Discussion - Week 4

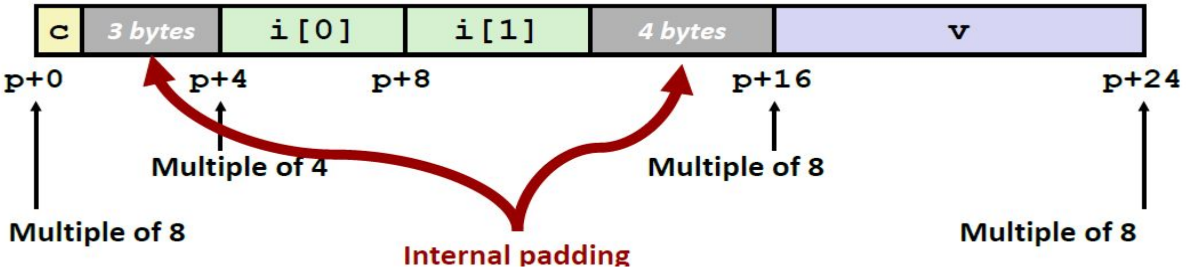
TA : Mathanky
Email : mathanky04@ucla.edu
Office Hours : 12.30PM - 2.30PM,
Boelter Hall 3256F

UNIONS

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element’s alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- **Example:**
  - K = 8, due to **`double`** element

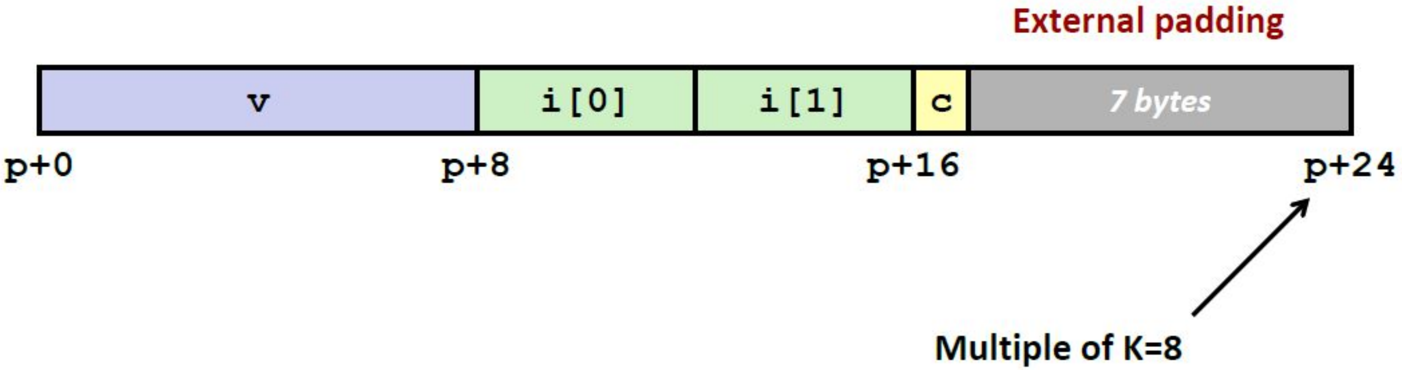
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



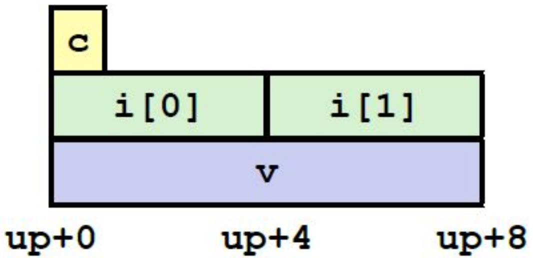
Compound Types in C

- **Arrays**
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - Way to circumvent type system

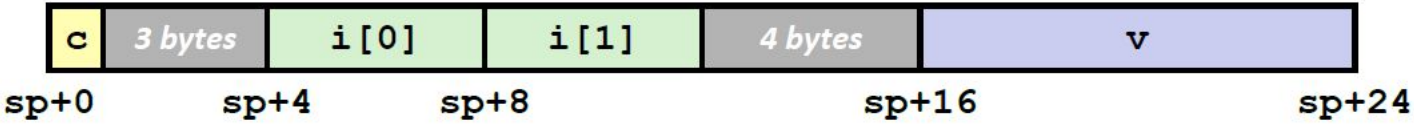
Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

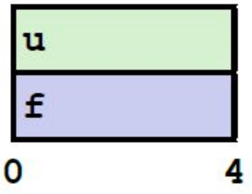



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(float) u` ?

