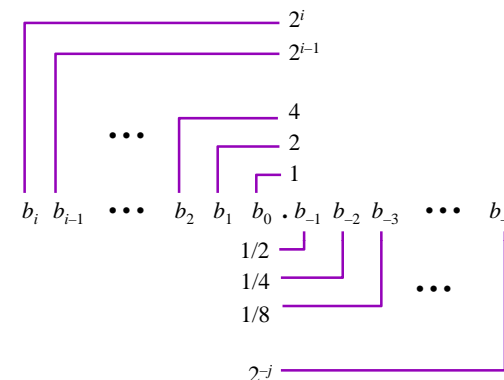


Real Arithmetic

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

$0.5625 \times 2 = 1.125$	first bit = 1
$0.125 \times 2 = 0.25$	second bit = 0
$0.25 \times 2 = 0.5$	third bit = 0
$0.5 \times 2 = 1.0$	fourth bit = 1

$$4.5625 = 100.1001_2$$

Fractional binary numbers examples

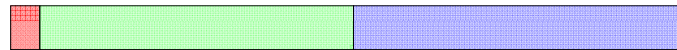


- | • Value | Representation |
|---------|------------------------------|
| 5-3/4 | 101.11_2 |
| 2-7/8 | 10.111_2 |
| 63/64 | 0.111111_2 |
| • Value | Representation |
| 1/3 | $0.0101010101[01]..._2$ |
| 1/5 | $0.001100110011[0011]..._2$ |
| 1/10 | $0.0001100110011[0011]..._2$ |

Fixed-point numbers



sign integer part fractional part



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.

5

IEEE floating point



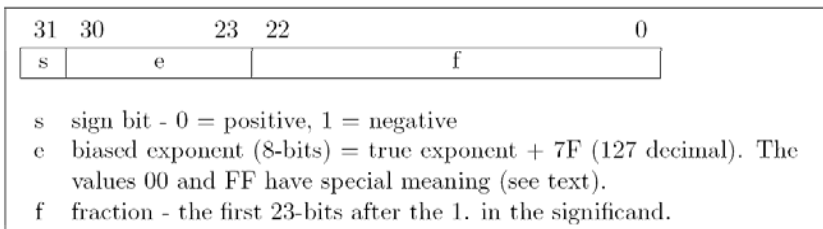
- IEEE Standard 754
 - Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many idiosyncratic formats
 - Supported by all major CPUs
- Driven by Numerical Concerns
 - Nice standards for rounding, overflow, underflow
 - Hard to make go fast
 - Numerical analysts predominated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

6

IEEE floating point format

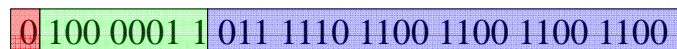


- IEEE defines two formats with different precisions: single and double



$$23.85 = 10111.110\overline{110}_2 = 1.0111110\overline{110}_2 \times 2^4$$

$$e = 127 + 4 = 83h$$



7

IEEE floating point format



$e = 0$ 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 <i>denormalized number</i> . These are discussed in the next section.
$e = FF$ and $f = 0$	denotes infinity (∞). There are both positive and negative infinities.
$e = FF$ and $f \neq 0$	denotes an undefined result, known as <i>NaN</i> (Not a Number).

special values



IEEE double precision

8

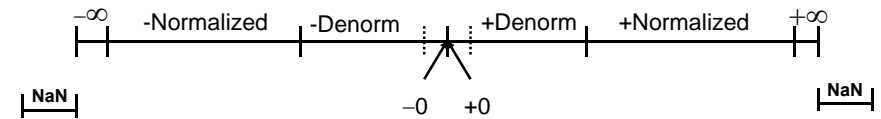
Denormalized numbers



- 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.
 - $1.0000..00 \times 2^{-126}$ the least “normalized” number
 - $0.1111..11 \times 2^{-126}$ the largest “denormalized” number
 - $1.001 \times 2^{-129} = 0.001001 \times 2^{-126}$
- 0 000 0000 0 001 0010 0000 0000 0000 0000

