

# Systems I: Computer Organization and Architecture

## Lecture 12: Floating Point Data

### Floating Point Representation

- Numbers too large for standard integer representations or that have fractional components are usually represented in scientific notation, a form used commonly by scientists and engineers.

- Examples:

$$4.25 \times 10^1$$

$$-3.35 \times 3$$

$$10^{-3}$$

$$-1.42 \times 10^2$$

## Normalized Floating Point Numbers

- We are most interested in normalized floating-point numbers, a format which includes:
  - sign
  - significand ( $1.0 \leq \text{Significand} < \text{Radix}$ )
  - integer power of the radix

## Examples of Normalized Floating Point Numbers

These are normalized:

- $+1.23456789 \times 10^1$
- $-9.987654321 \times 10^{12}$
- $+5.0 \times 10^0$

These are ***not*** normalized:

- $+11.3 \times 10^3$      *significand > radix*
- $-0.0002 \times 10^7$      *significand < 1.0*
- $-4.0 \times 10^{1/2}$      *exponent not integer*

## Converting From Binary To Decimal

$$\begin{aligned} \mathbf{1.00101}_2 &= 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &\quad + 0 \times 2^{-4} + 1 \times 2^{-5} \\ &= 1 + 0/2 + 0/4 + 1/8 + 0/16 + 1/32 \\ &= 1 + 0.125 + 0.03125 \\ &= \mathbf{1.5625} \\ &= 37/32 = 1.5625 \end{aligned}$$

## Converting From Decimal To Binary

Let's start with  $3.4625 \times 10^1 = 34.625$

Let's deal separately with the 34 (which equals  $100010_2$ )

$$2 \times .625 = 1.25 \text{ (save the integer part)}$$

$$2 \times .25 = 0.5 \text{ (no integer part to save)}$$

$$2 \times .50 = 1.00 \text{ (save the integer part)}$$

Let's write them left to right in order:

$$34.625_{10} = 100010.101_2$$

## Converting From Decimal To Binary – Another Example

$$1.23125 \times 10^1 = 12.3125$$

$$12_{10} = 1100_2$$

$$2 \times .3125 = \mathbf{0.625}$$

$$2 \times .625 = \mathbf{1.25}$$

$$2 \times .25 = \mathbf{0.50}$$

$$2 \times .50 = \mathbf{1.0}$$

$$\mathbf{12.3125}_{10} = \mathbf{1100.0101}_2$$

## Normalizing Floating Point Data

Floating point data is normalized so that there is the significand is always one:

$$100001.101_2 = 1.00001101 \times 2^5$$

$$1100.0101_2 = 1.1000101 \times 2^3$$

Since the most significant bit is always 1, we can assume that it is implied and that we do not actually have to represent it.

## Biased Exponents

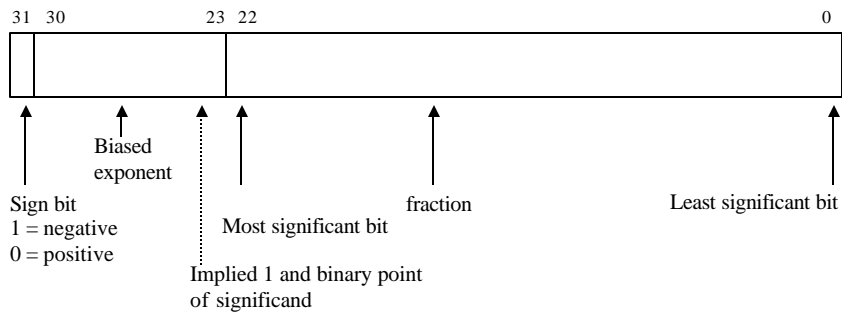
- Short floating point numbers uses 8-bits for the exponents, which we want to range from -128 to +127.
- A biased exponent uses some value other than 0 as the baseline, which must be subtracted to get the actual exponent value.
- Example (in short floating point):
  - exponent 135 =  $135 - 127 = 2^8$
  - exponent 120 =  $120 - 127 = 2^{-7}$

