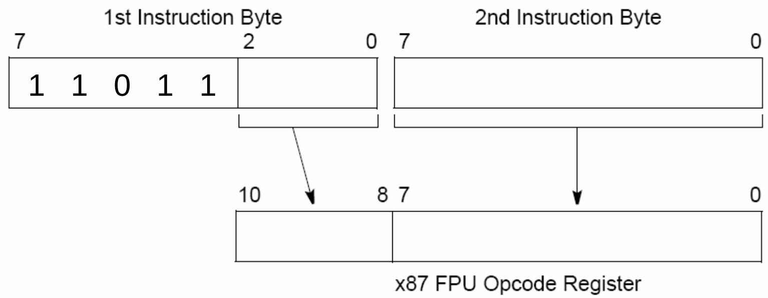
- Floating-point values are transferred to and from memory and stored in 10-byte temporary format. When storing, convert back to integer, long, real, long real.

Special-purpose registers

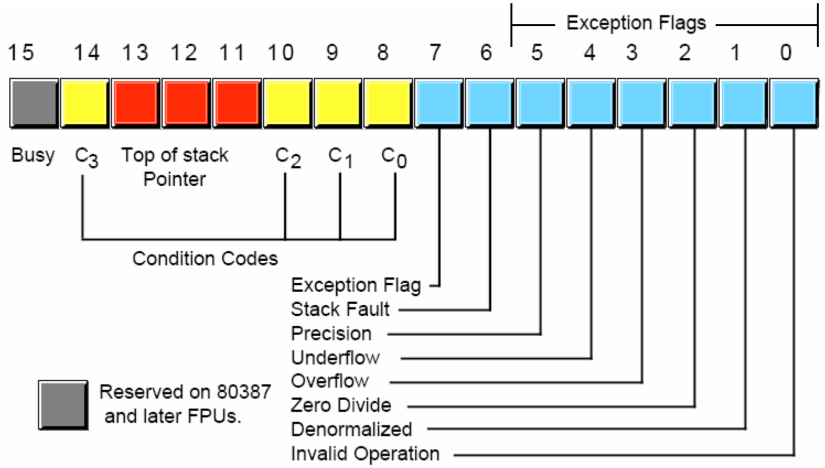


Special-purpose registers

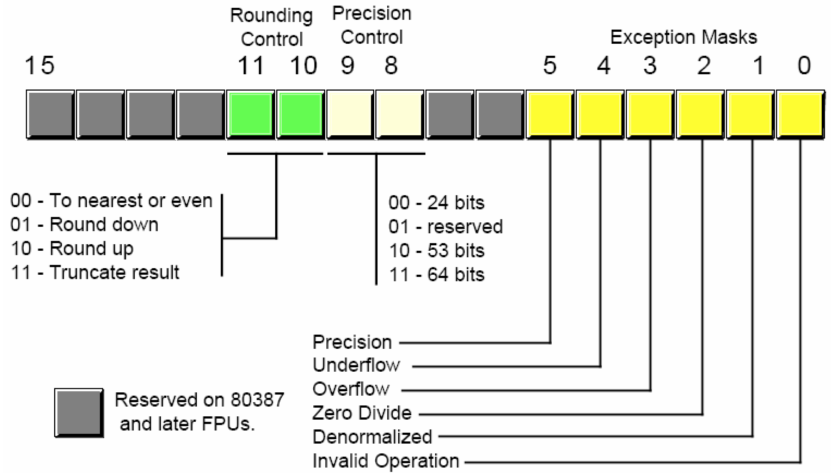
- Last data pointer stores the memory address of the operand for the last non-control instruction. Last instruction pointer stored the address of the last non-control instruction. Both are 48 bits, 32 for offset, 16 for segment selector.



Status register



Control register



Initial 037Fh

Instruction format

- Begin with ‘F’. The second letter could be ‘B’ (binary-coded decimal), ‘I’ (binary integer) or none (real).
- Up to two operands, at least one of them is a floating-point register. Hence, no memory-memory operation. No immediate and CPU register operands.

Instruction format

Table 17-9 Basic FPU Instruction Formats.

Instruction Format	Mnemonic Format	Operands (Dest, Source)	Example
Classical Stack	F*op*	{ST(1),ST}	FADD
Classical Stack, extra pop	F*op*P	{ST(1),ST}	FSUBP
Register	F*op*	ST(n),ST ST, ST(n)	FMUL ST(1),ST FDIV ST,ST(3)
Register, pop	F*op*P	ST(n),ST	FADDP ST(2),ST
Real Memory	F*op*	{ST},memReal	FDIVR
Integer Memory	FI*op*	{ST},memInt	FSUBR hours

{...}: implied operands

Classic stack

- 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.

