Floating Point

CSCI 237: Computer Organization
6th Lecture, Feb 21, 2025

Jeannie Albrecht

Administrative Details

- All twelve puzzles due next Tue/Wed at midnight
  - Questions?
- The last few are tricky, but fun
  - Come see me and/or TAs for tips if necessary
- If you are getting a parse error with dlc, fix it before submitting
  - Usually happens when you don't declare variables at the top of your functions
  - Autograder may fail you
- Glow HW 1 due today
- HW 2 will be posted Sunday/Monday

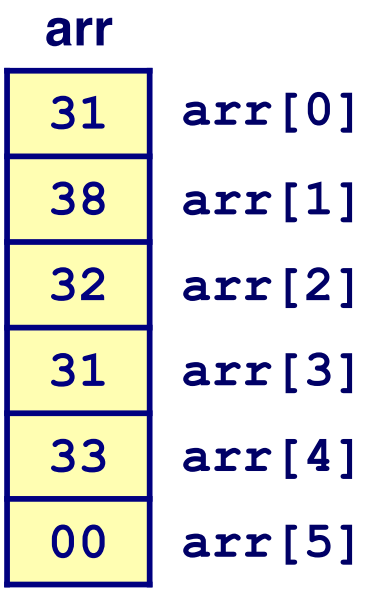
Last Time: Pointers

- Pointers, arrays, and strings in C (we'll finish strings today!)
- Using & and * correctly
- `printf` using
  - `%s: string`
  - `%c: char`
  - `%p: pointer (address)`
  - `%d: decimal`
  - `%x: hexadecimal`
  - `%lx: long hexadecimal`
  - `%llx: long long hexadecimal`
  - (see example in a few slides)

Recap: Representing Arrays In Memory

- Each array element is the size of the specified data type
- Array elements laid out in memory in *contiguous* memory locations
- Array name also refers to the *address of* the first element in the array
  - `arr == &arr[0]`
- Any operation done with an array can also be done with pointers (aka pointer arithmetic)
  - `int *ptr = arr;`
  - `ptr == &arr[0]`
  - `(ptr+1) == &arr[1]`

```
int arr[6];
```



```
int main(int argc, char *argv[]) {
  int arr[4];

  arr[0] = 0;
  arr[1] = 1;
  arr[2] = 2;
  arr[3] = 3;

  int *ptr = arr;
  // Above equivalent to
  // int *ptr;
  // ptr = arr;

  printf("Addresses: \n");
  printf("\t arr     \t %llx \n", (unsigned long long) arr);
  printf("\t &arr[0] \t %llx \n", (unsigned long long)(&arr[0]));
  printf("\t ptr     \t %llx \n", (unsigned long long) ptr);
  printf("\t &ptr    \t %llx \n", (unsigned long long) &ptr);

  return 0;
}
```

Addresses:

arr	7ffee62258a0
&arr[0]	7ffee62258a0
ptr	7ffee62258a0
&ptr	7ffee6225888

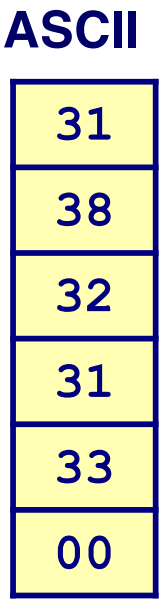
Summary: Why use pointers?

- Modular programming
  - Allows you to pass the memory address of a local variable to another function
    - Allows other function to read and update variable so that modifications are reflected in function where variable created
  - Avoids need to create a copy of the variable contents (so less memory is used)
- Enables single name for collection of elements/memory addresses (i.e., array)
  - Only need one pointer/array name to access all of the elements of an array
- More on this later!