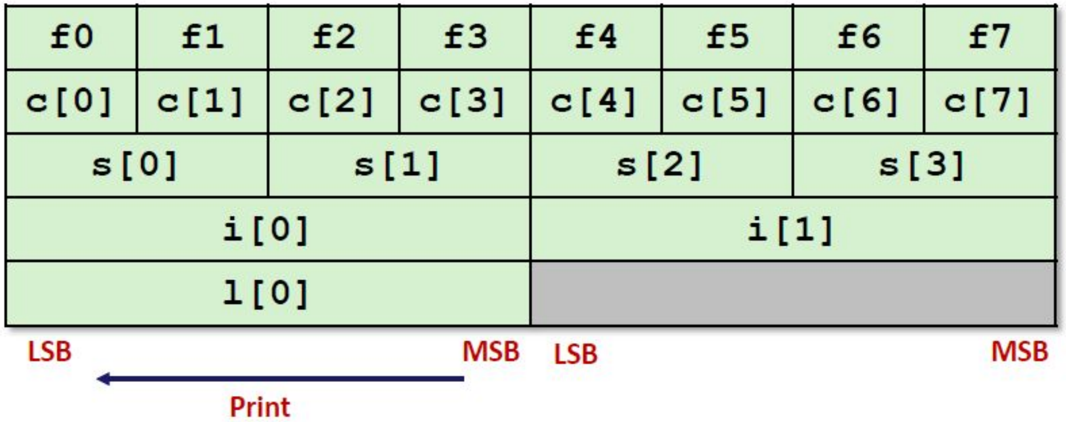
Same as `(unsigned) f` ?

Byte Ordering Revisited

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- **Big Endian**
  - Most significant byte has lowest address
  - Sparc, *Internet*
- **Little Endian**
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- **Bi Endian**
  - Can be configured either way
  - ARM

Byte Ordering on IA32

Little Endian

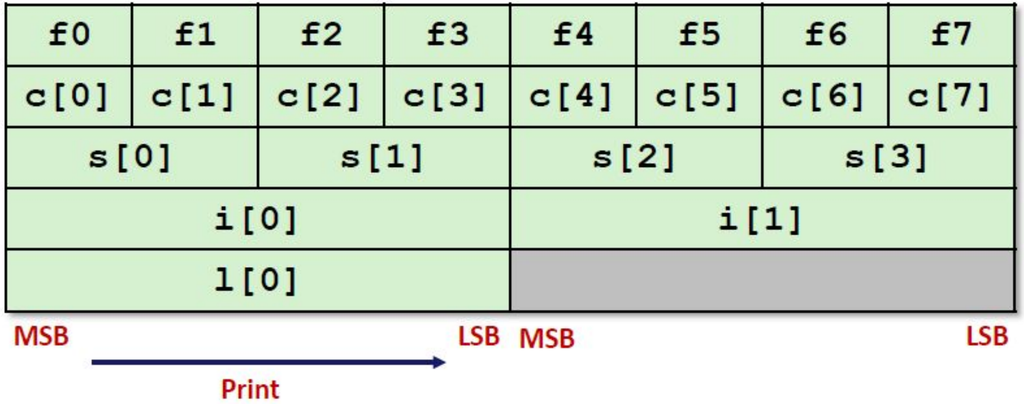


Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian



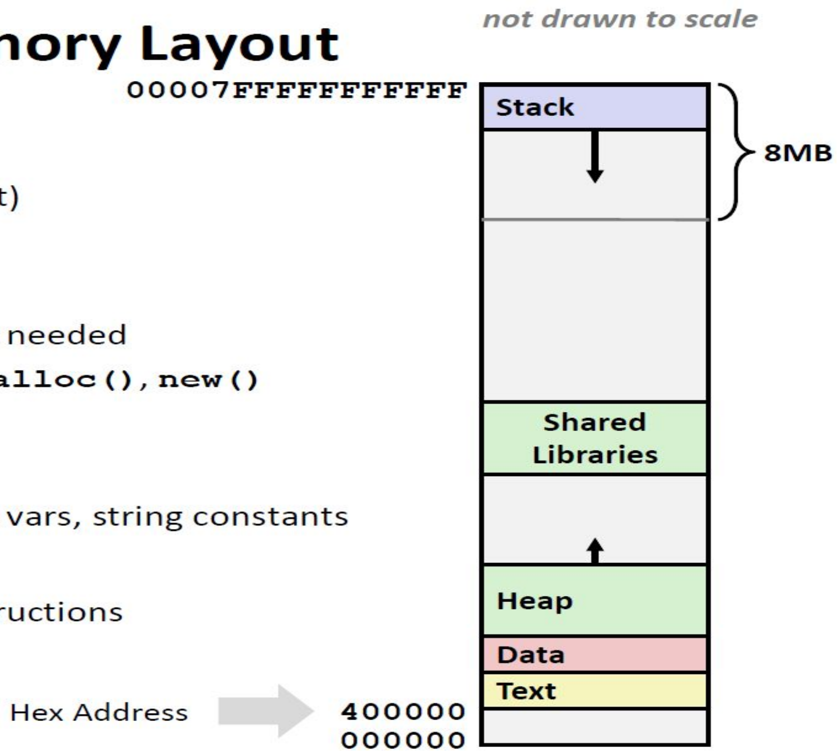
Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Machine-Level Programming : Advanced

