

Assembly Language for x86 Processors

6th Edition

Kip R. Irvine

Chapter 12: Floating-Point Processing and Instruction Encoding

Slide show prepared by the author

Revision date: 2/15/2010

(c) Pearson Education, 2010. All rights reserved. You may modify and copy this slide show for your personal use, or for use in the classroom, as long as this copyright statement, the author's name, and the title are not changed.

IEEE Floating-Point Binary Reals

- Types
 - Single Precision
 - 32 bits: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional part of the significand.
 -
 - Double Precision
 - 64 bits: 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fractional part of the significand.
 -
 - Double Extended Precision
 - 80 bits: 1 bit for the sign, 16 bits for the exponent, and 63 bits for the fractional part of the significand.

Floating Point Representation

- Floating point numbers are finite precision numbers used to approximate real numbers
- We will describe the IEEE-754 Floating Point Standard since it is adopted by most computer manufacturers: including Intel
- Like the scientific notation, the representation is broken up in 3 parts

Scientific notation: $-245.33 = -2.4533 \times 10^{-2} = -2.4533\text{E}-2$

- A sign s (either 0 or 1) ‘-’
- An exponent e -2
- A mantissa m (sometimes called a significand) -2.4533
- So that a floating point number N is written as: $(-1)^s \times m \times 10^e$
- Or, if m is in binary, N is written as:

$$N = (-1)^s \times m \times 2^e$$

- Were the binary mantissa is normalized such that :

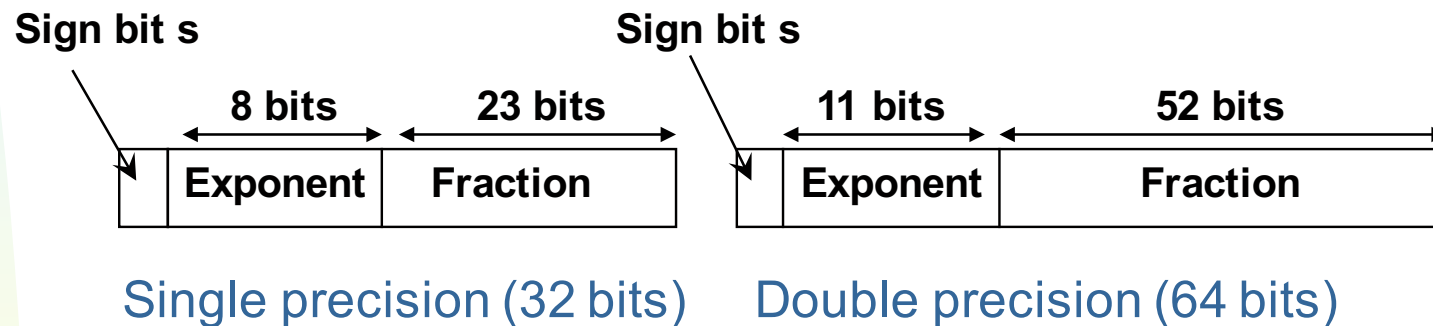
3 ▪ $m = 1.f$ with $1 \leq m < 2$ and $0 \leq f < 1$

Floating Point Representation (cont.)

- Hence we can write N in terms of fraction f : $0 \leq f < 1$

$$N = (-1)^s \times (1 + f) \times 2^e$$

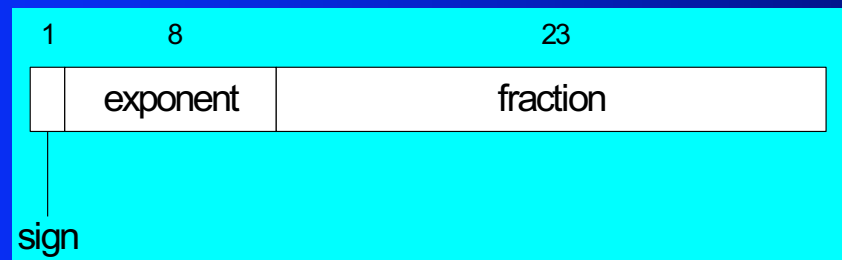
- The IEEE-754 standard defines the following formats:



- Hence, the value 1 in $1+f$ ($= 1.f$) is NOT stored: it is implied!
 - Mantissa: $1 \leq m = 1.f = 1+f < 2 \rightarrow 0 \leq f < 1$
- Extended precision formats (on 80 bits) with more bits for the exponent and fraction is also defined for use by the FPU

Single-Precision Format

Approximate normalized range: 2^{-126} to 2^{+127} .
Also called a *short real*.



The three formats are similar but differ only in their sizes. Thus, our discussions will focus only on the Single-Precision format.

- Double-Precision: 2^{-1022} to 2^{+1023}
- Extended-Precision: 2^{-32766} to 2^{+32767}

Components of a Single-Precision Real

- Sign **s**
 - 1 = negative, 0 = positive
- Significand **m**
 - **All** decimal digits to the left & right of decimal point
 - Weighted positional notation
 - Example: $123.154 = (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (1 \times 10^{-1}) + (5 \times 10^{-2}) + (4 \times 10^{-3})$
- Exponent **e**
 - signed integer: **$-126 \leq e \leq +127$** for single precision
 - integer bias: **an unsigned biased exponent $E = e + \textit{bias}$ is stored in the *exponent field* instead, where bias =**
 - 127 for single precision (thus $0 \leq E < 256$)
 - 1023 for double precision (thus $0 \leq E < 2048$)
 - 32767 for extended precision (thus $0 \leq E < 65536$)

The Exponent

- Sample Exponents represented in Binary
- Add bias 127 (for single-precision) to the actual exponent e to produce the biased exponent $E = e + 127$

Exponent (E)	Biased (E + 127)	Binary
+5	132	10000100
0	127	01111111
-10	117	01110101
+127	254	11111110
-126	1	00000001
-1	126	01111110

- Example:
 - Floating point number 1.27 has exponent $e = 0$. Hence: $E = 0 + 127 = 127 = 7Fh$ is stored in the exponent field
 - Floating point number $12.3 = 1.537 \cdot 2^3$ has $e = 3$. Hence: $E = 3 + 127 = 130 = 82h$ is stored in the exponent field
 - Floating point number $0.15 = 1.2 \cdot 2^{-3}$ has $e = -3$. Hence: $E = -3 + 127 = 124 = 7Ch$ is stored in the exponent field.