Representing Strings

```
char S[6] = "18213";
```

- Strings in C
  - Represented by array of characters
  - Each character encoded in **ASCII format**
    - ASCII is a character encoding standard that represents text as numbers
    - Standard 7-bit binary encoding of character set
    - Character “0” has code 0x30
      - Digit *i* has code 0x30+*i*
  - String should always be **null-terminated**
    - Final character = 0
- Compatibility
  - Byte ordering (endian-ness) not an issue



```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```

ptr:

```
arr Donuts!
```

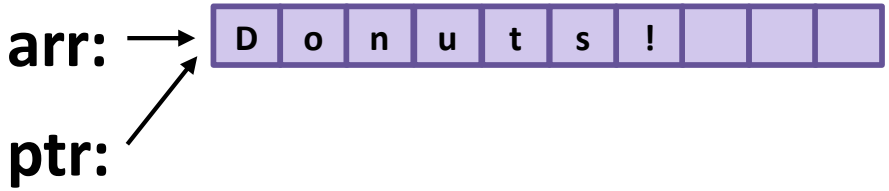
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!
```

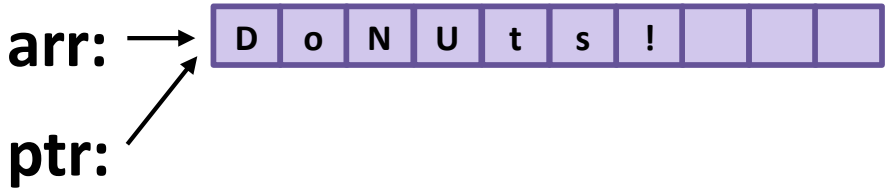
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!
```

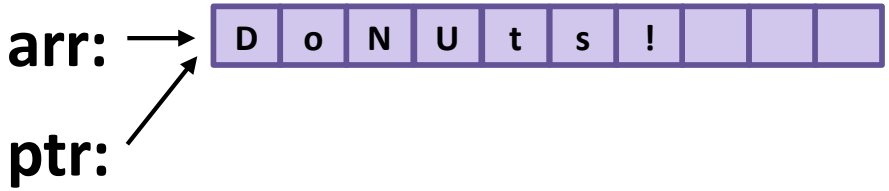
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!


After uppercase
arr DoNUts!
```

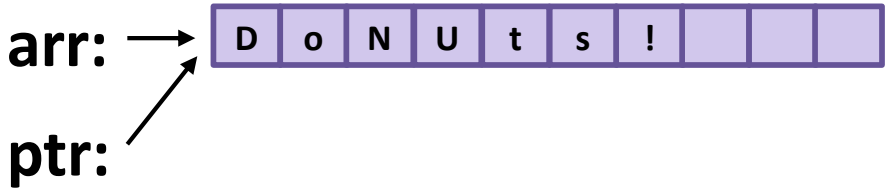
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!


After uppercase
arr DoNUts!
ptr DoNUts!
```

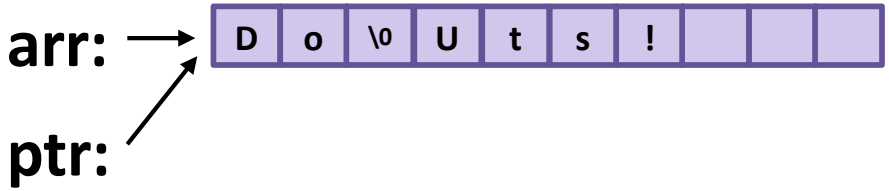
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!


After uppercase
arr DoNUts!
ptr DoNUts!
```

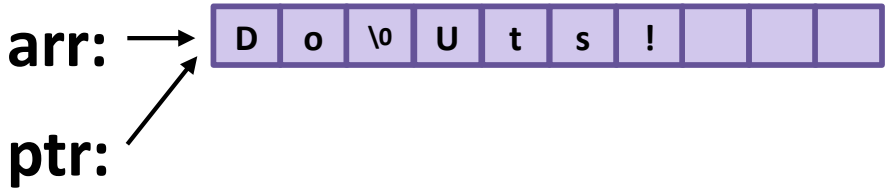
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!


After uppercase
arr DoNUts!
ptr DoNUts!


After null terminator
arr Do
```

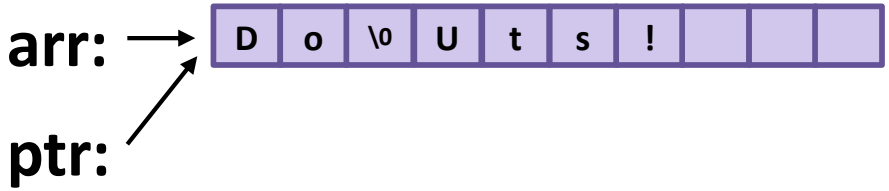
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  ptr = arr;
  printf("ptr %s\n", ptr);

  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  printf("\nAfter null terminator\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```




```
arr Donuts!

ptr Donuts!