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

	Before		After
ST(0)	100.0	ST(0)	120.0
ST(1)	20.0	ST(1)	

Real memory and integer memory

- 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 and register pop

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

```
FADD   st,st(1)        ; ST(0) = ST(0) + ST(1)
FDIVR  st,st(3)        ; ST(0) = ST(3) / ST(0)
FMUL   st(2),st        ; ST(2) = ST(2) * ST(0)
```

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

```
FADDP 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)	

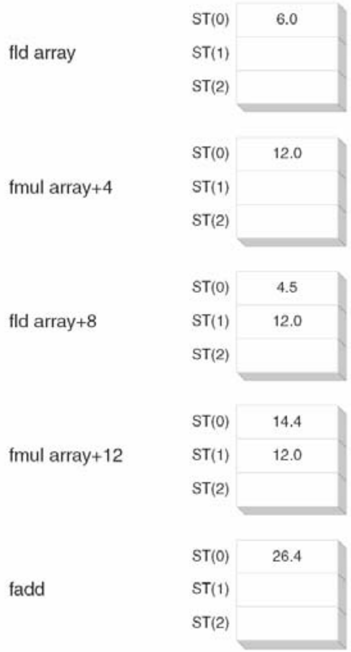
Example: evaluating an expression

(6.0 * 2.0) + (4.5 * 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
```

```
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 π
`FLDL2T`	stores $\log_2(10)$
`FLDL2E`	stores $\log_2(e)$
`FLDLG2`	stores $\log_{10}(2)$
`FLDLN2`	stores $\ln(2)$

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.

Register

`FXCH ST`*n*	exchanges the values in `ST0` and `ST`*n* on the stack (where *n* is register number from 1 to 7).
`FFREE ST`*n*	frees up a register on the stack by marking the register as unused or empty.

Addition

FADD *src*	ST0 += *src*. The *src* may be ~~any coprocessor register~~ or a single or double precision number in memory.
FADD *dest*, ST0	*dest* += ST0. The *dest* may be any coprocessor register.
~~FADDP *dest* or~~ FADDP *dest*, ST0	*dest* += ST0 then pop stack. The *dest* may be any coprocessor register.
FIADD *src*	ST0 += (float) *src*. Adds an integer to ST0. The *src* must be a word or double word in memory.

Addition

FADD *m32fp*	Add *m32fp* to ST(0) and store result in ST(0).
FADD *m64fp*	Add *m64fp* to ST(0) and store result in ST(0).
FADD ST(0), ST(i)	Add ST(0) to ST(i) and store result in ST(0).
FADD ST(i), ST(0)	Add ST(i) to ST(0) and store result in ST(i).
FADDP ST(i), ST(0)	Add ST(0) to ST(i), store result in ST(i), and pop the register stack.
FADDP	Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
FIADD *m32int*	Add *m32int* to ST(0) and store result in ST(0).
FIADD *m16int*	Add *m16int* to ST(0) and store result in ST(0).

Subtraction

FSUB *src*	ST0 -= *src*. The *src* may be ~~any coprocessor register~~ or a single or double precision number in memory.
FSUBR *src*	ST0 = *src* - ST0. The *src* may be ~~any coprocessor register~~ or a single or double precision number in memory.
FSUB *dest*, ST0	*dest* -= ST0. The *dest* may be any coprocessor register.
FSUBR *dest*, ST0	*dest* = ST0 - *dest*. The *dest* may be any coprocessor register.
~~FSUBP *dest* or~~ FSUBP *dest*, ST0	*dest* -= ST0 then pop stack. The *dest* may be any coprocessor register.
~~FSUBRP *dest* or~~ FSUBRP *dest*, ST0	*dest* = ST0 - *dest* then pop stack. The *dest* may be any coprocessor register.
FISUB *src*	ST0 -= (float) *src*. Subtracts an integer from ST0. The *src* must be a word or double word in memory.
FISUBR *src*	ST0 = (float) *src* - ST0. Subtracts ST0 from an integer. The *src* must be a word or double word in memory.

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
```

Multiplication

`FMUL src`	`ST0 *= src`. The *src* may be ~~any coprocessor register~~ or a single or double precision number in memory.
`FMUL dest, ST0`	`dest *= ST0`. The *dest* may be any coprocessor register.
~~`FMULP dest or`~~ `FMULP dest, ST0`	`dest *= ST0` then pop stack. The *dest* may be any coprocessor register.
`FIMUL src`	`ST0 *= (float) src`. Multiplies an integer to `ST0`. The *src* must be a word or double word in memory.
`FMULP`	**`ST(1)=ST(0)*ST(1),`** pop **`ST(0)`**

```
; Compute Z := sqrt(x**2 + y**2);
fld     x           ;Load X.
fld     st(0)       ;Duplicate X on TOS.
fmul                ;Compute X**2.

fld     y           ;Load Y.
fld     st(0)       ;Duplicate Y on TOS.
fmul                ;Compute Y**2.

fadd                ;Compute X**2 + Y**2.
fsqrt               ;Compute sqrt(x**2 + y**2).
fst     Z           ;Store away result in Z.
```