The mantissa must first be *normalized* **before biasing** the exponent

Normalizing Binary Floating-Point Numbers

- Mantissa m is normalized when a single 1 appears to the left of the binary point
- Unnormalized: shift binary point until exponent is zero
- Examples

Unnormalized	Normalized
1110.1	1.1101×2^3
.000101	1.01×2^{-4}
1010001.	1.010001×2^6

- Hence we can write N in terms of fraction f , $0 \leq f < 1$

$$N = (-1)^s \times (1 + f) \times 2^e$$

- **The value 1 in $1+f$ (= $1.f$) is NOT stored: it is implied!**

Representation for the Fraction

- In base 2, any fraction $f < 1$ can be written as:

$$f = b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \dots = (b_{-1}, b_{-2}, b_{-3}, \dots)$$

Where each $b_i \in \{0,1\}$ is a bit

and b_{-1} is the most significant bit (msb) of the fraction

- The algorithm to find each bit of a fraction f (ex: $f = .6$):
 - The msb of the fraction is 1 iff $f \geq \frac{1}{2}$. Hence the msb = 1 iff $2f \geq 1$.
 - Let f' be the fraction part of $2f$. Then the next msb of f is 1 iff $2f' \geq 1$.
 - Let f'' be the fraction part of $2f'$. Then the next msb of f is 1 iff $2f'' \geq 1$.
 - ... and so on

Converting Fractions to Binary Reals

- Express as a sum of fractions having denominators that are powers of 2 (or, sum of negative powers of 2)
- Examples

Decimal Fraction	Factored As...	Binary Real
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

Representation for the Fraction (cont.)

- Example: find all the bits of fraction $f = .15$

2×0.15	$= 0.30$	msb = 0
2×0.30	$= 0.60$	0
2×0.60	$= 1.20$	1
2×0.20	$= 0.40$	0
2×0.40	$= 0.80$	0
2×0.80	$= 1.60$	1
2×0.60	repeat of last 4 bits	

$$\text{Hence: } 0.15_{\text{ten}} = \overline{0.001001}_{\text{two}} = 0.00\mathbf{1001100110011001}_{\text{two}}\dots$$

When truncation is used, the following 23 bits will be stored in the single precision fraction field: $00\mathbf{100110011001100110011}$

Defining Floating Point Values in ASM

- We can use the DD directive to define a single precision floating point value. Ex:

```
float1 REAL4 17.15      ;single precision float
float2 REAL4 1.715E+1   ;same value as above
```

- The bits will be placed in memory according to the IEEE standard for single precision. Here we have:

- $17 = 10001b$ and $0.15 = 0.001001b$
- $17.15 = 10001.001001b = 1.0001001001b \times 2^4$
- Hence $e=4$. So $E = 127+4 = 131 = 10000011b$

- So if truncation is used for rounding, we have:

```
MOV eax,float1
; eax = 0 10000011 00010010011001100110011
; eax = 41893333h
;so float3 REAL4 41893333h is same as above definitions
float1 and float2
```

- We can use the DQ directive to define a double precision floating point value. Ex:

```
double1 dq 0.001235 ;double precision value
double2 dq 1.235E-3 ;same value as above
```

Rounding

- **Most of the real numbers are not exactly representable with a finite number of bits.**
 - Many rational numbers (like $1/3$ or 17.15) cannot be represented exactly in an IEEE format
- **Rounding refers to the way in which a real number will be approximated by another number that belongs to a given format**
 - Ex: if a format uses only 3 decimal digit to represent a fraction, should $2/3$ be represented as 0.666 or 0.667 ?
- **Truncation is only one of the methods used for rounding. Three other methods are supported by the IEEE standard:**
 - Round to nearest number (the default for IEEE)
 - Round towards + infinity
 - Round towards – infinity
- **Rounding to nearest is usually the best rounding method so it is chosen as the default. But since other methods are occasionally better, the IEEE standard specifies that the programmer can choose one of these 4 rounding methods.**

Real-Number Encodings

- Normalized finite numbers
 - all the nonzero finite values that can be encoded in a normalized real number between zero and infinity
- Positive and Negative Infinity
- NaN (not a number)
 - bit pattern that is not a valid FP value
 - Two types:
 - Quiet NaN: **does not cause an exception**
 - Signaling NaN: **causes an exception**
 - **Example: Divide-by-Zero**

Representation of Specific Values

$$N = (-1)^s \times (1 + f) \times 2^e$$

- Recall that exponent e uses a biased representation. It is represented by unsigned int E such that $e = E - 0111...1b$
- Let F be the unsigned integer obtained by concatenating the bits of the fraction f
- Hence a floating point number N is represented by (S,E,F) and the “1” in $1+f = 1.f$ is implied (not represented or included in F).
- Then note that we have no representation for zero!!
- Because of this, the IEEE standard specifies that zero is represented by $E = F = 0$
 - Hence, because of the sign bit, we have both a positive and a negative zero
- Only a few bits are allocated to E . So, a priori, numbers with very large (and very low) magnitudes cannot be represented.

Representation of Specific Values (cont.)