9

Summary of Real Number Encodings



$$(3.14 + 1e20) - 1e20 = 0$$

$$3.14 + (1e20 - 1e20) = 3.14$$

10

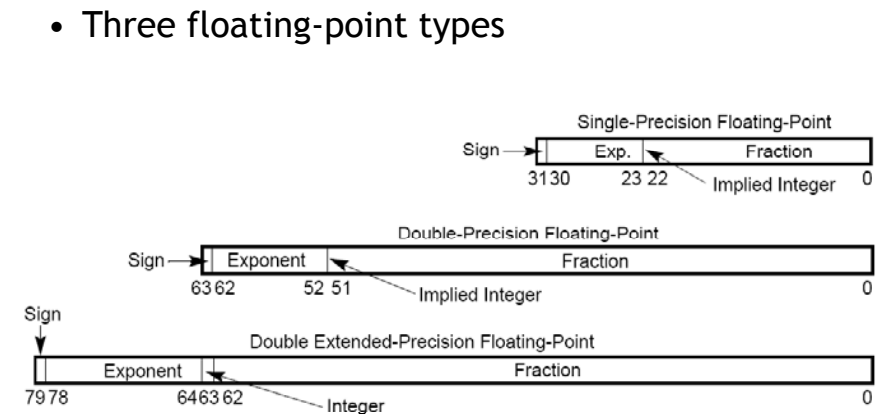
IA-32 floating point architecture



- 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.

11

FPU data types

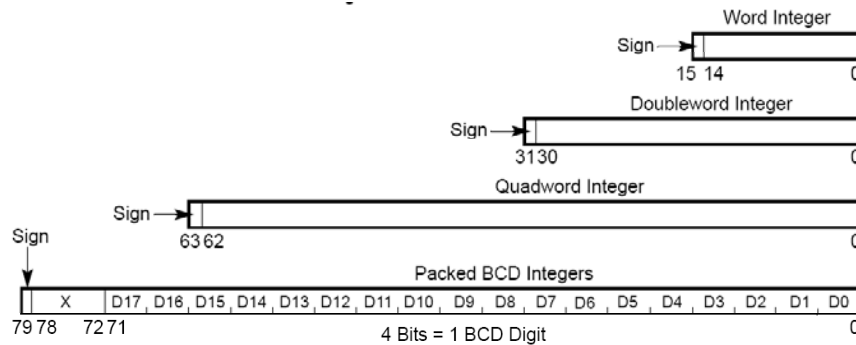


12

FPU data types



- Four integer types



13

FPU registers



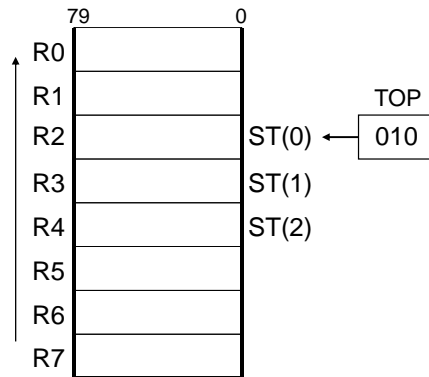
- Data register
- Control register
- Status register
- Tag register

14

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, 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.