After uppercase
arr DoNUts!
ptr DoNUts!


After null terminator
arr Do
ptr Do
```

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char arr[10];
  char *ptr = NULL;
  strcpy(arr, "Donuts!");

  printf("arr %s\n", arr);
  printf("address of arr %p\n", &arr);
  printf("ptr string %s\n", ptr);
  printf("ptr pointer %p\n", ptr);
  printf("address of ptr %p\n", &ptr);

  ptr = arr;
  printf("\nAfter ptr = arr\n");
  printf("arr %s\n", arr);
  printf("address of arr %p\n", &arr);
  printf("ptr string %s\n", ptr);
  printf("*ptr char %c\n", *ptr); //same as arr[0]!
  printf("ptr pointer %p\n", ptr);
  printf("address of ptr %p\n", &ptr);


  arr[2] = 'N';
  ptr[3] = 'U';
  printf("\nAfter uppercase\n");
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);

  *(ptr+2) = '\0';
  *ptr = 'B';
  printf("\nAfter null terminator\n");
  printf("arr+2 %s\n", arr+3);
  printf("arr %s\n", arr);
  printf("ptr %s\n", ptr);
}
```

```
jeannie@rugger9-2021 slides % ./donuts
arr Donuts!
address of arr 0x16b25b7ce
ptr string (null)
ptr pointer 0x0
address of ptr 0x16b25b7b8

After ptr = arr
arr Donuts!
address of arr 0x16b25b7ce
ptr string Donuts!
*ptr char D
ptr pointer 0x16b25b7ce
address of ptr 0x16b25b7b8

After uppercase
arr DoNUts!
ptr DoNUts!

After null terminator
arr+2 Uts!
arr Bo
ptr Bo
```

Rule of thumb about pointers:

After declaring a pointer, only use the * again when trying to "dereference" or "follow" the pointer

Today: IEEE Floating Point

- **Recap: Fractional binary numbers**
- IEEE floating point standard
- Properties
- Floating point in C
- Summary
- Intro to machine level programming (Ch 3)