## Representing Floating Point Values In Memory

There are three standard formats for representing floating-point numbers:

- 32-bit format (*single-precision*)
- 64-bit format (*double-precision*)
- 80-bit format (*extended precision*)

# Short Floating Point Numbers



# Representing Values

$$-12.4375_{10} = -1100.0111_2$$

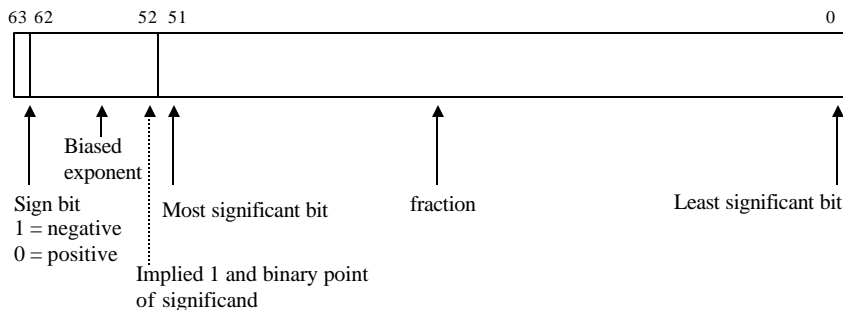
Short:  $-1.10001110000\dots 0000_2 \times 2^3 +127$

1 10000010      10001110000... 0000

$$1100\ 0001\ 0100\ 0111\ 0000\ \dots\ 0000_2$$

$$= \text{C1470000h}$$

# Long Floating Point Numbers



# Representing Values

$$-12.4375_{10} = -1100.0111_2$$

Long:  $-1.10001110000...0000_2 \times 2^3 +1023$

1 10000000010      10001110000...0000

$$1100\ 0000\ 0010\ 1000\ 1110\ 0000\ \dots\ 0000_2$$

$$= \text{C028E00000000000h}$$





## Specifying Floating Point Data In Assembly Language

- We can use the:
  - `dd` (define doubleword) directive to allocate storage for single-precision floats
  - `dq` (define quadword) to allocate storage for double-precision floats and
  - `dt` (define tenbyte) for extended-precision floats.

## Specifying Floating Point Data - An Example

- Allocating storage and initializing values

```
ShortOne dd 1.0
LongOne dq 1.0
Pi dd 0.314159265E1
IntRate dt 13.25E-1
```

Allocating storage without initializing:

```
Mass dd ?
CoefFric dq ?
Temp dt ?
```

## Floating Point Operations

- Floating point operations include:
  - moving and rounding data
  - conversion
  - addition
  - subtraction
  - multiplication
  - division
  - remainder
  - comparison

## Moving Floating Point Data

- Moving floating point data can be done using the standard `mov` instruction in Assembly language.
- If the source and destination are different length, care must be taken in conversion to ensure that exponent and significand are properly converted.

## Data Conversion

- Integer and floating point data cannot be used interchangeably; data conversion is necessary and real-to-integer conversion is not without potential problems:
  - Underflow – a magnitude too small to represent as an integer.
  - Overflow – a magnitude too small to represent as an integer.
  - Inexact result – a loss of all of part of the fractional part of the floating-point fvalue.

## Floating Point Addition

- To add two floating point values, they have to be aligned so that they have the same exponent.
- After addition, the sum may need to be normalized.
- Potential errors include overflow, underflow and inexact results.
- Examples:

$$\begin{array}{r} 2.34 \times 10^3 \\ + \underline{0.88 \times 10^3} \\ \hline 3.22 \times 10^3 \end{array} \qquad \begin{array}{r} 6.22 \times 10^8 \\ + \underline{3.93 \times 10^8} \\ \hline 10.15 \times 10^8 = 1.015 \times 10^9 \end{array}$$