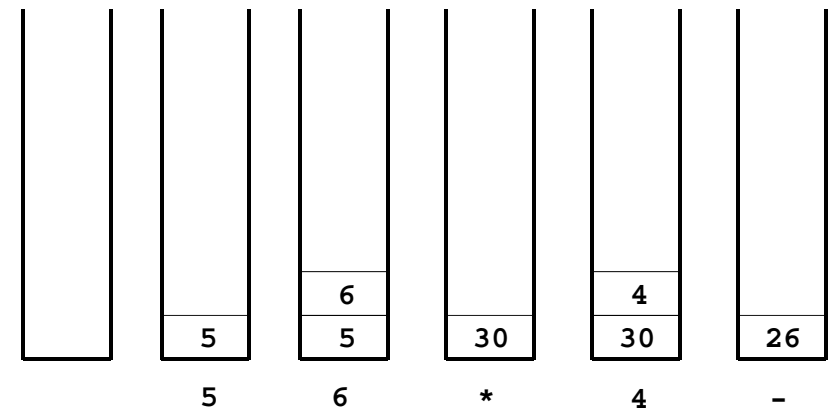


15

Postfix expression

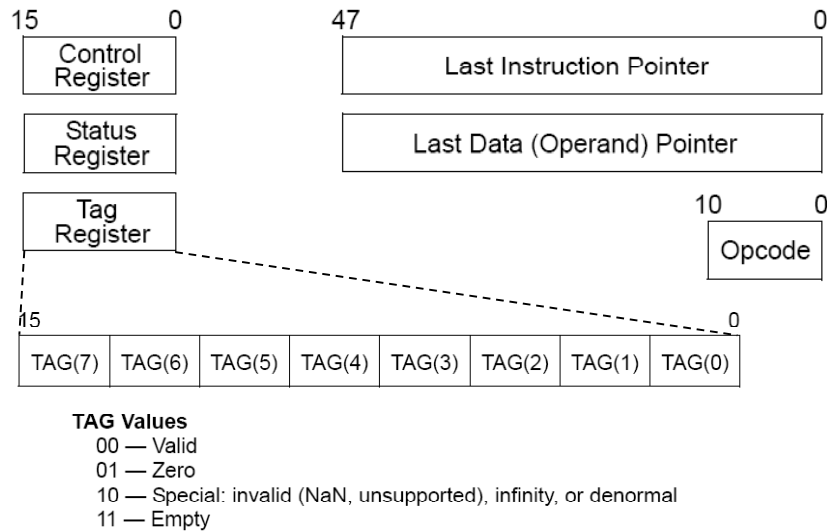


- $(5*6)-4 \rightarrow 5 \ 6 \ * \ 4 \ -$



16

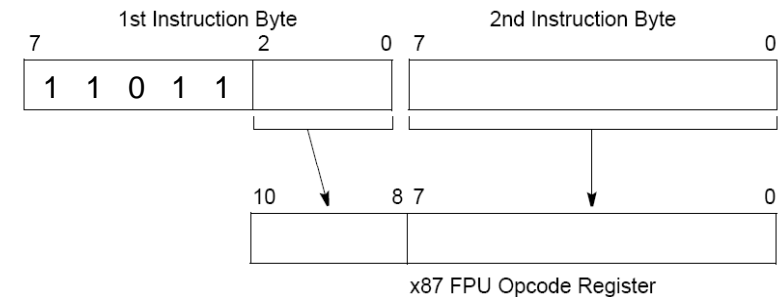
Special-purpose registers



17

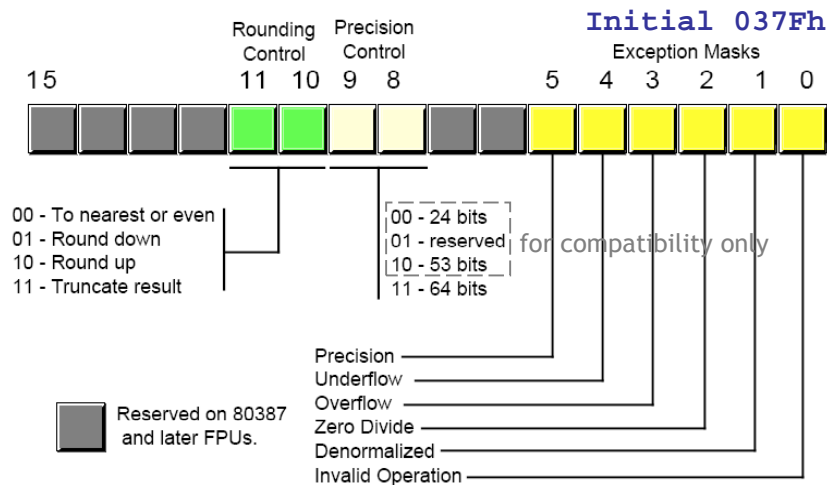
Special-purpose registers

- 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.



18

Control register



The instruction **FINIT** will initialize it to 037Fh.