- Hence, the IEEE standard has reserved the following interpretation when E contains only ones
 - + infinity when $S = 0$, $E = 111..1$, and $F = 0$
 - - infinity when $S = 1$, $E = 111..1$, and $F = 0$
 - Not a Number (NaN) when $E = 111..1$, and $F \neq 0$
- Hence “normal” floating point values exist only for $E < 111..11$.
- The +/- infinity value arises when a computation gives a number that would require $E \geq 111..11$
- The +/- infinity value can be used in operands with predictable results. Ex:
 - $+infty + N = +infty$
 - $-infty + N = -infty$
 - $+infty + +infty = +infty$
- Undefined values are represented by NaN. Examples:
 - $+infty + -infty = NaN$
 - $+infty / +infty = NaN$
 - $0 / 0 = NaN$

Denormalized Numbers

- Now, the smallest nonzero magnitude would be when $E=0$ and $F = 00..01$. This would give a value of $1.00...01 \times 2^{-127}$ in single precision
- To allow smaller magnitudes to be represented, IEEE have introduced denormalized numbers
- A denormalized number has $E=0$ and $F \neq 0$. The implicit “1” to the left of “.” now becomes “0”.
 - Hence, the smallest nonzero single precision denormalized number is
$$0.00...01 \times 2^{-127} = 2^{-23} \times 2^{-127} = 2^{-150}$$
 - The largest single precision denormalized number is then
$$2^{-127} \times (1 - 2^{-23}).$$
- Hence normal numbers, called normalized numbers, use E such that $0 < E < 11...1$.
 - The smallest (positive) single precision normal number is then
$$1.00...0 \times 2^{-126}$$

Real-Number Encodings (cont)

- Specific encodings (single precision):

Value	Sign, Exponent, Significand
Positive zero	0 00000000 000000000000000000000000
Negative zero	1 00000000 000000000000000000000000
Positive infinity	0 11111111 000000000000000000000000
Negative infinity	1 11111111 000000000000000000000000
QNaN	x 11111111 1xxxxxxxxxxxxxxxxxxxxxxxxx
SNaN	x 11111111 0xxxxxxxxxxxxxxxxxxxxxxxxx ^a

Examples (Single Precision)

- Order: sign bit, exponent bits, and fractional part (mantissa)

Binary Value	Biased Exponent	Sign, Exponent, Fraction
-1.11	127	1 01111111 1100000000000000000000
+1101.101	130	0 1000010 1011010000000000000000
-.00101	124	1 01111100 0100000000000000000000
+100111.0	132	0 1000100 0011100000000000000000
+.0000001101011	120	0 01111000 1010110000000000000000

Converting Single-Precision to Decimal

1. If the MSB is 1, the number is negative; otherwise, it is positive.
2. The next 8 bits represent the exponent. Subtract binary 01111111 (decimal 127), producing the unbiased exponent. Convert the unbiased exponent to decimal.
3. The next 23 bits represent the significand. Notate a “1.”, followed by the significand bits. Trailing zeros can be ignored. Create a floating-point binary number, using the significand, the sign determined in step 1, and the exponent calculated in step 2.
4. Unnormalize the binary number produced in step 3. (Shift the binary point the number of places equal to the value of the exponent. Shift right if the exponent is positive, or left if the exponent is negative.)
5. From left to right, use weighted positional notation to form the decimal sum of the powers of 2 represented by the floating-point binary number.

Example

Convert 0 10000010 010110000000000000000000 → Decimal
S E F → (s,e,f)

1. The number is positive. $S = 0$
 - $s = +$
2. The unbiased exponent is binary 00000011, or decimal 3.
 - $e = E - 127 = 10000010 - 01111111 = 130 - 127 = +3$
3. Combining the sign s , exponent e , and significand f , the binary number is $+1.01011 \times 2^3$.
 - $F = 010110000000000000000000 \rightarrow f = \underline{1}.01011$
4. The unnormalized binary number is $+1010.11$.
 - Shift the binary point "." until the unbiased exponent $e = 0$
5. The decimal value is $+10 \frac{3}{4}$, or $+10.75$.
 - Simply convert $+1010.11$ to decimal: $+1010.11 \rightarrow +10.75$

Summary of IEEE Floating Point Numbers

- **Each number is represented by (S,E,F)**
 - S represents the sign of the number
 - The exponent “e” of the number is: $e = E - 011..1b$
 - F is the binary number obtained by concatenating the bits of the fraction
- **Normalized numbers have: $0 < E < 11..1$**
 - The implicit bit on the left of the decimal point is 1
- **Denormalized numbers have: $E = 0$ and $F \neq 0$**
 - The implicit bit on the left of the decimal point is 0
- **Zero is represented by $E = F = 0$**
- **+/- Infinity is represented by $E = 11..1$ and $F = 0$**
- **NaN is represented by $E = 11..1$ and $F \neq 0$**

Exercises

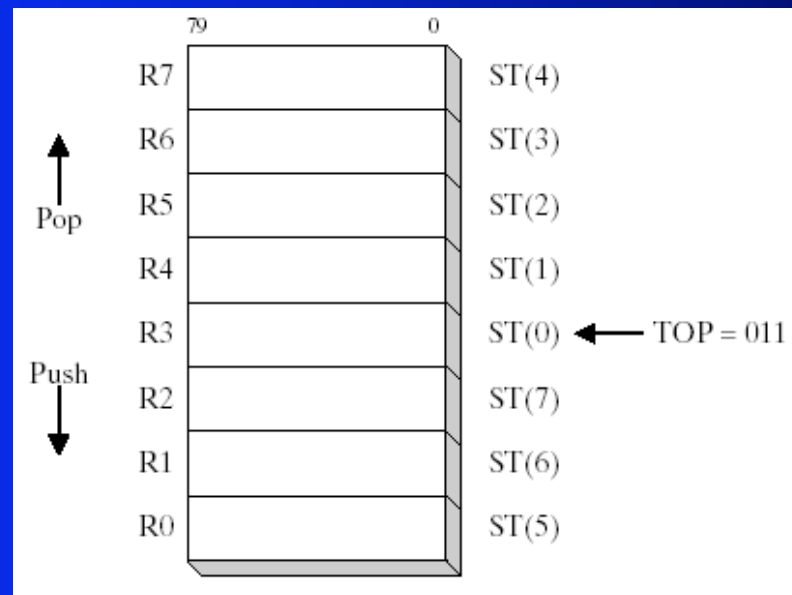
- **Exercise 1:** Find the IEEE single precision representation, in hexadecimal, of the following decimal numbers (assume that truncation is used for rounding):
 - 1.0
 - 0.5
 - -83.7
 - $1.1E-41$
- **Exercise 2:** Give the decimal value represented by the IEEE single precision representation given below in hexadecimal:
 - 45AC0000h
 - C4800000h
 - 3FE00000h