x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only



Memory Allocation Example

not drawn to scale

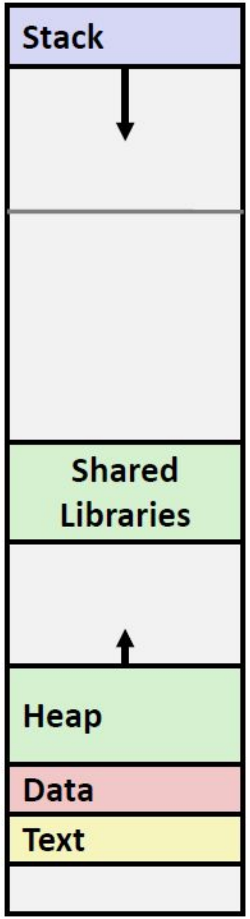
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

Where does everything go?

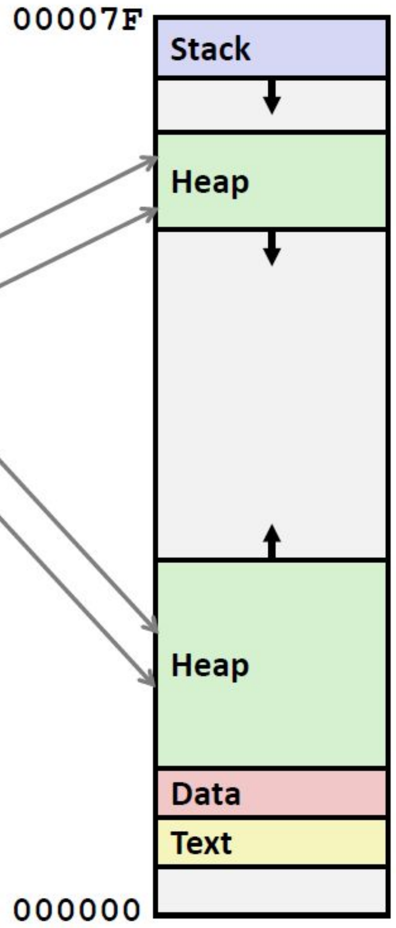


x86-64 Example Addresses

address range ~2^{47}

`local`	`0x00007ffe4d3be87c`
`p1`	`0x00007f7262a1e010`
`p3`	`0x00007f7162a1d010`
`p4`	`0x000000008359d120`
`p2`	`0x000000008359d010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x0000000000400590`