Recap: Fractional binary numbers

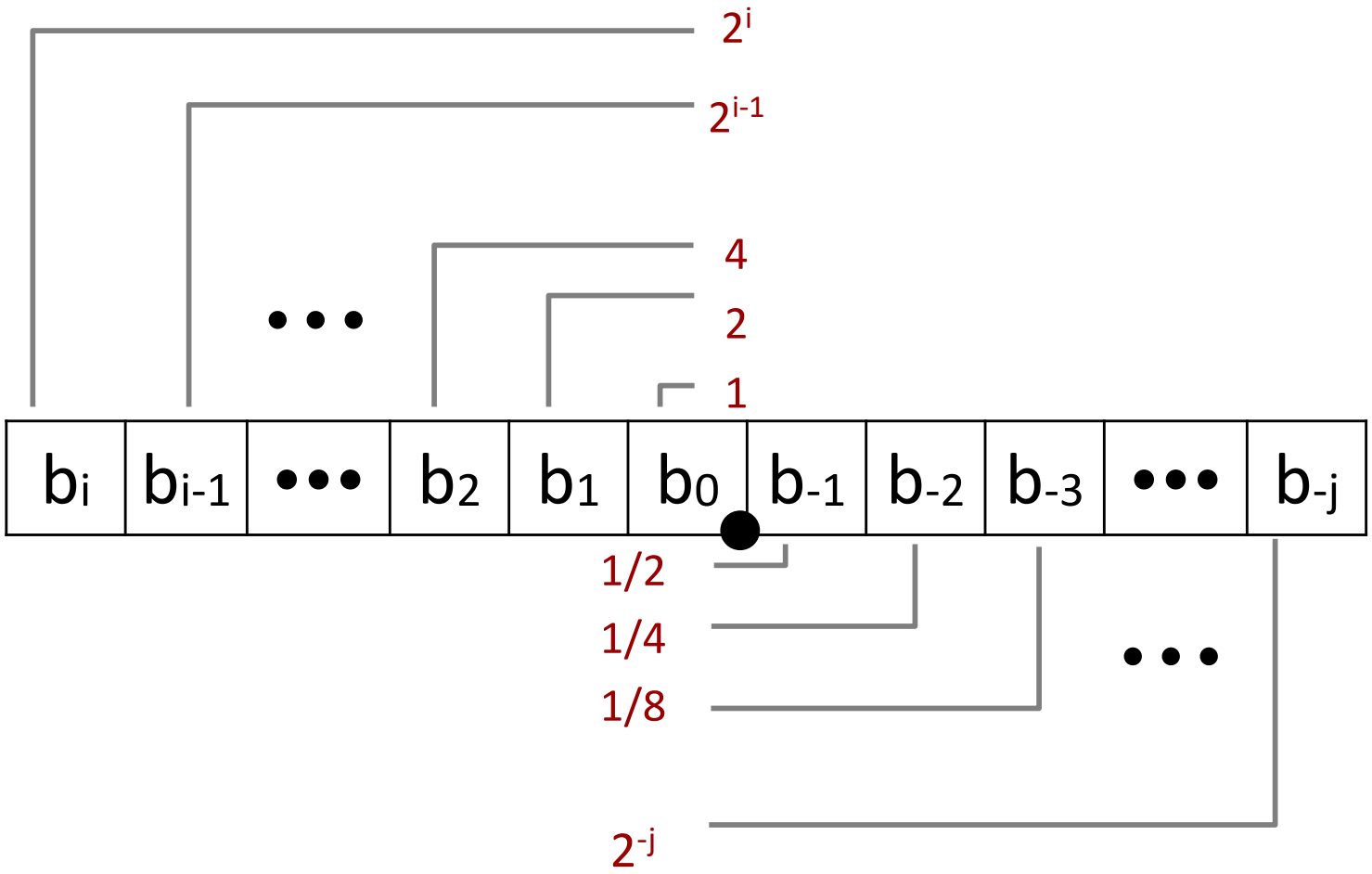
- What is decimal representation of 1011.101_2?

  $2^3 2^2 2^1 2^0 . 2^{-1} 2^{-2} 2^{-3}$

  - $(2^3 * 1) + (2^2 * 0) + (2^1 * 1) + (2^0 * 1) + (2^{-1} * 1) + (2^{-2} * 0) + (2^{-3} * 1) = 11.625$

    $\frac{1}{2} + \frac{1}{4} + \frac{1}{8}$

Fractional Binary Numbers



- Bits to right of "binary point" represent *fractional* powers of 2
- Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

Fractional Binary Numbers: Examples

- Value — Representation

Value	Representation
$23/4 = 5\ 3/4 = 4 + 1 + \frac{1}{2} + \frac{1}{4}$	101.11_2
$23/8 = 2\ 7/8 = 2 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}$	10.111_2
$23/16 = 1\ 7/16 = 1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}$	1.0111_2

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - If we use notation $1.0 - \varepsilon$, then adding more bits brings ε closer and closer to 0

Fractional Binary Number Limitations

- Limitation #1
  - Can only **exactly** represent fractional numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations
  - Value Representation
    - 1/3 $0.0101010101[01]\ldots_2$
    - 1/5 $0.00110011001[0011]\ldots_2$
    - 1/10 $0.000110011001[0011]\ldots_2$

- Limitation #2
  - If we standardize representation, we would have one fixed location for binary point within the *w* bits
    - Limited range of numbers (very small values? very large?)

IEEE Floating Point to the Rescue!

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - IEEE 754 is supported by all major CPUs

- Driven by numerical concerns (aforementioned limitations)
  - Nice standards for rounding, overflow, underflow
  - Problem: Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

- Result: expressive (but complex!) format for encoding fractional numbers.