Division

`FDIV src`	`ST0 /= src`. The *src* may be ~~any coprocessor register~~ or a single or double precision number in memory.
`FDIVR src`	`ST0 = src / ST0`. The *src* may be ~~any coprocessor register or~~ a single or double precision number in memory.
`FDIV dest, ST0`	`dest /= ST0`. The *dest* may be any coprocessor register.
`FDIVR dest, ST0`	`dest = ST0 / dest`. The *dest* may be any coprocessor register.
~~`FDIVP dest or`~~ `FDIVP dest, ST0`	`dest /= ST0` then pop stack. The *dest* may be any coprocessor register.
~~`FDIVRP dest or`~~ `FDIVRP dest, ST0`	`dest = ST0 / dest` then pop stack. The *dest* may be any coprocessor register.
`FIDIV src`	`ST0 /= (float) src`. Divides `ST0` by an integer. The *src* must be a word or double word in memory.
`FIDIVR src`	`ST0 = (float) src / ST0`. Divides an integer by `ST0`. The *src* must be a word or double word in memory.

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.

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
fcom, fcomp, fcompp, ficom, ficomp	0	0	X	0	ST > source
	0	0	X	1	ST < source
	1	0	X	0	ST = source
	1	1	X	1	ST or source undefined
	X = Don't care				

Comparisons

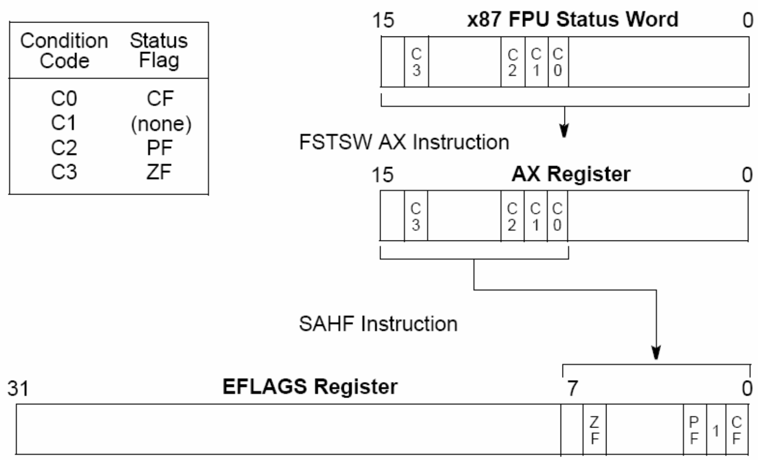
- The above instructions change FPU's status register of FPU and the following instructions are used to transfer them to CPU.

`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

Condition Code	Status Flag
C0	CF
C1	(none)
C2	PF
C3	ZF



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:
```

Pentium Pro new comparison

- Pentium Pro supports two new comparison instructions that directly modify CPU's FLAGS.

FCOMI *src*	compares ST0 and *src*. The *src* must be a coprocessor register.
FCOMIP *src*	compares ST0 and *src*, then pops stack. The *src* must be a coprocessor register.

The format should be

FCOMI ST(0), ***src***
FCOMIP ST(0), ***src***

Example: max=max(x,y)

```
.686
.data
x     REAL8     1.0
y     REAL8     2.0
max   REAL8     ?
.code
      fld       y
      fld       x          ; ST0=x, ST1=y
      fcomip    st(0), st(1)
      jna       y_bigger
      fcomp     st(0)      ; pop y from stack
      fld       x          ; ST0=x
y_bigger:
      fstp      max
```

Miscellaneous instructions

FCHS	ST0 = - ST0 Changes the sign of ST0
FABS	ST0 = |ST0| Takes the absolute value of ST0
FSQRT	ST0 = $\sqrt{\texttt{ST0}}$ Takes the square root of ST0
FSCALE	ST0 = ST0 $\times 2^{\lfloor \texttt{ST1} \rfloor}$ multiples ST0 by a power of 2 quickly. ST1 is not removed from the coprocessor stack.

```
.data
x    REAL4    2.75
five REAL4    5.2
.code
     fld      five      ; ST0=5.2
     fld      x         ; ST0=2.75, ST1=5.2
     fscale             ; ST0=2.75*32=88
                        ; ST1=5.2
```

Example: quadratic formula

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

```
fild    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
fld     b               ; stack: b, b, -4*a*c
fmulp   st1,st0         ; stack: b*b, -4*a*c
faddp   st1,st0         ; stack: b*b - 4*a*c
```

Example: quadratic formula

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

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
```