not drawn to scale



Recall: Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)   ->    3.14
fun(1)   ->    3.14
fun(2)   ->    3.1399998664856
fun(3)   ->    2.00000061035156
fun(4)   ->    3.14
fun(6)   ->    Segmentation fault
```

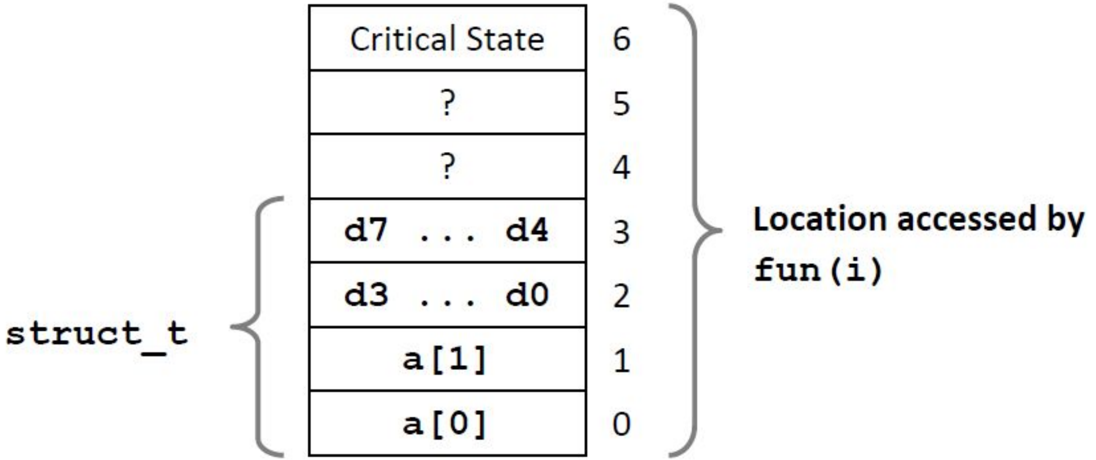
- Result is system specific

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

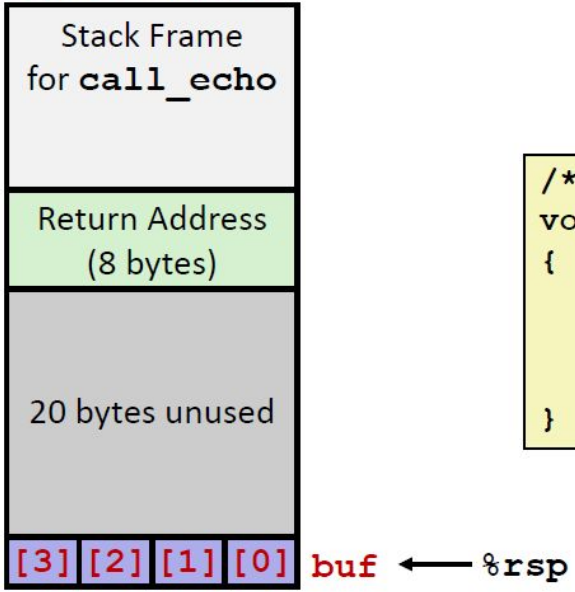
`fun(0)`	->	3.14
`fun(1)`	->	3.14
`fun(2)`	->	3.1399998664856
`fun(3)`	->	2.00000061035156
`fun(4)`	->	3.14
`fun(6)`	->	Segmentation fault

Explanation:



Buffer Overflow Stack

Before call to gets

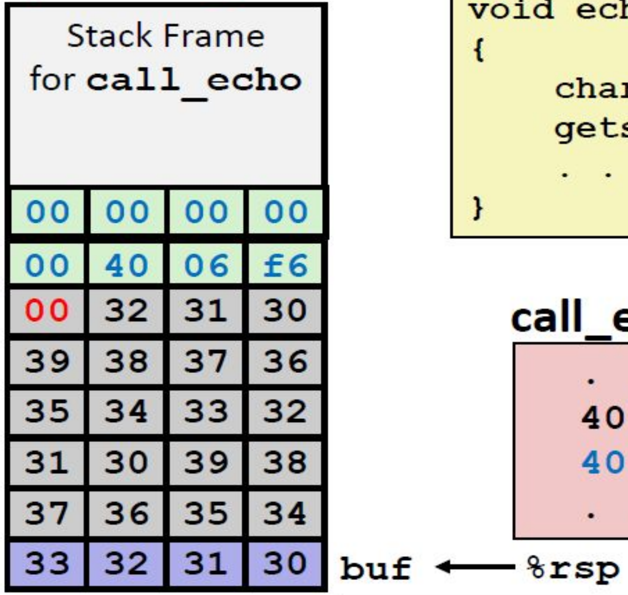



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer Overflow Stack Example #1

After call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
  . . .
  4006f1:  callq   4006cf <echo>
  4006f6:  add     $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

"01234567890123456789012\0"

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
  . . .
  4006f1:   callq   4006cf <echo>
  4006f6:   add     $0x8,%rsp
  . . .
```