The Floating Point Unit (FPU)*

- **A FPU unit, designed to perform efficient computation with floating point numbers, is built (directly) on the Pentium processors**
 - It is backward compatible with older numerical coprocessors that were provided on a separate chip (ex: 8087 up to 387)
 - Use the .387, or .487, or .587, or .687 ... to enable assembly of FPU/coprocessor instructions
- **There are 8 general-purpose FPU registers; each 80-bit wide.**
 - Single-precision or double-precision values of the IEEE-754 standard are placed within those 80 bits in an extended format specified by Intel. (Intel FPUs conforms to the IEEE-754 standard)

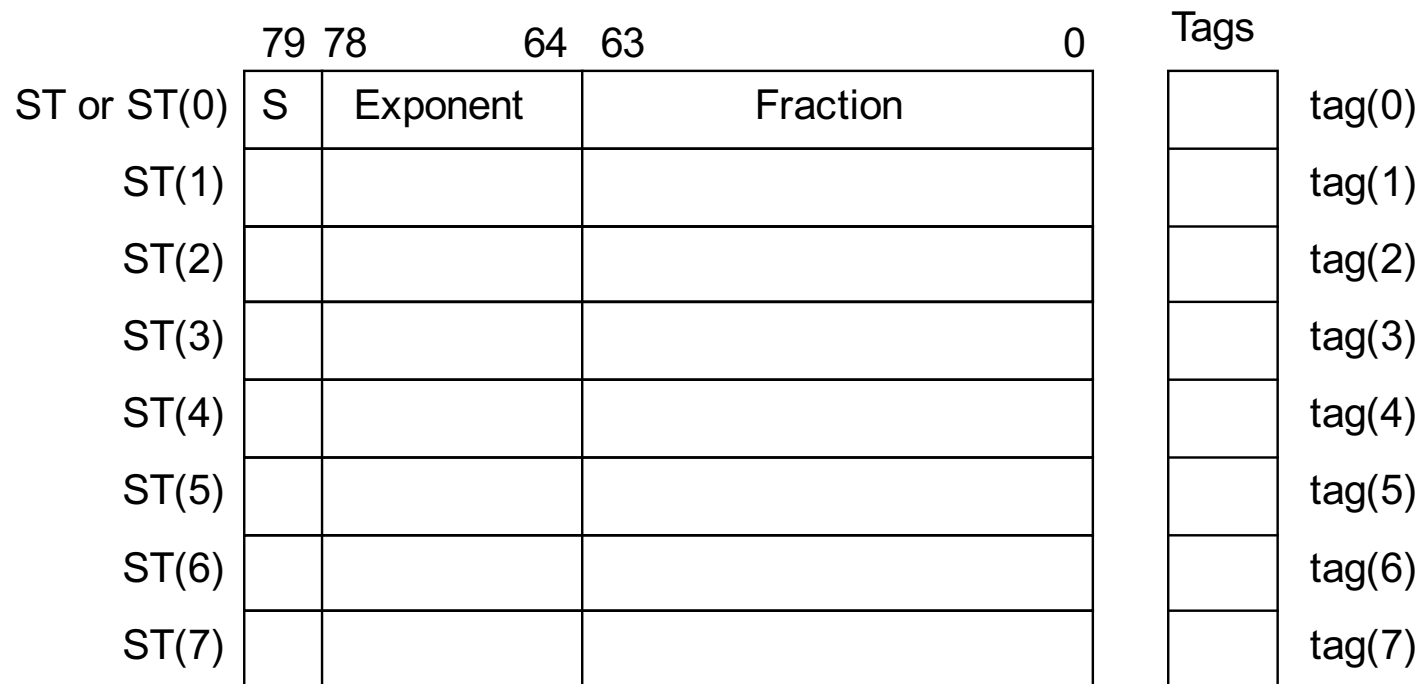
FPU Register Stack

- Eight individually addressable 80-bit data **general-purpose** registers named R0 through R7, organized as a stack
- Three-bit field named TOP in the FPU status word identifies the register number that is currently the top of stack.



General-Purpose FPU Registers

- They are organized as a stack maintained by the FPU
- The current top of the stack is referred by ST (Stack Top) or ST(0). ST(1) is the register just below ST and ST(n) is the n-th register below ST
- 15 bits are reserved for the exponent: $e = E - 3FFFh$
- The “1” in the mantissa 1.f is stored as an explicit 1 bit at position 63. Hence f is stored from bit 0 to bit 62.

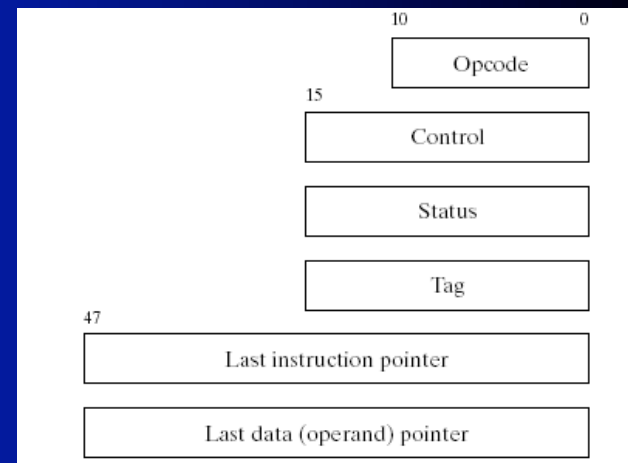


The Tag Register

- **The Tag register is a 16-bit register**
 - The first 2 bits, called $\text{tag}(0)$, specify the “type” of data contained in $\text{ST}(0)$.
 - $\text{Tag}(i)$ specify the “type” of data contained in $\text{ST}(i)$ for $i=0..7$
 - The 2-bit value of $\text{tag}(i)$ indicates the following about the content of $\text{ST}(i)$:
 - `00 : st(i) contains a valid number`
 - `01 : st(i) contains zero`
 - `10 : st(i) contains NaN or infty`
 - `11 : st(i) is empty`