## Floating Point Subtraction

- Subtracting floating point values also requires re-alignment so that they have the same exponent.
- After subtraction, the difference may need to be normalized.
- Potential errors include overflow, underflow and inexact results, and the difference may have one significant bit less than the operands..
- Examples:

$$\begin{array}{r} 2.34 \times 10^3 \\ - 0.88 \times 10^3 \\ \hline 1.46 \times 10^3 \end{array}$$

$$\begin{array}{r} 6.44 \times 10^4 \\ - 6.23 \times 10^4 \\ \hline 0.21 \times 10^4 = 2.1 \times 10^3 \end{array}$$

## Floating Point Multiplication

- Multiplying floating point values does not requires re-alignment - realigning may lead to loss of significance.
- After multiplication, the product may need to be normalized.
- Potential errors include overflow, underflow and inexact results.
- Examples:

$$\begin{array}{r} 2.4 \times 10^{-3} \\ \times 6.3 \times 10^2 \\ \hline 15.12 \times 10^1 = 1.512 \times 10^2 \end{array}$$

## Floating Point Division

- Dividing floating point values does not require re-alignment.
- After division, the (floating point) quotient may need to be normalized – there is no remainder
- Potential errors include overflow, underflow, inexact results and attempts to divide by zero.
- Examples:

$$1.86 \times 10^{13} \div 7.44 \times 10^5 = \begin{array}{l} 0.25 \times 10^8 \\ 2.5 \times 10^7 \end{array}$$

## Floating Point Remainder

- There is usually no remainder in floating point division, because the quotient can be a floating point value itself.
- Sometimes we want a remainder, i.e., the difference between the dividend and the product of the quotient rounded to the nearest integer) and the divisor:
- $s \text{ REM } t = s - t \times \text{NINT}(s/t)$
- Remainder will not produce inexact results, underflow or overflow but can lead to an attempt to divide by zero.

## Floating Point Comparison

- There are usually three results that can happen as a result of floating point comparison:
  - less than
  - greater than
  - equal to
- In some instances, there is a fourth result: unordered, which occurs if one of the values is the result of an arithmetic error.
- These errors can result from adding or subtracting infinite values and are called NaNs (for *Not a Number*).

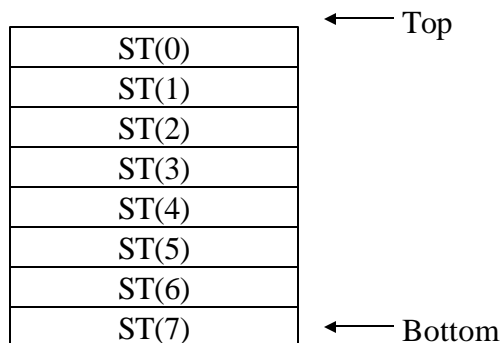
## The Intel Floating Point Co-processors

- Early Intel processors (8088/8086, 80286, 80386) had no floating point capabilities; unless you wished to emulate floating point operations using software routines, you needed to add a co-processor (8087, 80287, 80387).
- 80486 and Pentium family processors include a floating point unit with an architecture that is the same as the coprocessors.

## The Intel Floating Point Unit

- The Floating Point Unit contains 13 registers:
  - Eight 80-bit data registers (ST(0), ST(1), .., ST(7)) that are usually accessed as a stack with ST(0) representing the top of the stack.
  - Three 16-bit registers, called the tag, control and status registers.
  - Two 32-bit registers that serve as exception pointers.

## Data Register



## Tag Register

tag 7	tag 6	tag 5	tag 4	tag 3	tag 2	tag 1	tag 0
-------	-------	-------	-------	-------	-------	-------	-------

tag	meaning
00	valid (finite nonzero number)
01	zero
10	invalid (infinite or NaN)
11	empty

## Control Register

- The control register contains six exception masks and three control fields
- If one of the exception masks is cleared and that exception occurs, the program is suspended and an interrupt is generated, which will either correct the problem or terminate the program.
- The control fields control rounding and the type of infinity used.



