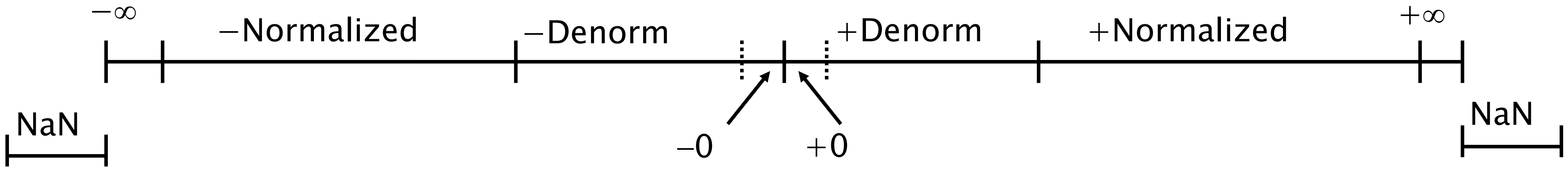
CSE 2421

IEEE 754 Floating Point (Part II)

IEEE 754 types of real number values

- IEEE uses three different encodings to represent real values:
  - 1. **Denormalized values:** These are *extremely* small values, very close to 0, including 0. If such a value becomes too small, it can cause *underflow* (the largest denormalized value is approximately $+/-1.18\times10^{-38}$).
  - 2. **Normalized values:** These are values greater than the largest denormalized value, but small enough to fit into the representation (that is, without overflow).
  - 3. **Special values:** These are values which are either +/- infinity (created by overflow) or NaN (not a number – values which are not real numbers, possibly caused by invalid operations, such as division by 0).
- We will primarily look at conversion for normalized values, but also briefly discuss the others.

Visualization: Floating Point Encodings



S, E, and F fields for different types of values

Type of value	S	E	F	"hidden" bit
0.0	0 or 1	all 0's	all 0's	0
denormalized	0 or 1	all 0's	not all 0's	0
normalized	0 or 1	>0, but not all 1's	any bit pattern	1
+/- infinity	0 or 1	all 1's	all 0's	
NaN*	0 or 1	all 1's	anything but all zeros	

*Not a Number

Denormalized Values

Were created for one primary purpose: **gradual underflow**. It's a way to keep the relative difference between tiny numbers small. If you go straight from the smallest normal number to zero (*abrupt underflow*), the relative change is infinite. If you go to denormals on underflow, the relative change is still not fully accurate, but at least more reasonable. And that difference shows up in calculations.

To put it a different way. Floating-point numbers are not distributed uniformly. There are always the same amount of numbers between successive powers of two: 2^{52} (for double precision). So without denormals, you always end up with a gap between 0 and the smallest floating-point number that is 2^{52} times the size of the difference between the smallest two numbers. Denormals fill this gap uniformly.

As an example about the effects of abrupt vs. gradual underflow, look at the mathematically equivalent x == y and x - y == 0. If x and y are tiny but different and you use abrupt underflow, then if their difference is less than the minimum cutoff value, their difference will be zero, and so the equivalence is violated.

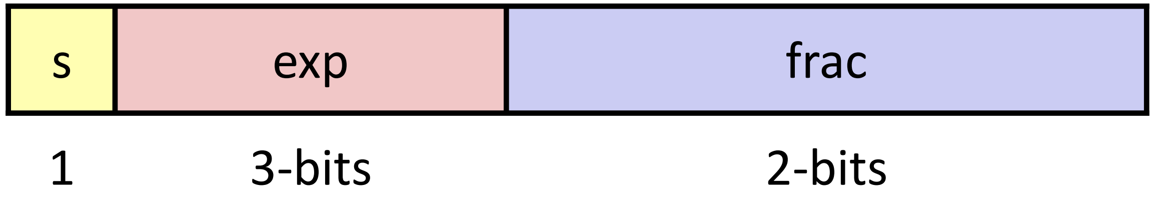
With gradual underflow, the difference between two tiny but different normal numbers gets to be a denormal, which is still not zero. The equivalence is preserved.

So, **using denormals on purpose is not advised, because they were designed only as a backup mechanism in exceptional cases**.

