

# Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

## Chapter 12: Floating-Point Processing and Instruction Encoding

*Slide show prepared by the author*

*Revision date: 1/15/2014*

(c) Pearson Education, 2014. 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.

# Chapter Overview

- **Floating-Point Binary Representation**
- Floating-Point Unit
- x86 Instruction Encoding

# Floating-Point Binary Representation

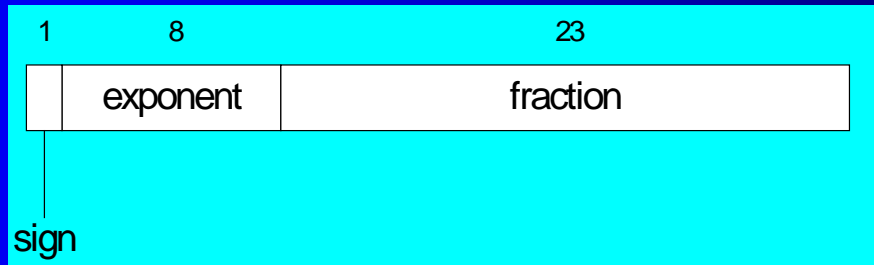
- IEEE Floating-Point Binary Reals
- The Exponent
- Normalized Binary Floating-Point Numbers
- Creating the IEEE Representation
- Converting Decimal Fractions to Binary Reals

# 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.

# Single-Precision Format

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



# Components of a Single-Precision Real

- Sign
  - 1 = negative, 0 = positive
- Significand
  - 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
  - unsigned integer
  - integer bias (127 for single precision)

# Decimal Fractions vs Binary Floating-Point

Binary Floating-Point	Base 10 Fraction
11.11	3 3/4
101.0011	5 3/16
1101.100101	13 37/64
0.00101	5/32
1.011	1 3/8
0.000000000000000000000001	1/8388608

Table 17-3 Binary and Decimal Fractions.

Binary	Decimal Fraction	Decimal Value
.1	1/2	.5
.01	1/4	.25
.001	1/8	.125
.0001	1/16	.0625
.00001	1/32	.03125

# The Exponent

- Sample Exponents represented in Binary
- Add 127 to actual exponent to produce the biased exponent

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



# Normalizing Binary Floating-Point Numbers

- Mantissa 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	$1.1101 \times 2^3$
.000101	$1.01 \times 2^{-4}$
1010001.	$1.010001 \times 2^6$

# 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
    - signaling

# 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 <sup>a</sup>

# 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 10000010 1011010000000000000000
-.00101	124	1 01111100 0100000000000000000000
+100111.0	132	0 10000100 0011100000000000000000
+.0000001101011	120	0 01111000 1010110000000000000000

# Converting Fractions to Binary Reals

- Express as a sum of fractions having denominators that are 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

# 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 101100000000000000000000 to  
Decimal

1. The number is positive.
2. The unbiased exponent is binary 00000011, or decimal 3.
3. Combining the sign, exponent, and significand, the binary number is  $+1.01011 \times 2^3$ .
4. The unnormalized binary number is  $+1010.11$ .
5. The decimal value is  $+10 \frac{3}{4}$ , or  $+10.75$ .

# What's Next

- Floating-Point Binary Representation
- **Floating-Point Unit**
- x86 Instruction Encoding

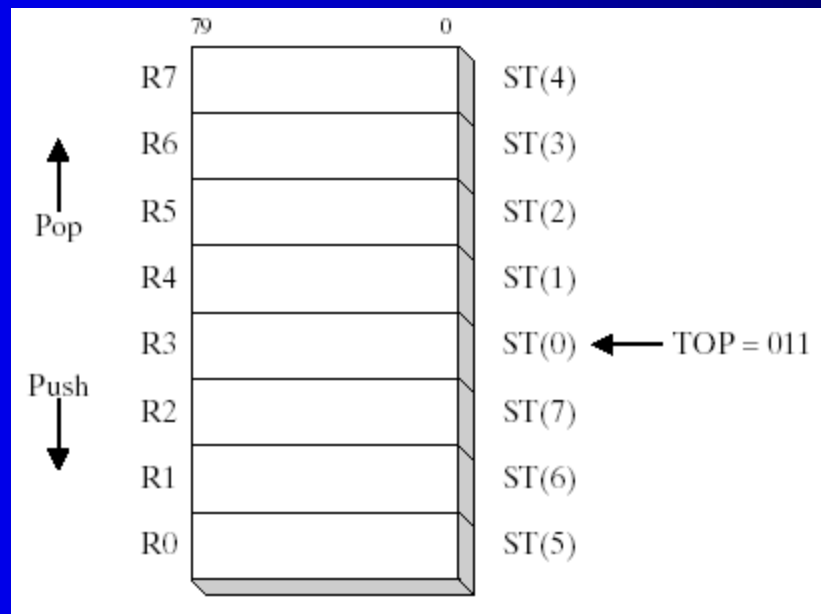


# Floating Point Unit

- FPU Register Stack
- Rounding
- Floating-Point Exceptions
- Floating-Point Instruction Set
- Arithmetic Instructions
- Comparing Floating-Point Values
- Reading and Writing Floating-Point Values
- Exception Synchronization
- Mixed-Mode Arithmetic
- Masking and Unmasking Exceptions