Special-Purpose Registers

- Opcode register: stores opcode of last noncontrol instruction executed
- Control register: controls precision and rounding method for calculations
- Status register: top-of-stack pointer, condition codes, exception warnings
- Tag register: indicates content type of each register in the register stack
- Last instruction pointer register: pointer to last non-control executed instruction
- Last data (operand) pointer register: points to data operand used by last executed instruction



Rounding

- FPU attempts to round an infinitely accurate result from a floating-point calculation
 - may be impossible because of storage limitations
- 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

Floating-Point Exceptions

- Six types of exception conditions
 - Invalid operation
 - Divide by zero
 - Denormalized operand
 - Numeric overflow
 - 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

FPU Instruction Set

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

FPU Instruction Set

- Operands
 - zero, one, or two
 - no immediate operands
 - no general-purpose CPU registers (EAX, EBX, ...)
 - integers must be loaded from memory onto the stack and converted to floating-point before being used in calculations
 - if an instruction has two operands, one must be a FPU register

Data allocation directives

- **Single-Precision:** Use the REAL4 or DD directive to allocate 32 bits of storage for a floating point number and store a value according to the IEEE-754 standard. Ex:

```
spno REAL4 1.0 ; spno = 3F800000h
```

- **Double-Precision:** Use the REAL8 or QWORD or DQ directive to allocate 64 bits of storage and store a IEEE double precision value. Ex:

```
dpno REAL8 1.0 ; dpno = 3FF0000000000000h
```

- **Extended Double-Precision:** Use the REAL10 or TBYTE or DT directive to allocate 80 bits (Ten bytes) of storage and store a floating point number according to Intel's 80-bit extended precision format. Ex:

```
epno REAL10 1.0 ; epno = 3FFF8000000000000000h
```

- **Exercise 3:** Explain why value 1.0 is represented as above in single precision, double precision, and extended precision.

FP Instruction Set

- Data Types

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

- Note that QWORD and TBYTE are integer data types, not real data type.
 - QWORD used for defining integers
 - TBYTE used for defining packed BCD integers

Load Floating-Point Value

- FLD
- copies floating point operand from memory into the top of the FPU stack, ST(0)

```
FLD m32fp  
FLD m64fp  
FLD m80fp  
FLD ST(i)
```

- Example

```
.data  
db1One    REAL8 234.56  
db1Two    REAL8 10.1  
.code  
fld  db1One           ; ST(0) = db1One  
fld  db1Two           ; ST(0) = db1Two, ST(1) = db1One
```

- Use
 - **F**ILD for loading integers
 - **F**BLD for loading BCD integers

FPU Data Transfer Instructions

- Use the **FLD source** instruction to transfer data from a memory source onto ST.
- The mem operand can either be a real4, real8, real10, a quad word, or a ten byte.
- The data is converted from the IEEE format to Intel's extended precision format during the data transfer to ST.
- **Example:**

```
.data
  A REAL8  4.78E-7
  B REAL10 5.6E+8
.code
  fld A
  fld B
```

The FPU stack after loading A and B

ST(0)	B
ST(1)	A
ST(2)	
ST(3)	
ST(4)	
ST(5)	
ST(6)	
ST(7)	

Data Transfer Instructions (cont.)

- **ST(n) can be used as an operand of FLD.**
 - A CPU register cannot be an operand of FLD
- **In that case FLD ST(n) copies the content of ST(n) onto ST.**
- **Example: If we now execute FLD ST(1) after the previous instructions. We get the following FPU stack:**

ST(0)	A
ST(1)	B
ST(2)	A
ST(3)	
ST(4)	
ST(5)	
ST(6)	
ST(7)	

Store Floating-Point Value

- **FST**
 - copies floating point operand from the top of the FPU stack into memory
- **FSTP**
 - pops the stack after copying

```
FST  m32fp  
FST  m64fp  
FST  ST(i)
```

- **Use**
 - **FIST (FISTP)** for storing as integers
 - **FBST (FBSTP)** for storing as BCD integers

Data Transfer Instructions (cont.)

- The **FST destination instruction** can be used to transfer data from ST to a memory destination.
 - The mem operand can either be 32 bits, 64 bits, or 80 bits.
- **The CPU and FPU are executing concurrently**
 - This is why we normally cannot directly transfer data between CPU registers and FPU registers
 - When the FPU transfers data onto memory that is to be manipulated by the CPU, we should instruct the CPU to wait that the FPU completes the data transfer.

- **Example:**

```
.data
    float1 REAL4 1.75
    result DWORD ?
.code
    fld float1
    ...FPU inst...
    fist result
FWAIT
    mov eax,result
```

- **FWAIT tells the CPU to wait that the FPU finishes the instruction just before FWAIT**
 - If FWAIT is not used, EAX may not contain the result returned by the FPU !!

Data Transfer Instructions (cont.)

- ST(n) can be used as operand of FST.

Ex:

```
fst st(3); copies ST to ST(3)
```

- FST does not change ST
- But FSTP destination copies ST onto destination and pops ST
 - FSTP also permits a 80-bit mem operand

- Example:

```
fld A
fld B
fld C
fstp result
finit ;clears stack
```

ST(0)	C
ST(1)	B
ST(2)	A

Before fstp result

ST(0)	B
ST(1)	A
ST(2)	

After fstp result

ST(0)	
ST(1)	
ST(2)	

After finit

Data Transfer Instructions (cont.)