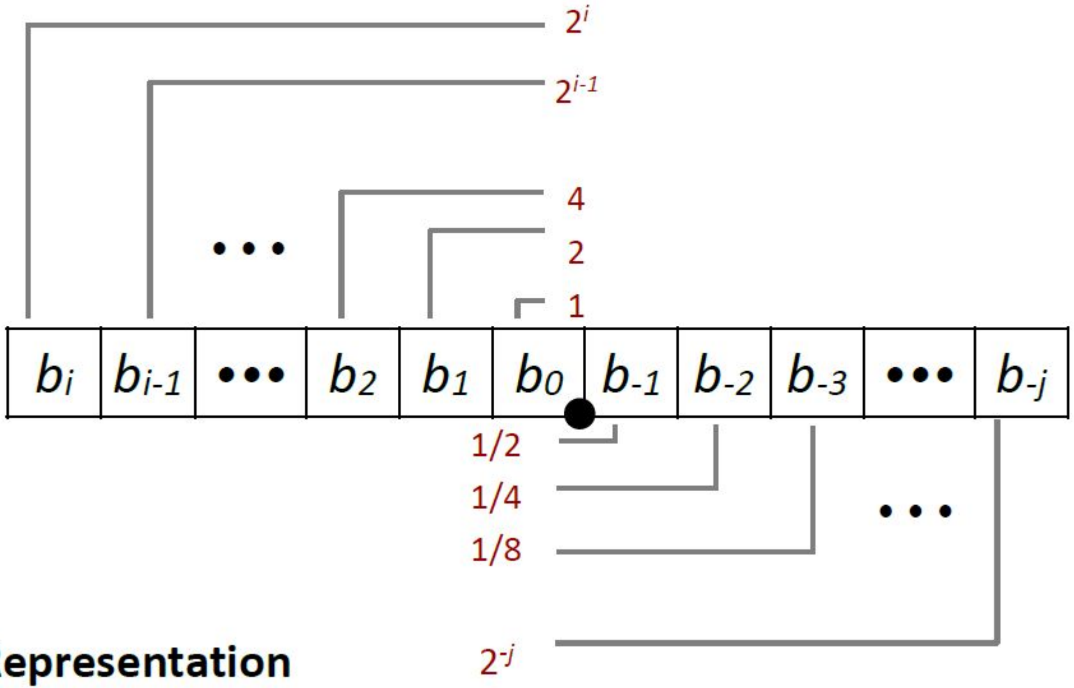
```
unix>./bufdemo-nsp
Type a string:01234567890123456789012334
Segmentation Fault
```

"01234567890123456789012334\0"

Overflowed buffer and corrupted return pointer

FLOATING POINT NUMBERS

Fractional Binary Numbers



- **Representation**
  - 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	
5 3/4 = 23/4	101.11_2	= 4 + 1 + 1/2 + 1/4
2 7/8 = 23/8	10.111_2	= 2 + 1/2 + 1/4 + 1/8
1 7/16 = 23/16	1.0111_2	= 1 + 1/4 + 1/8 + 1/16

$23 = 16 + 4 + 2 + 1 = 10111_2$

- **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
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- **Limitation #1**
  - Can only exactly represent 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 `0.001100110011[0011]`$\ldots_2$
    - 1/10 `0.0001100110011[0011]`$\ldots_2$

- **Limitation #2**
  - Just one setting of binary point within the *w* bits
    - Limited range of numbers (very small values? very large?)

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 *M*** normally a fractional value in range [1.0,2.0).
  - **Exponent *E*** weights value by power of two

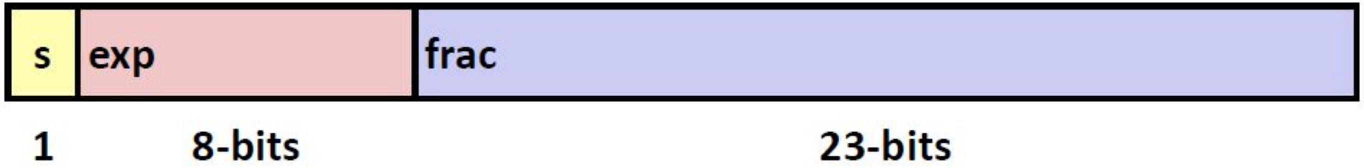
- **Encoding**
  - MSB **s** is sign bit ***s***
  - **exp** field encodes ***E*** (but is not equal to E)
  - **frac** field encodes ***M*** (but is not equal to M)