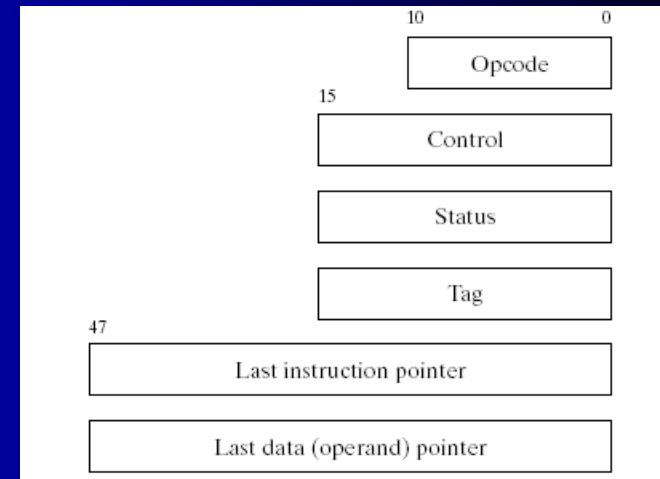
# FPU Register Stack

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



# 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
  - I = integer
  - no letter: floating point
- Examples
  - FLBD    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 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

# 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



# 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  
dblOne    REAL8 234.56  
dblTwo    REAL8 10.1  
.code  
fld  dblOne          ; ST(0) = dblOne  
fld  dblTwo          ; ST(0) = dblTwo, ST(1) = dblOne
```

# 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)
```

# Arithmetic Instructions

- Same operand types as FLD and FST

Table 17-12 Basic Floating-Point Arithmetic Instructions.

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

# Floating-Point Add

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

FADD<sup>4</sup>

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.

# 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 <sup>a</sup>	1	1	1	(None)

<sup>a</sup>If 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:
```

# Floating-Point I/O

- Irvine32 library procedures
  - ReadFloat
    - reads FP value from keyboard, pushes it on the FPU stack
  - WriteFloat
    - writes value from ST(0) to the console window in exponential format
  - ShowFPUStack
    - displays contents of FPU stack

# 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
fld 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
```



# What's Next

- Floating-Point Binary Representation
- Floating-Point Unit
- **x86 Instruction Encoding**

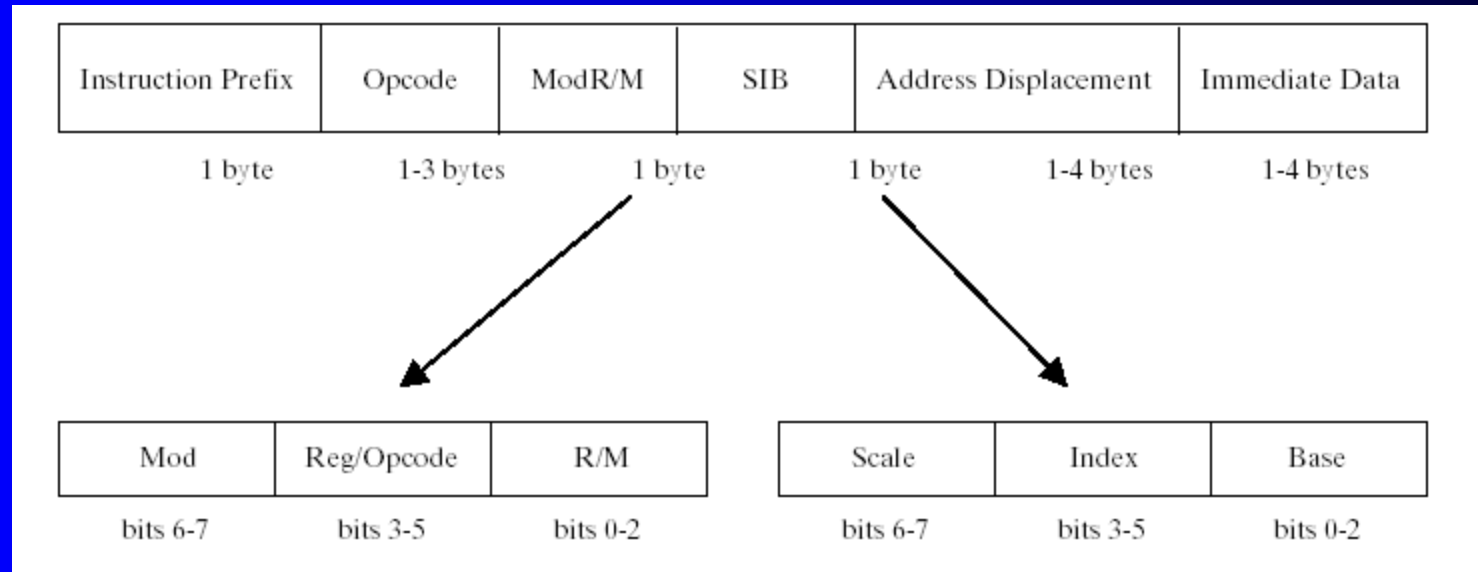
# x86 Instruction Encoding

- 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,dl                ; 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,dl                ; 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