- The **FXCH** instruction swaps the content of two registers. It can be used either with zero or one operand.
 - If no operands are used, **FXCH** swaps the content of **ST** and **ST(1)**
 - If one operand is used, then it must be **ST(n)**. Example:

fld A

fld B

fld C

fxch ST(2) ; FXCH swaps the content of ST and ST(2)

ST(0)	C
ST(1)	B
ST(2)	A

Before fxch ST(2)

ST(0)	A
ST(1)	B
ST(2)	C

After fxch ST(2)

IEEE Format Conversion

- The `FLD` source instruction loads a memory floating point value onto `ST` in an extended 80 bit format regardless of whether source is a single precision, double precision, or extended precision floating point value
- The `FST[P]` destination instruction stores `ST` into memory regardless of whether destination is 32, 64, or 80 bits
- Hence we can convert from one format to another simply by pushing onto and popping from the FPU stack. Ex:

```
.data
    Adouble REAL8 -7.77E-6 ; double-precision value
    Afloat  REAL4 ?       ; single-precision value
.code
    FLD  Adouble ; double to extended precision
    FSTP Afloat  ; extended to single precision
```

Integer-to-Floating Point Conversion

- To convert from integer to floating point format we can use the **FILD** instruction. Ex:

```
.data
  A DWORD 5
.code
  FILD A ; Stores 5.0 on ST(0)
```

- To convert from floating point to integer format we can use the **FIST** instruction. Ex:

```
.data
  A REAL4 5.64
  B DWORD ?
.code
  FLD A ; stores 5.64 on ST(0)
  FIST B ; stores 6 in variable B
```

- The **FIST** instruction takes the floating point value in **ST(0)** and rounds it to an integer before storing it in the destination operand.
 - By default, the rounding method used is “round to the nearest” (but this can be changed by the programmer)

Floating-Point I/O

- Irvine32 library procedures
 - ReadFloat
 - reads FP value from keyboard, pushes it on the FPU stack. **Accept the following formats:**
 - 35, +35., -3.5, .35, 3.5E5, 3.5E005,
 - -3.5E+5, 3.5E-4, +3.5E-4
 - WriteFloat
 - writes value from ST(0) to the console window in exponential format
 - ShowFPUStack
 - displays contents of FPU stack

Arithmetic Instructions

- Same operand types as FLD and FST

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

- **All** of these instructions have their **FIXXX** counterparts except FCHS. Example: FIDIVR, ...
- They can have up to two operands as long as one of them is a FPU register.
 - **CPU registers are not allowed as operands**
 - A memory operand must be 32, or 64 bits
 - Memory-to-memory operations are not allowed.
 - Several addressing modes are provided.

Addressing Modes for Arithmetic Instructions

Addressing Mode	Mnemonic	Dest, Source	Example
Classical Stack	F XXX	{ST(1), ST}	FADD
Register (form 1)	F XXX	ST(n), ST	FMUL ST(1),ST
Register (form 2)	F XXX	ST, ST(n)	FDIV ST,ST(3)
Register + pop	F XXX P	ST(n), ST	FADDP ST(2),ST
Memory	F[I] XXX	{ST}, mem	FDIVR varA

- **Keyword ~~XXX~~ may be one of:**
 - ADD : add source to destination
 - SUB : subtract source from destination $D = D - S$
 - SUBR : subtract destination from source $D = S - D$
 - MUL : multiply source into destination
 - DIV : divide destination by source $D = D / S$
 - DIVR : divide source by destination $D = S / D$
- **The result is always stored into the destination operand**
- **Operands surrounded by ~~{...}~~ are implicit operands: not coded explicitly, by the programmer, in the instruction**

Classical Stack Addressing Mode

- The classical stack addressing mode is invoked when we use **FXXX** **without operands**.
 - ST is the implied source operand
 - ST(1) is the implied destination operand
- The result of the instruction is temporarily stored into ST(1) and then the stack is popped. Hence, ST will then contain the result. Example:

FLD A

FLD B

FLD C

FSUB ;st(1) = st(1) - st(0) = B-C

- **Note: ST(0) would contain C-B if FSUBR was used instead**

ST(0)	C
ST(1)	B
ST(2)	A

Before FSUB

ST(0)	B - C
ST(1)	A
ST(2)	

After FSUB

Register Addressing Mode

- **Uses explicitly two registers as operands** where one of them must be ST.
 - ST can either be the source or the destination operand, so two forms are permitted: **ST, ST(n)** or **ST(n), ST**.
 - In fact, both operands can be ST.
- **The stack is not popped after the operation. Ex:**

FLD A

FLD B

FLD C

FMUL ST(2),ST ; **st(2) = st(2) * st(0) = A*C**

ST(0)	C
ST(1)	B
ST(2)	A

Before FMUL st(2),st

ST(0)	C
ST(1)	B
ST(2)	A x C

After FMUL st(2),st

Register + Pop Addressing Mode

- **Uses explicitly two registers as operands.**
- **The source operand must be ST and the destination operand must be ST(n) where n must be different from 0.**
- **The result of the operation is first stored into ST(n) and then the stack is popped (so the result is then in ST(n-1). Ex:**

FLD A

FLD B

FLD C

FMULP ST(2),ST ;st(2) = st(2) * st(0) = A*C

;then pop st(0) is popped,

;hence st(1) = A*C, in the end

ST(0)	C
ST(1)	B
ST(2)	A

Before FMULP st(2),st

ST(0)	B
ST(1)	A x C
ST(2)	

After FMULP st(2),st

Memory Addressing Mode

- **ST is an implicit destination operand**
- **The source operand is either a 32, or a 64 bit memory operand**
- **Here is an example program that computes the area of a circle**

```
INCLUDE Irvine32.inc

.data
    pi      REAL4  3.14159
    radius  REAL4  2.0
    area    REAL4  ?

.code
main PROC
    fld    pi
    fld    radius
    fmul   radius ;mem addr.
           ; ST = radius*radius
           ; ST(1) = pi
    fmul   ; ST = area
    call  WriteFloat
           ; display area
    exit
main ENDP
END main
```

Floating-Point Add

- FADD
 - adds source to destination
 - No-operand version pops the FPU stack after subtracting
- Examples:

```
FADD4  
FADD m32fp  
FADD m64fp
```

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

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

```
FSUB5  
FSUB m32fp  
FSUB m64fp  
FSUB ST(0), ST(i)  
FSUB ST(i), ST(0)
```

- Example:

```
fsub mySingle          ; ST(0) -= mySingle  
fsub array[edi*8]     ; ST(0) -= array[edi*8]
```