s	exp	frac

Precision options

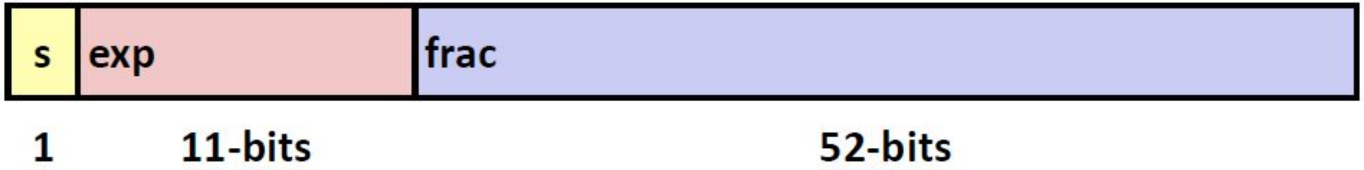
- **Single precision: 32 bits**

  $\approx$ 7 decimal digits, $10^{\pm 38}$



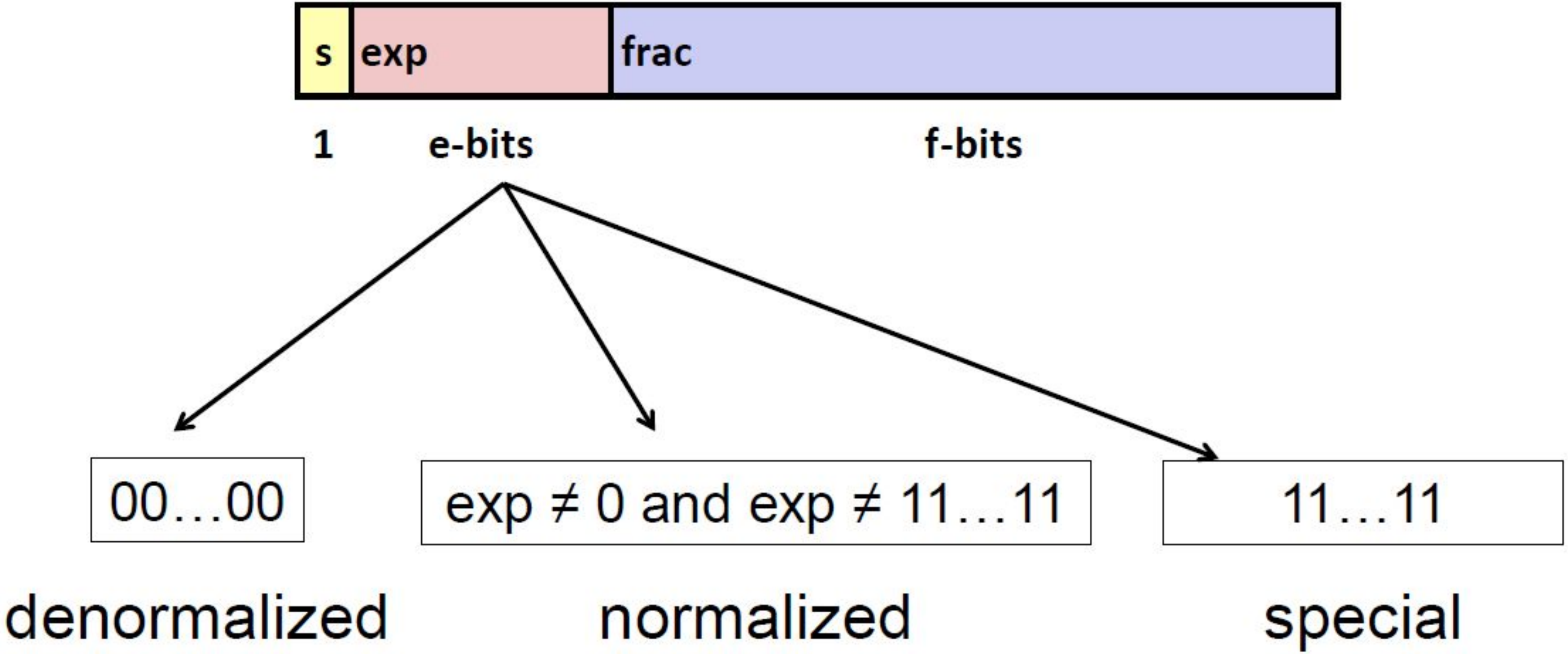
- **Double precision: 64 bits**

  $\approx$ 16 decimal digits, $10^{\pm 308}$



- **Other formats: half precision, quad precision**

Three "kinds" of floating point numbers



"Normalized" Values

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

- **When: exp ≠ 000…0 and exp ≠ 111…1**

- **Exponent coded 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 coded with implied leading 1: *M* =** $1.\text{xxx…x}_2$
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

Normalized Encoding Example

$$v = (-1)^s M\, 2^E$$
$$E = Exp - Bias$$

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

  M = $1.\underline{1101101101101}_2$

  `frac`= $\underline{1101101101101}0000000000_2$

- **Exponent**

  E = 13

  $Bias$ = 127

  Exp = 140 = 10001100_2

- **Result:**

0	10001100	11011011011010000000000
`s`	`exp`	`frac`

Denormalized Values

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

- **Condition:** exp = 000…0

- **Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - Bias$)**
  - Same exponent as smallest normalized numbers, but leading 0: consistent
- **Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$**
  - `xxx…x`: bits of `frac`
- **Cases**
  - `exp` = `000…0`, `frac` = `000…0`
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - Numbers closest to 0.0
    - Equispaced