Floating Point Representation

Example:
$15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}$

- Numerical Form:

$$(-1)^s * M * 2^E$$

  - Sign bit s determines whether number is negative or positive
  - Significand (or mantissa) M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

- Encoding
  - s field *encodes* sign bit s (0 for positive, 1 for negative value)
  - exp field *encodes* E (but is **not equal** to E)
  - frac field *encodes* M (but is **not equal** to M)

FP precision options

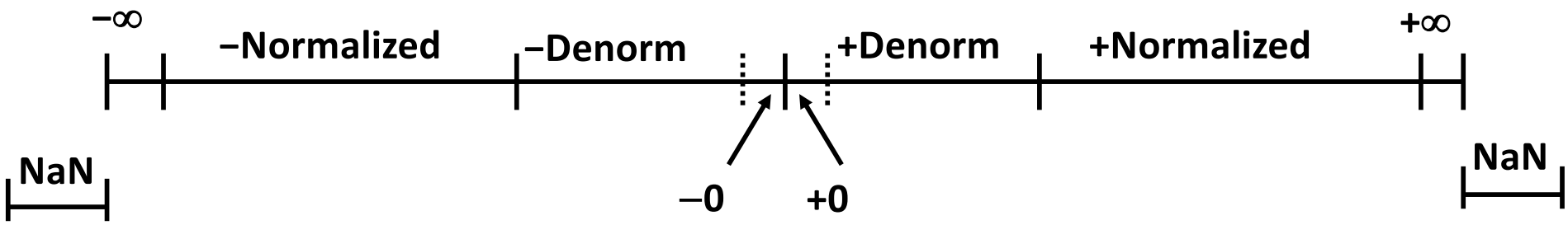
- Single precision: 32 bits
  ≈ 7 decimal digits, $10^{\pm 38}$

- Double precision: 64 bits
  ≈ 16 decimal digits, $10^{\pm 308}$

- Other formats: half precision, quad precision

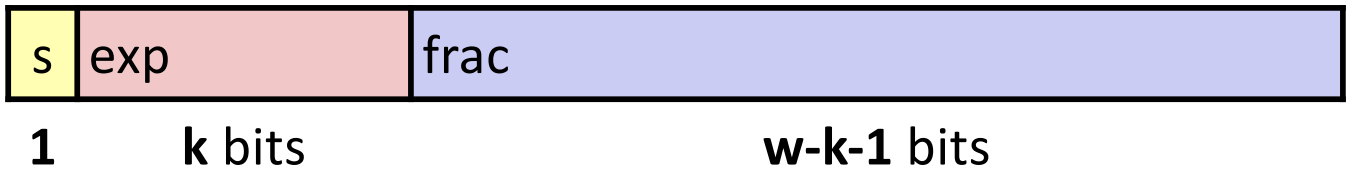
3 “Cases” in Floating Point Format

- Special values: infinity, negative infinity, and NaN
- So-called “normalized” form = “normal numbers”
- So-called “denormalized” form = “numbers that are very close to 0”



"Normalized" Values

$v = (-1)^s \times M \times 2^E$



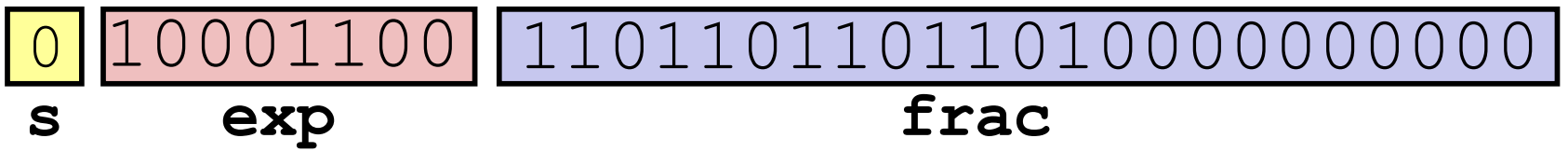
- Condition: **exp ≠ 000…0** and **exp ≠ 111…1**
- Exponent encoded as a biased value: E = *exp* – bias
  - *exp*: unsigned value of *exp* field
  - *bias* = $2^{k-1} - 1$, where k is number of exponent bits
    - Single precision: 127 (exp: 1…254, E: -126…127)
    - Double precision: 1023 (exp: 1…2046, E: -1022…1023)
- Significand encoded with implied leading 1: M = $1.xxx\ldots x_2$
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Note: Get extra leading bit for "free" with implicit leading 1