Floating-Point Multiply

- 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)
```

The no-operand versions of FMUL and FDIV pop the stack after multiplying or dividing.

Arithmetic with an Integer

- The **memory addressing** mode also supports an integer for its explicit operand.
 - The arithmetic instruction must now be **FIXXX**
 - **The single operand must be either 16 or 32 bits integer**

```
.data
    five      DWORD 5          ; an integer
    my_float  REAL4 3.3        ; a floating point

.code
    ...
    fld my_float ; ST = 3.3
    fimul five   ; ST = 16.5
    fiadd five   ; ST = 21.5
```

Exercise 4

- Suppose that we have the following FPU stack content before each instruction below:

`ST(0) = 1.1, ST(1) = 1.2, ST(2) = 1.3, and the rest of the FPU stack is empty.`

`Give the stack content after the execution of each instruction:`

- `fstp result` ; result is a dword variable
- `fdivr st(2),st`
- `fmul`
- `fsubrp st(1),st`
- `fadd`
- `fdivp st,st(1)`

Other FPU Instructions

These instructions use ST as an Implicit operand and store the result back into ST:

Instruction	Description
FABS	Convert ST to positive
FCHS	Change the sign of ST
FSQRT	Compute SQRT of ST
FSIN	Compute SIN of ST
FCOS	Compute COS of ST

These instructions push a constant onto ST:

Instruction	Constant
FLDZ	"+0.0"
FLD1	"+1.0"
FLDPI	Pi
FLDL2T	LOG ₂ (10)
FLDL2E	LOG ₂ (e)
FLDLG2	LOG ₁₀ (2)
FLDLN2	LOG _e (2)

Example: Finding the roots of a quadratic equation using:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For more instructions see Intel's Documentation at:

<http://www.intel.com/design/litcentr/index.htm>

In particular, see Intel's Architecture Software Developer's Manual Vol. 1 & 2

Comparing FP Values

- FCOM instruction
- Operands:

Instruction	Description
FCOM	Compare ST(0) to ST(1)
FCOM <i>m32fp</i>	Compare ST(0) to <i>m32fp</i>
FCOM <i>m64fp</i>	Compare ST(0) to <i>m64fp</i>
FCOM ST(<i>i</i>)	Compare ST(0) to ST(<i>i</i>)

FCOM

- Condition codes set by FPU
 - codes similar to CPU flags

Condition	C3 (Zero Flag)	C2 (Parity Flag)	C0 (Carry Flag)	Conditional Jump to Use
ST(0) > SRC	0	0	0	JA, JNBE
ST(0) < SRC	0	0	1	JB, JNAE
ST(0) = SRC	1	0	0	JE, JZ
Unordered ^a	1	1	1	(None)

^aIf an invalid arithmetic operand exception is raised (because of invalid operands) and the exception is masked, C3, C2, and C0 are set according to the row marked *Unordered*.

Branching after FCOM

- Required steps:
 1. Use the FNSTSW 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.

Fortunately, the FCOMI instruction does steps 1 and 2 for you.

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

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:
```

Exception Synchronization

- Main CPU and FPU can execute instructions concurrently
 - if an unmasked exception occurs, the current FPU instruction is interrupted and the FPU signals an exception
 - But the main CPU does not check for pending FPU exceptions. It might use a memory value that the interrupted FPU instruction was supposed to set.
 - Example:

```
.data
intVal DWORD 25
.code
fild intVal          ; load integer into ST(0)
inc intVal           ; increment the integer
```

Exception Synchronization

- (continued)
- For safety, insert a `fwait` instruction, which tells the CPU to wait for the FPU's exception handler to finish:

```
.data
intVal DWORD 25
.code
fild intVal      ; load integer into ST(0)
fwait           ; wait for pending exceptions
inc intVal      ; increment the integer
```

FPU Code Example

expression: $\text{valD} = -\text{valA} + (\text{valB} * \text{valC}).$

```
.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
```

Mixed-Mode Arithmetic

- Combining integers and reals.
 - Integer arithmetic instructions such as ADD and MUL cannot handle reals
 - FPU has instructions that promote integers to reals and load the values onto the floating point stack.

- Example: $Z = N + X$

```
.data
N SDWORD 20
X REAL8 3.5
Z REAL8 ?

.code
fild N           ; load integer into ST(0)
fwait           ; wait for exceptions
fadd X           ; add mem to ST(0)
fstp Z           ; store ST(0) to mem
```


Masking and Unmasking Exceptions

- Exceptions are masked by default
 - Divide by zero just generates infinity, without halting the program
- If you unmask an exception
 - processor executes an appropriate exception handler
 - Unmask the divide by zero exception by clearing bit 2:

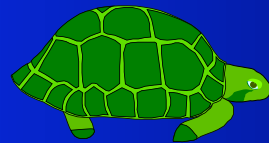
```
.data
```

```
ctrlWord WORD ?
```

```
.code
```

```
fstcw ctrlWord ; get the control word  
and ctrlWord,111111111111011b ; unmask divide by zero  
fldcw ctrlWord ; load it back into FPU
```