Special Values

- **Condition: `exp` = `111`…`1`**

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

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

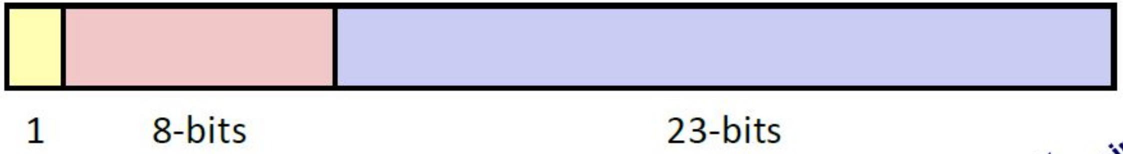
C float Decoding Example

$v = (-1)^s M\, 2^E$
$E = \texttt{exp} - Bias$

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

float: `0xC0A00000`

binary: ____ ____ ____ ____ ____ ____ ____ ____



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

$v = (-1)^s M 2^E$
$E = \texttt{exp} - Bias$

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

float: `0xC0A00000`

binary: `1100 0000 1010 0000 0000 0000 0000 0000`

1	1000 0001	010 0000 0000 0000 0000 0000
1	8-bits	23-bits

E = exp – Bias = 129 – 127 = 2 (decimal)

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 Operations: Basic Idea

- $\mathtt{x} +_{\mathtt{f}} \mathtt{y} = \mathtt{Round}(\mathtt{x} + \mathtt{y})$

- $\mathtt{x} \times_{\mathtt{f}} \mathtt{y} = \mathtt{Round}(\mathtt{x} \times \mathtt{y})$

- **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 (illustrate with $ rounding)**

	$1.40	**$1.60**	**$1.50**	**$2.50**	**–$1.50**
Towards zero	$1 ↓	$1 ↓	$1 ↓	$2 ↓	–$1 ↑
Round down ($-\infty$)	$1 ↓	$1 ↓	$1 ↓	$2 ↓	–$2 ↓
Round up ($+\infty$)	$2 ↑	$2 ↑	$2 ↑	$3 ↑	–$1 ↑
Nearest Even (default)	$1 ↓	$2 ↑	$2 ↑	$2 ↓	–$2 ↓

Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
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

FP Multiplication

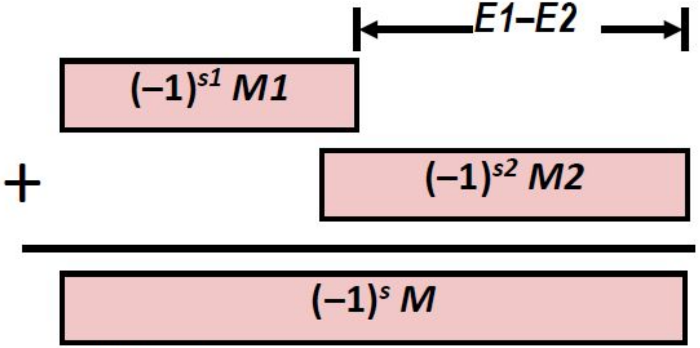
- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \text{x}\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*: $s1$ ^ $s2$
  - Significand *M*: $M1 \times M2$
  - Exponent *E*: $E1 + E2$
- **Fixing**
  - If $M \geq 2$, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit `frac` precision
- **Implementation**
  - Biggest chore is multiplying significands

4 bit mantissa: $1.010*2^{2}$ x $1.110*2^{3}$ = $10.0011*2^{5}$
= $1.00011*2^{6}$ = $1.001*2^{6}$

Floating Point Addition

- $(-1)^{s1}\, M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\, M2\ \ 2^{E2}$
  - Assume $E1 > E2$
- **Exact Result:** $(-1)^{s}\, M\ \ 2^{E}$
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: $E1$
- **Fixing**
  - If $M \geq 2$, shift M right, increment E
  - if $M < 1$, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit `frac` precision

Get binary points lined up



$$1.010 * 2^{2} + 1.110 * 2^{3} = (0.1010 + 1.1100) * 2^{3}$$
$$= 10.0110 * 2^{3} = 1.00110 * 2^{4} = 1.010 * 2^{4}$$