## Floating Point Move Instructions

Floating point *move* instructions include:

- **`fld source`** – convert to ext. real and push on stack
- **`fld source`** – convert from integer and push
- **`fst dest`** – store (without popping)
- **`fstp dest`** – store and pop
- **`fist dest`** – convert to integer and store
- **`fistp dest`** – convert to integer, store and pop
- **`fxch`** – exchange ST and ST(1)
- **`fld1`** – push 1 on the stack
- **`fldz`** – push zero on the stack

## Floating Point Arithmetic Instructions

- Floating point *arithmetic* instructions include:
  - **`fadd`**    Add
  - **`fsub`**    Subtract
  - **`fsubr`**    Subtract Reversed (minuend and subtrahend are in reverse order)
  - **`fmul`**    Multiply
  - **`fdiv`**    Divide
- Without an operand, the operands are popped from the stack and the result is pushed.
- With a single operand, the second operand is specified; the first operand the result are on the top of the stack

## Special Floating Point Operations

- In addition to the standard arithmetic operations, there are a few that do not always have integer counterparts:
  - `fchs` - change sign (negation)
  - `fabs` - absolute value
  - `frndint` - round to nearest integer
  - `fsqrt` - square root
- In all cases, the operand is on the top of the stack

## Using Comparison Instructions

- Comparison instructions set the status bits in the FPU control register. To use these as the basis for a conditional branch, we must load these values into the main flag register.
- We use the sequence:

```
fstsw    StatusReg    ; store status word in
                        ; memory
mov ax, StatusReg      ; Copy into AX register
sahf                    ; copy copy from AH to
                        ; the flag register
... ..
StatusReg    dw    ?
```

## Comparison Instructions

- The comparison instructions set the status bits in the FPU control register depending on the top two values in ST.
- The comparison instructions include:
  - **fcom** compares ST with ST(1) or the operand
  - **fcomp** compares ST with ST(1) or the operand (and pops ST)
  - **ficom** compare ST with an integer operand
  - **ficomp** compare ST with an integer operand and pops ST
  - **ftst** compares ST to zero

## Control Instructions

- Control instructions give the programmer control over the 8087 or FPU.
- They include:
  - **fwait** Suspend 8086 (non-FPU) operations until 8087 (or FPU) is not busy
  - **finit** Initialize tag, control and status registers.
  - **fstcw** Save Control Word (where indicated by operand)

## Example: Calculating A Square Root

- Newton's algorithm is frequently used to calculate square root.
- We start with an initial guess that  $X_0$  is the square root of A. We then find a revised guess  $X_1$  where:

$$X_1 = (A/X_0 + X_0) / 2$$

- We repeat this until the difference of  $X_N$  and  $X_{N-1}$  is within an acceptably small.

## The Square Root Procedure

```
; procedure to find the square root using Newton's method
; Can be called by programs in higher languages as well
; Parameters and local values
A          equ     dword ptr [bp + 4]
Xold       equ     dword ptr [bp - 4]
Xnew       equ     dword ptr [bp - 8]
TestResult equ     word  ptr [bp - 10]

.data
Two        dd      2.0
MaxRelErr  dd      0.5E-6

.code
NewSqrt    proc
; set up bp register to point to parameter
           push    bp
           mov     bp, sp
```

```

; allocate stack space for local variables
      sub     sp, 10
; Xnew = 1.0
      fldl
      fstp   Xnew
REPEAT:
; Xold = Xnew
      fld    Xnew
      fst    Xold ; copy of Xold remains on
                  ; stack
; Xnew = (A/Xold + Xold)
      fld    A
      fld    Xold
      fdiv
      fld    Xold
      fadd
      fld    Two
      fdiv
      fst    Xnew ; copy of Xnew remains on
                  ; stack

```

```

; test MaxRelErr * Xnew > abs(Xnew - Xold)
      fsub
      fabs
      fld    MaxRelErr
      fld    Xnew
      fmul
      fcompp

      fstsw  TestResult
      fwait
      mov    ax, TestResult
      sahf
      jna   REPEAT
; UNTIL MaxRelErr * Xnew > abs(Xnew - Xold)

; return Xnew in FPU stack, restore the non-FPU stack
      fld    Xnew
      add    sp, 10
      pop    bp
      ret    4
newsqrt  endp
end      newsqrt

```