The End



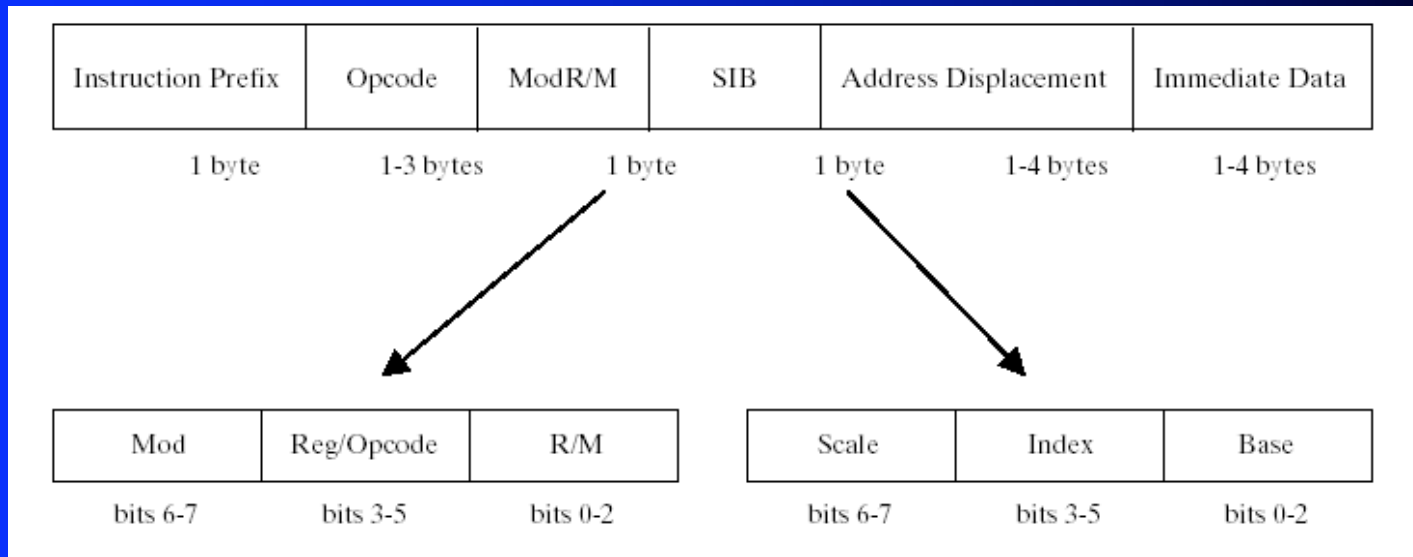
x86 Instruction Encoding

- x86 Instruction Format
- Single-Byte Instructions
- Move Immediate to Register
- Register-Mode Instructions
- x86 Processor Operand-Size Prefix
- Memory-Mode Instructions

x86 Instruction Format

- Fields
 - Instruction prefix byte (operand size)
 - opcode
 - Mod R/M byte (addressing mode & operands)
 - scale index byte (for scaling array index)
 - address displacement
 - immediate data (constant)
- Only the opcode is required

x86 Instruction Format



Single-Byte Instructions

- Only the opcode is used
- Zero operands
 - Example: AAA
- One implied operand
 - Example: INC DX

Move Immediate to Register

- Op code, followed by immediate value
- Example: move immediate to register
- Encoding format: $B8+rw\ dw$
 - (B8 = opcode, $+rw$ is a register number, dw is the immediate operand)
 - register number added to B8 to produce a new opcode

Table 17-21 Register Numbers (8/16 bit).

Register	Code
AX/AL	0
CX/CL	1
DX/DI	2
BX/BL	3
SP/AH	4
BP/CH	5
SI/DH	6
DI/BH	7

Register-Mode Instructions

- Mod R/M byte contains a 3-bit register number for each register operand
 - bit encodings for register numbers:

R/M	Register	R/M	Register
000	AX or AL	100	SP or AH
001	CX or CL	101	BP or CH
010	DX or DL	110	SI or DH
011	BX or BL	111	DI or BH

- Example: MOV AX, BX

mod	reg	r/m
11	011	000

x86 Operand Size Prefix

- Overrides default segment attribute (16-bit or 32-bit)
- Special value recognized by processor: 66h
- Intel ran out of opcodes for x86 processors
 - needed backward compatibility with 8086
- On x86 system, prefix byte used when 16-bit operands are used

x86 Operand Size Prefix


- Sample encoding for 16-bit target:

```
.model small
.286
.stack 100h
.code
main PROC
    mov     ax,dx                ; 8B C2
    mov     al,d1                ; 8A C2
```

- Encoding for 32-bit target:

```
.model small
.386
.stack 100h
.code
main PROC
    mov     eax,edx              ; 8B C2
    mov     ax,dx                ; 66 8B C2
    mov     al,d1                ; 8A C2
```

overrides default
operand size



Memory-Mode Instructions

- Wide variety of operand types (addressing modes)
- 256 combinations of operands possible
 - determined by Mod R/M byte
- Mod R/M encoding:
 - mod = addressing mode
 - reg = register number
 - r/m = register or memory indicator

mod	reg	r/m
00	000	100

MOV Instruction Examples

- Selected formats for 8-bit and 16-bit MOV instructions:

Opcode	Instruction	Description
88/r	MOV eb,rb	Move byte register into EA byte
89/r	MOV ew,rw	Move word register into EA word
8A/r	MOV rb,eb	Move EA byte into byte register
8B/r	MOV rw,ew	Move EA word into word register
8C/0	MOV ew,ES	Move ES into EA word
8C/1	MOV ew,CS	Move CS into EA word
8C/2	MOV ew,SS	Move SS into EA word
8C/3	MOV DS,ew	Move DS into EA word
8E/0	MOV ES,mw	Move memory word into ES
8E/0	MOV ES,rw	Move word register into ES
8E/2	MOV SS,mw	Move memory word into SS

Sample MOV Instructions

Assume that `myWord` is located at offset 0102h.

Instruction	Machine Code	Addressing Mode
<code>mov ax,myWord</code>	A1 02 01	direct (optimized for AX)
<code>mov myWord,bx</code>	89 1E 02 01	direct
<code>mov [di],bx</code>	89 1D	indexed
<code>mov [bx+2],ax</code>	89 47 02	base-disp
<code>mov [bx+si],ax</code>	89 00	base-indexed
<code>mov word ptr [bx+di+2],1234h</code>	C7 41 02 34 12	base-indexed-disp

Summary

- Binary floating point number contains a sign, significand, and exponent
 - single precision, double precision, extended precision
- Not all significands between 0 and 1 can be represented correctly
 - example: 0.2 creates a repeating bit sequence
- Special types
 - Normalized finite numbers
 - Positive and negative infinity
 - NaN (not a number)

Summary - 2

- Floating Point Unit (FPU) operates in parallel with CPU
 - register stack: top is ST(0)
 - arithmetic with floating point operands
 - conversion of integer operands
 - floating point conversions
 - intrinsic mathematical functions
- x86 Instruction set
 - complex instruction set, evolved over time
 - backward compatibility with older processors
 - encoding and decoding of instructions

The End