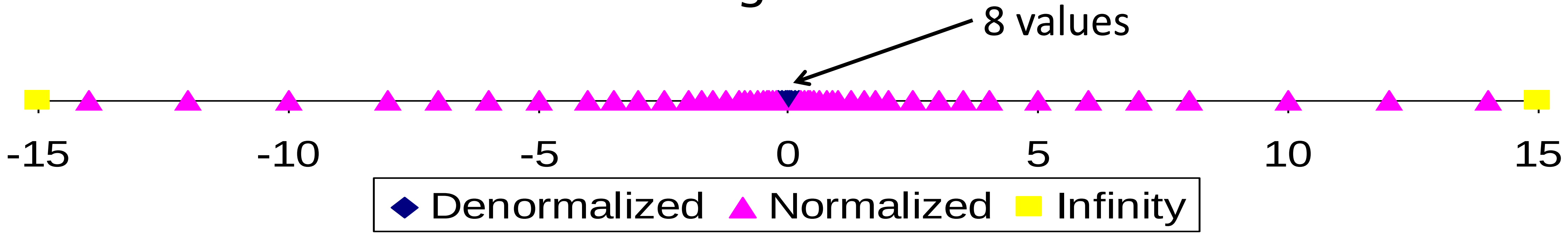
Reference:http://stackoverflow.com/questions/15140847/denormalized-numbers-ieee-754-floating-point

Distribution of Values

- 6-bit IEEE-like format (fewer bits makes things easier to see)
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$

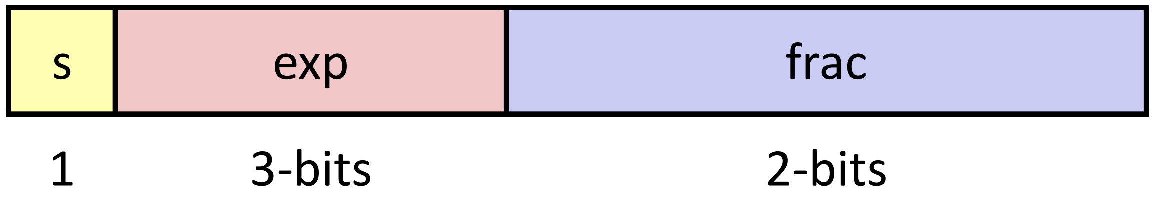


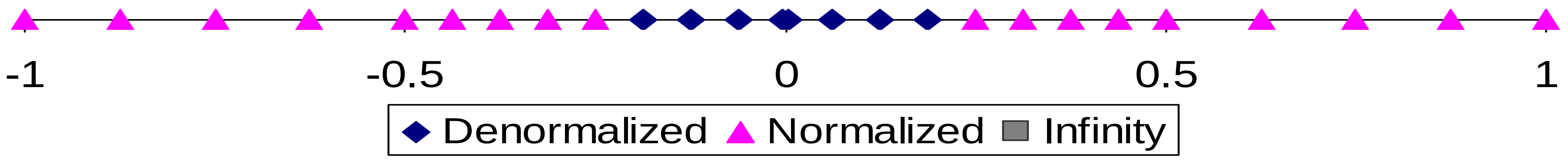
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3





What are the trade-offs in floating point?

- Can’t use integers for everything
- Trying to cover a much broader range of real values; but something has to give, and it’s the precision
- Pi a good example:
  - Whether or not a rational number has a terminating expansion depends on the base.
    - For example, in base-10 the number 1/2 has a terminating expansion (0.5) while the number 1/3 does not (0.333...).
    - In base-2 only rational numbers with denominators that are powers of 2 (such as 1/2 or 3/16) are terminating. Any rational number with a denominator that has a prime factor other than 2 will have an infinite binary expansion.

Range and Precision for Normalized Numbers

- Next, we look somewhat more at the range and precision of IEEE single precision values
- To begin, let’s see the range of values which can be encoded by the F field of a single precision number

Range of F for Normalized Numbers

- Minimum value which can be encoded:
  - 1.0000 0000 0000 0000 0000 000

- Clearly, this = 1.0

Range of F for Normalized Numbers (cont)

- Maximum value which can be encoded:
  - 1.1111 1111 1111 1111 1111 111

$= \quad 1 + ((2^{23} - 1) / (2^{23}))$

$= \quad 1 + (((1024 * 1024 * 8) - 1) / (1024 * 1024 * 8))$

$= \quad 1 + 0.99999988$

$= \quad 1.99999988$

IEEE 754 Single Precision Range

Using these values, we can see the range of values which can be represented in IEEE single precision for various exponents

- Please see the following pdf file on Piazza:

  IEEE_754_Single_Precision_Range_Table.pdf

Floating Point Precision

- Question: It is usually said that floating point values are an approximation to real numbers, but does the floating point representation always allow for a fractional part of the value? (In other words, are there values which can only be encoded in IEEE 754 as integers?)
- Intuitively, since we have a limited number of bits for F, we should guess that the answer is that a fractional portion of a value cannot always be encoded.
- Let's look at single precision.

Floating Point Precision

- Question: Can we determine the minimum value in IEEE single precision that does not allow for the representation of any fractional portion of the value? In other words, the minimum value which can only represent an integer value?