19

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 .0001 produces 1.100
 - rounding down by subtracting .0001 produces 1.011

20

Rounding



method	original value	rounded value
Round to nearest even	1.0111	1.100
Round down	1.0111	1.011
Round up	1.0111	1.100
Truncate	1.0111	1.011

method	original value	rounded value
Round to nearest even	-1.0111	-1.100
Round down	-1.0111	-1.100
Round up	-1.0111	-1.011
Truncate	-1.0111	-1.011

21

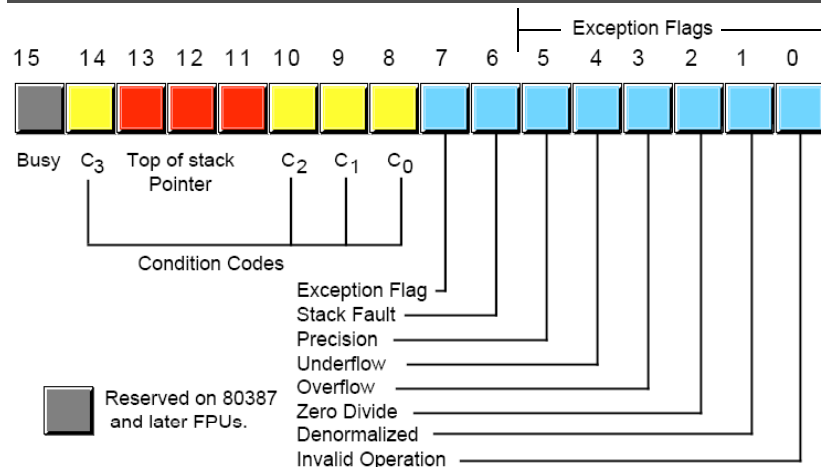
Floating-Point Exceptions



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

22

Status register



C₃-C₀: condition bits after comparisons

23

FPU data types



```
.data
bigVal REAL10 1.212342342234234243E+864
.code
fld bigVal
```

Table 17-11 Intrinsic Data Types.

Type	Usage
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

24

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

25

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

26

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**)

```
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)	

27

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)
```

28

Register operands (2-operand)



- 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	200.0	232.0
ST(1)	32.0	232.0	

29

Example: evaluating an expression



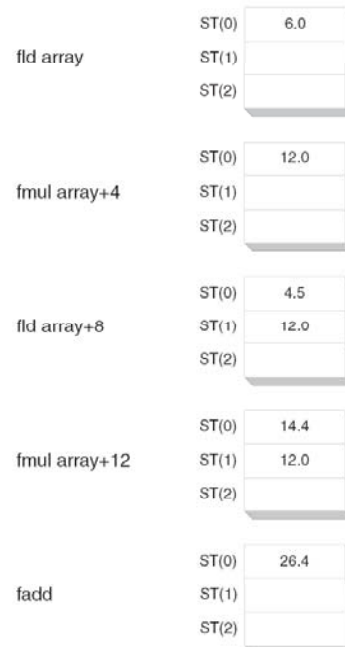
```
INCLUDE Irvine32.inc      (6.0 * 2.0) + (4.5 * 3.2)
.data
array      REAL4 6.0, 2.0, 4.5, 3.2
dotProduct REAL4 ?

.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
```

30

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

32

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
```

33

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 be 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 be 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.

34

Store



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

ST(0)	200.0
ST(1)	32.0

35

Arithmetic instructions



Table 17-12 Basic Floating-Point Arithmetic Instructions.

FCHS	Change sign
FADD	Add source to destination
FSUB	Subtract source from destination
FSUBR	Subtract destination from source
FMUL	Multiply source by destination
FDIV	Divide destination by source
FDIVR	Divide source by destination

FCHS ; change sign of ST

FABS ; ST = |ST|

36

Floating-Point add



- FADD

- adds source to destination FADD⁴
- No-operand version pops the FPU stack after addition FADD *m32fp*
FADD *m64fp*

- Examples:

fadd st(1), st(0)	Before:	ST(1)	234.56
		ST(0)	10.1
	After:	ST(1)	244.66
		ST(0)	10.1

37

Floating-Point subtract



- FSUB

- subtracts source from destination. FSUB⁵
- No-operand version pops the FPU stack after subtracting FSUB *m32fp*
FSUB *m64fp*
FSUB ST(0), ST(*i*)
FSUB ST(*i*), ST(0)

- Example:

```
fsub mySingle      ; ST -= mySingle
fsub array[edi*8] ; ST -= array[edi*8]
```

38

Floating-point multiply/divide



- FMUL

- Multiplies source by destination, stores product in destination FMUL⁶
- FMUL *m32fp*
FMUL *m64fp*
FMUL ST(0), ST(*i*)
FMUL ST(*i*), ST(0)

- FDIV

- Divides destination by source, then pops the stack FDIV⁷
- FDIV *m32fp*
FDIV *m64fp*
FDIV ST(0), ST(*i*)
FDIV ST(*i*), ST(0)

39

Miscellaneous instructions



FCHS ST0 = - ST0 Changes the sign of ST0
 FABS ST0 = |ST0| Takes the absolute value of ST0
 FSQRT ST0 = $\sqrt{ST0}$ Takes the square root of ST0
 FSCALE ST0 = ST0 $\times 2^{\lfloor ST1 \rfloor}$ 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
```

40

Example: compute distance



```

; compute D=sqrt(x^2+y^2)
fld x          ; load x
fld st(0)      ; duplicate x
fmul          ; x*x

fld y          ; load y
fld st(0)      ; duplicate y
fmul          ; y*y

fadd          ; x*x+y*y
fsqrt
fst D
    
```

41

Example: expression



```

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

42

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
    
```

43

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,	0	0	X	0	ST > source
fcompp,	0	0	X	1	ST < source
ficom,	1	0	X	0	ST = source
ficompp	1	1	X	1	ST or source undefined
	X = Don't care				

4

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.

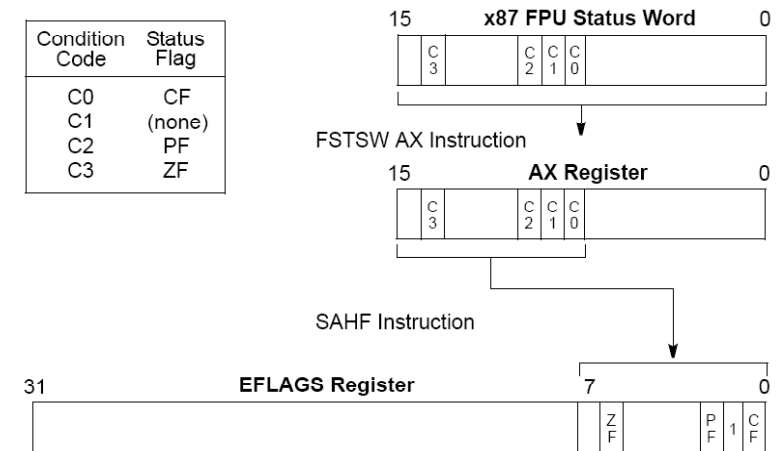
- SAHF** copies C_0 into carry, C_2 into parity and C_3 to zero. Since the sign and overflow flags are not set, use conditional jumps for unsigned integers (**ja**, **jae**, **jb**, **jbe**, **je**, **jz**).

45

Comparisons



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



46

Branching after FCOM



- Required steps:
 - Use the **FSTSW** instruction to move the FPU status word into **AX**.
 - Use the **SAHF** instruction to copy AH into the **EFLAGS** register.
 - Use **JA**, **JB**, etc to do the branching.
- Pentium Pro supports two new comparison instructions that directly modify CPU's FLAGS.

```
FCOMI ST(0), src ; src=STn
FCOMIP ST(0), src
```

Example

```
fcomi ST(0), ST(1)
jnb Label1
```

47

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:
```

48

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:
```

49

Comparing for equality



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

instruction	FPU stack
fld two	ST(0): +2.0000000E+000
fsqrt	ST(0): +1.4142135+000
fmul ST(0), ST(0)	ST(0): +2.0000000E+000
fsub two	ST(0): +4.4408921E-016

50

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:
```

51

Example: quadratic formula



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

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

52

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
fld   1.0            ; 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)
```

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

53

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
```

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

54

Other instructions



- **F2XM1** ; $ST=2^{ST(0)}-1$; ST in $[-1,1]$
- **FYL2X** ; $ST=ST(1)*\log_2(ST(0))$
- **FYL2XP1** ; $ST=ST(1)*\log_2(ST(0)+1)$

- **FPTAN** ; $ST(0)=1;ST(1)=\tan(ST)$
- **FPATAN** ; $ST=\arctan(ST(1)/ST(0))$
- **FSIN** ; $ST=\sin(ST)$ in radius
- **FCOS** ; $ST=\cos(ST)$ in radius
- **FSINCOS** ; $ST(0)=\cos(ST);ST(1)=\sin(ST)$

55