Normalized Encoding Example

$v = (-1)^s \times M \times 2^E$
$E = exp - bias$

- Value: `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= (-1)^0 \times 1.1101101101101_2 \times 2^{13}$
- Sign: S
  S = 0
- Significand: M
  M = 1.1101101101101_2
  frac = $11011011011010000000000_2$
- Exponent: E
  E = 13
  bias = $2^{k-1}-1 = 2^{8-1}-1 = 127$ (single precision)
  exp = $13 + 127 = 140 = 10001100_2$

$E = exp - bias$
$E = exp - 127$
$exp = E + 127$



Normalized Decoding Example

$v = (-1)^s \times M \times 2^E$
$E = exp - bias$

- Sign: S

  S = 0

- Significand: M

  frac = $01100110101010110000000_2$

  M = $1.01100110101010110000000_2$

- Exponent: E

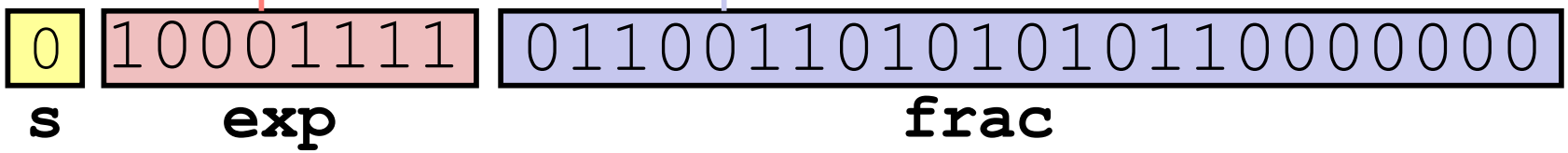
  bias = $2^{k-1}-1 = 2^{8-1}-1 = 127$

  exp = $10001111_2 = 143$

  E = 143 – 127 = 16

E = *exp* – bias

$(-1)^0 \times 1.01100110101010111 \times 2^{16} = 91819$



Denormalized Values

$v = (-1)^s \times M \times 2^E$
$E = 1 - bias$

- Condition: **exp = 000…0**
- Exponent value: E = 1 – Bias
- Significand M coded with implied leading 0: M = 0.xxx…x_2
  - **xxx…x**: bits of **frac**
- Cases
  - **exp** = 000…0, **frac** = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0 😩
  - **exp** = 000…0, **frac** ≠ 000…0
    - Numbers *closest to 0.0*
    - Equi-spaced

Special Values

- Condition: **`exp` = `111…1`**

- Case: **`exp` = `111…1`, `frac` = `000…0`**
  - **Represents value ∞ (infinity)**
  - Operation that overflows
  - Both positive and negative (depending on sign bit)
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- Case: **`exp` = `111…1`, `frac` ≠ `000…0`**
  - **Not-a-Number (NaN)**
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard
- Properties
- Floating point in C
- Summary
- Intro to machine code?

Special Properties of the IEEE Encoding

- FP Zero Same as Integer Zero
  - All bits = 0
  - Yay

- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield? 🤷‍♀️
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

FP Rounding Properties: Basic Idea

- $x +_f y = \text{Round}(x + y)$
- $x \times_f y = \text{Round}(x \times y)$
- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly **overflow** if exponent too large
    - Possibly round to fit into `frac`

Rounding

- Rounding Modes (illustrated with $ rounding as in book)

	$1.40	$1.60	$1.50	$2.50	–$1.50
Towards zero	$1↓	$1 ↓	$1 ↓	$2 ↓	–$1 ↑
Round down (–∞)	$1↓	$1 ↓	$1 ↓	$2 ↓	–$2 ↓
Round up (+∞)	$2↑	$2 ↑	$2 ↑	$3 ↑	–$1 ↑
Nearest even (default)	**$1↓**	**$2 ↑**	**$2 ↑**	**$2↓**	**–$2 ↓**

Nearest even tries to find “closest” whole dollar amount.
What happens when halfway in between two values?

