

FLOATING POINT: REPRESENTATIONS

- 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
 1. A *sign* s (either 0 or 1)
 2. An *exponent* e
 3. A *mantissa* m (sometimes called a *significand*)
- So that a floating point number N is written as

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

Where the mantissa is normalized such that

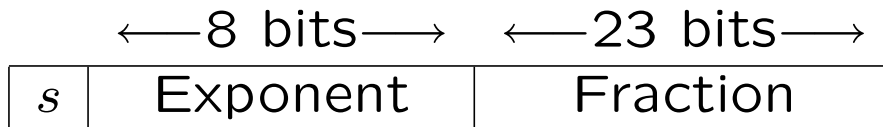
$$1 \leq m < 2$$

Floating Point Representation

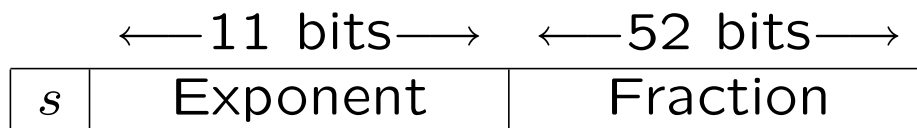
- We can write N in terms of fraction $f : 0 \leq f < 1$

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

- The IEEE-754 standard defines the following formats



Single-Precision (32 bits)



Double-Precision (64 bits)

- Hence, value 1 in $1 + f$ ($= 1.f$) is *not* stored: it is implied!
- Extended precision formats (on 80 bits and more), with more bits for the exponent and fraction are also defined and used inside *FPU*'s for more precision

Representation for the Exponent

- The exponent e uses a biased representation: a (unsigned) number E is stored in the *exponent field* such that $e = E - 011\dots1b$. Hence
 1. $e = E - 127$ for single-precision ($0 \leq E < 255$)
 2. $e = E - 1023$ for double-precision ($0 \leq E < 2048$)
- To obtain E (and the fraction) from a floating point value N , we must first write N as

$$(-1)^s \times (1.f) \times 2^e$$

Where the fraction f must satisfy $0 \leq f < 1$

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

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 MSB of the fraction

- The algorithm to find each bit of a fraction f (ex: $f = .6$)

1. The MSB of the fraction is 1 iff $f \geq \frac{1}{2}$. Hence the MSB = 1 iff $2f \geq 1$

2. Let f' be the fraction part of $2f$. Then the next MSB of f is 1 iff $2f' \geq 1$

3. Let f'' be the fraction part of $2f'$. Then the next MSB of f is 1 iff $2f'' \geq 1$

4. ... and so on

- Example: find all bits of $f = 0.15$

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	

Hence $0.15 = 0.00\overline{1001}b = 0.00100110011001\dots b$. When truncation is used, the following 23 bits will be stored in the single-precision fraction field: 00100110011001100110011

Defining Floating Point Values in ASM

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

```
Float1 DD 17.15      ;single-precision float
Float2 DD 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

1. $17 = 10001b$ and $0.15 = 0.001001b$

2. $17.15 = 10001.001001b = 1.0001001001b \times 2^4$

3. 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
```

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

```
Double1 DQ 0.001235 ;double-precision float
Double1 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 $\frac{1}{3}$ 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 digits to represent a fraction, should $\frac{2}{3}$ be represented as 0.666 or 0.667?

- Truncation is only one of the methods used for rounding. Three other methods are supported by IEEE
 1. Round to the nearest number (default IEEE)
 2. Round towards $+\infty$
 3. Round towards $-\infty$
- 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

Representation of Specific Values

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

- Recall that the exponent e uses a biased representation. It is represented by unsigned integer E such that $e = E - 0111\dots1b$
- Let F be the unsigned integer obtained by concatenating the bits of the fractions f
- Hence, a floating point number N is represented by (s, E, F) and the the "1" is implied (not represented)
- Then note that we have no representation of 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 (Continued)

- Hence the IEEE standard has reserved the following interpretation when E contains only ones

1. $+\infty$ when $s = 0, E = 111 \dots 1$ and $F = 0$

2. $-\infty$ when $s = 1, E = 111 \dots 1$ and $F = 0$

3. Not a number (NaN) when $E = 111 \dots 1, F \neq 0$

Hence, *normal* floating point values exist only for $E < 111 \dots 11$

- The $\pm\infty$ value arises only when a computation gives a number that would require $E > 111 \dots 11$

- The $\pm\infty$ value can be used in operands with predictable results

1. $+\infty + N = +\infty$

2. $-\infty + N = -\infty$

3. $+\infty + +\infty = +\infty$

- Undefined values are represented by NaN

1. $+\infty + -\infty = \text{NaN}$

2. $\frac{\pm\infty}{\pm\infty} = \text{NaN}$

3. $\frac{0}{0} = \text{NaN}$

Denormalized Numbers

- Now, the smallest non-zero magnitude would be $E = 0$ and $F = 00 \dots 01$. This would give a value of $1.00 \dots 01 \times 2^{-127}$ 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"

1. Hence, the smallest non-zero single-precision denormalized number is

$$0.00 \dots 01 \times 2^{-127} = 2^{-23} \times 2^{-127} = 2^{-150}$$

2. The largest single-precision denormalized number is then

$$2^{-127} \times (1 - 2^{-23})$$

- Hence, normal numbers, called *normalized numbers*, use E such that $0 < E < 11 \dots 1$

The smallest (positive) single-precision normalized number is then

$$1.00 \dots 0 \times 2^{-126}$$

Summary of IEEE Floating Point Numbers

- Each number is represented by (s, E, F)
 1. s represents the sign of the number
 2. The exponent e of the number is $e = E - 011 \dots 1b$
 3. F is the binary number obtained by concatenating the bits of the fraction

- Normalized numbers have: $0 < E < 11 \dots 1$

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 $E = F = 0$

- $s\infty$ is represented by $E = 11 \dots 1$ and $F = 0$

- NaN is represented by $E = 11 \dots 1$ 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. 1.0
2. 0.5
3. -83.7
4. $1.1\text{E}-41$

Note that you may try to verify your results by first storing these values in a variable defined by a DD directive and then moving the variable content into a register. However, note that rounding to the nearest is used instead of truncation ...

Exercise #2 Give the decimal value represented by the IEEE single-precision representation given below in hexadecimal

1. 45AC0000h
2. C4800000h
3. 3FE00000h

FPU: The Floating Point Unit

- *FPU*, designed to perform efficient computation with floating point numbers, is built (directly) on the Pentium processors
 1. It is backward compatible with older numerical co-processors that were provided on a separate chip (ex: 8087 up to 80387)
 2. Use the .387, .487, .587 or .687, ... to enable assembly of FPU/co-processor 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 FPU's conform to the IEEE-754 standard)

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

	79	78	64	63	0	Tags	
ST(0)	s	Exponent		Fraction			tag(0)
ST(1)							tag(1)
ST(2)							tag(2)
ST(3)							tag(3)
ST(4)							tag(4)
ST(5)							tag(5)
ST(6)							tag(6)
ST(7)							tag(7)

The Tag Register

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