Does this ever print? When?

```
int main()
{
        float f1 = 0.0, f2 = 1.0;
        int i = 1000000;

        do
        {
                i++;
                f1 = i;
                f2 = i+1;
        } while (f1 != f2);

    printf("Stopped at %d (0x%X):  f1 = %f and f2 = %f\n", i, i, f1, f2);

}
```

Floating Point Precision

- At what point are all the bits in the F field required to encode the integer portion of the value, leaving no bits for any fractional portion?

- Notice on slide 6 & 7 that the gap between numbers gets larger every time the exponent increases. With enough exponent bits, eventually that gap will be greater than one.

Floating Point Precision

- Here is the encoding (assume positive sign):
- 0 1001 0110 0000 0000 0000 0000 0000 000
- This value is 1.0000 0000 0000 0000 0000 000 X 2^{23}

It does print!

```
Stopped at 16777216 (0x1000000):  f1 =
16777216.000000 and f2 = 16777216.000000
```

There are 6 zeroes in that hex number, 4 bits each.

At 2^{24} we can’t tell integer values apart with single precision floats

Floating Point Precision

- We have seen that in some cases, the encoding allows for values which are no more precise than integers (i.e., only integer values can be encoded).
- Question: Are there cases where IEEE 754 encoded values are even *less precise* than integers?

Floating Point Precision

- Consider:

  $1.00000000000000000000000 * 2^{30} =$

  1,073,741,824.0 decimal

  and

  $1.00000000000000000000001 * 2^{30} =$

  1,073,741,952.0 decimal

That’s a difference of 128!!

(Remember that int: -2^{31} to $2^{31} - 1$)

Floating Point Precision

- Consider:

  $1.00000000000000000000000 * 2^{50} =$

  1125899906842624.0 decimal

  and

  $1.00000000000000000000001 * 2^{50} =$

  1125900041060352.0 decimal

That's a difference of 134,217,728!!

Floating point example

- Put the decimal number 76.875 into the IEEE standard single precision floating point representation.
- This value will be a normalized value, because it is clearly larger than the largest denormalized value (it is not a value less than 1, but very close to 0), and it is neither infinity nor NaN
- **5 step process** to obtain Sign(S), Exponent(E), and Mantissa(F)

Floating point example

Step 1:

Get a binary representation for 76.875, with at least the number of bits for the F field + 1 (24 for single precision, or 53 for double precision). To do this, get unsigned binary representations for the stuff to the left and right of the decimal point separately.

76 is 1001100:
(divide by 2)

2	**76**	**0**
2	**38**	**0**
2	**19**	**1**
2	**9**	**1**
2	**4**	**0**
2	**2**	**0**
2	**1**	**1**

.875 can be gotten using the algorithm:

	Whole number part
.875 x 2 = 1.75	1
.75 x 2 = 1.5	1
.5 x 2 = 1.0	1
.0 x 2 = 0.0	0

.875 is .1110

Start with the top numbers, work outwards from the binary point:

So, a binary representation for 76 is 1001100 and .875 is .111000000…

Putting the halves back together again, and writing 24 bits:

76.875 is 1001100.11100000000000000

Floating point example

Step 2:

Normalize the binary representation. (i.e. Make it look like scientific notation) with a single 1 to the left of the binary point (for now – more on this later):

1.00110011100000000000 x 2^6

NOTE: the bit to the left of the binary point will be considered the 'bit' (it does not appear in the encoding, but is ***implied by the value of the exponent***)

Floating point example

Step 3:

6 is the true exponent (e). For the standard form, to get E, it needs to be in 8-bit, biased-127 representation (add 127 to the true exponent to get the biased-127 representation).

```
    6
+ 127
-----
  133
```

Convert this decimal value to binary: 133 in 8-bit, unsigned representation is 1000 0101

This is the bit pattern used for E in the standard form.

Floating point example

Step 4:

The mantissa/significand stored (F) is the stuff to the right of the radix point in the normalized form (**1.00110011100000000000000** x 2^6)

We need **23 bits** of it for single precision.

0011 0011 1000 0000 0000 000

Floating point example

Step 5: Put it all together (and include the correct sign bit: the original number is positive, so the sign bit is 0):

```
S       E                    F
0   1000 0101   0011 0011 1000 0000 0000 000
```

```
0b 0100 0010 1001 1001 1100 0000 0000 0000
0x   4    2    9    9    C    0    0    0
```

Floating point example

- Given binary 1100 0110 1000 0110 0010 1100 0110 0000

S = 1

E = 1000 1101 = 141 = 127+14 → e = 14

F = 000 0110 0010 1100 0110 0000

f = 1.000 0110 0010 1100 0110 0000 x 2^{14}

1000 0110 0010 110.0 0110 0000 = – 17,174.1875