Mathematical Properties of FP Add

- **Compare to those of Abelian Group**
  - Closed under addition? *Yes*
    - But may generate infinity or NaN
  - Commutative? *Yes*
  - Associative? *No*
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
  - 0 is additive identity? *Yes*
  - Every element has additive inverse? *Almost*
    - Yes, except for infinities & NaNs
- **Monotonicity**
  - $a \geq b \Rightarrow a+c \geq b+c$? *Almost*
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- **Compare to Commutative Ring**
  - Closed under multiplication? ***Yes***
    - But may generate infinity or NaN
  - Multiplication Commutative? ***Yes***
  - Multiplication is Associative? ***No***
    - Possibility of overflow, inexactness of rounding
    - Ex: `(1e20*1e20)*1e-20= inf, 1e20*(1e20*1e-20)= 1e20`
  - 1 is multiplicative identity? ***Yes***
  - Multiplication distributes over addition? ***No***
    - Possibility of overflow, inexactness of rounding
    - `1e20*(1e20-1e20)= 0.0, 1e20*1e20 – 1e20*1e20 = NaN`
- **Monotonicity**
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$? ***Almost***
    - Except for infinities & NaNs

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
Gcc/x86-64 on shark

- `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`

PRACTICE PROBLEMS

What is the value of y after both of the following operations?

```
x = x ^ (~y);
y = y ^ x;
```

What is the value of y after both of the following operations?

```
x = x ^ (~y);
y = y ^ x;
```

~x

Say x = 0111 and y is 1010

0111 ^ 0101 = 0010

1010^0010 = 1000 which is ~x

Given the following declarations, do the statements below always evaluate to true?

```
int x = rand();
Int y = rand();
unsigned ux = rand();
```

a.

```
x > ux  ====> (~x+1)  < 0
```

b.

```
ux - 2 >= -2 ====> ux <= 1
```

c.

```
(x^y)^x == (x+y)^((x+y)^y)
```

d.

```
(x < 0) && (y < 0) == (x + y) < 0
```

Given the following declarations, do the statements below always evaluate to true?

```
int x = rand();
Int y = rand();
unsigned ux = rand();
```

a.

x > ux ====> (~x+1) < 0 **FALSE**

b.

ux - 2 >= -2 ====> ux <= 1 **TRUE**

c.

(x^y)^x == (x+y)^((x+y)^y) **TRUE**

d.

(x < 0) && (y < 0) == (x + y) < 0 **FALSE**

```
char** apple[5][9];
char* banana[1][9];
char strawberry[4][2];
```

How many bytes of space would these declarations require?

```
char** apple[5][9];
char* banana[1][9];
char strawberry[4][2];
```

How many bytes of space would these declarations require?

360 bytes (8 * 5 * 9) +
72 bytes (8 * 1 * 9) +
8 bytes (1 * 4 * 2)

Consider the following struct:

```
typedef struct {
    char first;
    int second;
    short third;
} stuff;
```

Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called array - defined as:

```
stuff array[2][2];
```

```
(gdb) x/48xb 0x7fffffffe020
0x7fffffffe020: 0x61    0x00    0x00    0x00    0x08    0x00    0x00    0x00
0x7fffffffe028: 0x02    0x00    0x00    0x00    0x62    0x00    0x00    0x00
0x7fffffffe030: 0x64    0x00    0x00    0x00    0x04    0x00    0x00    0x00
0x7fffffffe038: 0x63    0x04    0x40    0x00    0xed    0x03    0x00    0x00
0x7fffffffe040: 0xc8    0x00    0xff    0xff    0x64    0x7f    0x00    0x00
0x7fffffffe048: 0x17    0xa6    0x00    0x00    0xe1    0x00    0x00    0x00
```

1005

Because of alignment, each object of type "`stuff`" is 12 bytes.

Due to how arrays are stored in memory,

- **The array is stored as:**
  array[0][0], array[0][1], array[1][0], array[1][1]

From the gdb output, we can tell that the array starts at `0x7fffffffe020`

- **array[1][0] is `0x7fffffffe038` to `0x7fffffffe043`**
  - **Note: this is in hex, so `0x7fffffffe038` + 8 = `0x7fffffffe040`**

`Second` is an integer, and is the 5th to 8th byte of an object of type "`stuff`"

- **These are bytes `0x7fffffffe03c` to `0x7fffffffe03f`**
- **They have the values `0xed`, `0x03`, `0x00`, `0x00`**
- **Since this system is little endian, the value is `0x000003ed`**
  - **This is equivalent to 1005**