Closer Look at Round-To-Even

- Default Rounding Mode
  - Hard to get any other kind of rounding without dealing with assembly code
  - All other rounding schemes are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that **least significant digit** is even
  - E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" when least significant bit is **0**
  - "Half way" when bits to right of rounding position = **100**…$_2$
- Examples
  - Round to nearest 1/4 (aka 2 bits right of binary point)

$Value_{10}$	Binary	Rounded	Rounding Direction	Rounded $Value_{10}$
2 3/32	10.00011_2	10.00_2	(<1/2 → down)	2
2 3/16	10.00110_2	10.01_2	(>1/2 → up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2 → up)	3
2 5/8	10.10100_2	10.10_2	(1/2 → down)	2 1/2

Floating Point wrapup

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Properties
- Floating point in C
- Summary
- Intro to machine level programming (Ch 3)

C float Decoding Example

$v = (-1)^s M 2^E$

E = exp – bias

float: `0xC0A00000`

binary:

1 8-bits 23-bits

E =

S =

M =

$v = (-1)^s M 2^E$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

C float Decoding Example

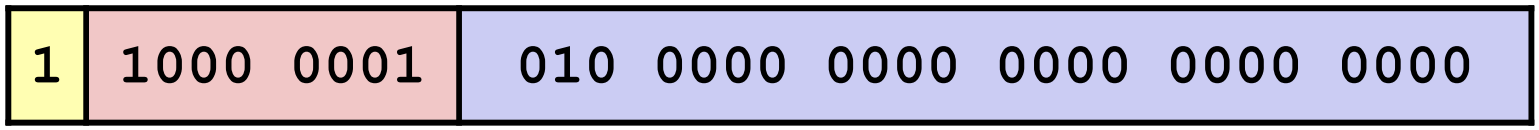
$v = (-1)^s M 2^E$

$E = exp - bias$

float: `0xC0A00000`

$Bias = 2^{k-1} - 1 = 127$

binary: 1100 0000 1010 0000 0000 0000 0000 0000



E =

S =

M =

$v = (-1)^s M 2^E$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

C float Decoding Example

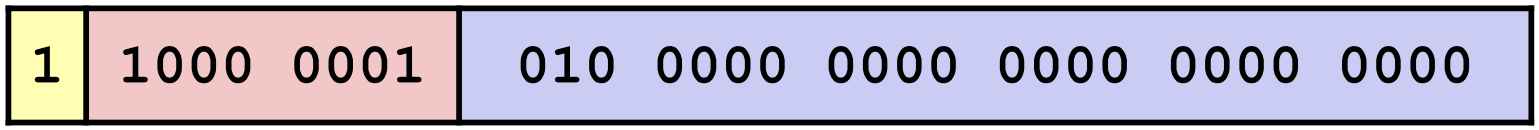
$v = (-1)^s M 2^E$

$E = exp - bias$

float: `0xC0A00000`

$Bias = 2^{k-1} - 1 = 127$

binary: 1100 0000 1010 0000 0000 0000 0000 0000



1 8-bits 23-bits

$E = exp - bias = 129 - 127 = 2$

$S = 1$ (negative number)

$M = 1.010\ 0000\ 0000\ 0000\ 0000\ 0000$

$= 1 + 1/4 = 1.25$

$v = (-1)^s M 2^E = (-1)^1 * 1.25 * 2^2 = -5$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Floating Point in C

- C Guarantees Two Levels
  - **`float`** single precision
  - **`double`** double precision
- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation 😩
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to “rounding mode”

Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NaN

- `x == (int)(float) x` ✗
- `x == (int)(double) x` ✓
- `f == (float)(double) f` ✓
- `d == (double)(float) d` ✗
- `f == -(-f);` ✓
- `2/3 == 2/3.0` ✗
- `d < 0.0 ⇒ ((d*2) < 0.0)` ✓
- `d > f ⇒ -f > -d` ✓
- `d * d >= 0.0` ✓
- `(d + f)- d == f` ✗

FP Summary

- IEEE Floating Point has clear mathematical properties 😊👍
- Represents numbers of form M x 2^E
- One can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not exactly the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers

- Next up: Chapter 3!