Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
      c.getMPG();
```

Assembly language:

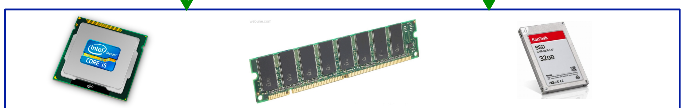
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Computer system:

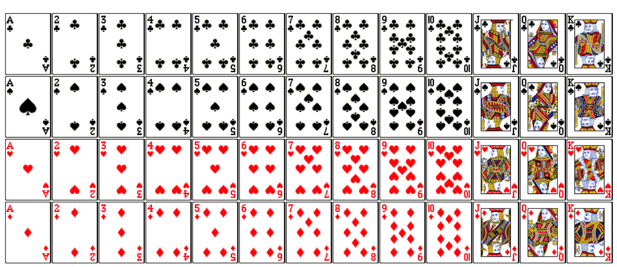


Integers

- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension

But before we get to integers….

- Encode a standard deck of playing cards.
- 52 cards in 4 suits
  - How do we encode suits, face cards?
- What operations do we want to make easy to implement?
  - Which is the higher value card?
  - Are they the same suit?



Two possible representations

- 52 cards – 52 bits with bit corresponding to card set to 1

low-order 52 bits of 64-bit word

  - "One-hot" encoding
  - Drawbacks:
    - Hard to compare values and suits
    - Large number of bits required

Two possible representations

- **52 cards – 52 bits with bit corresponding to card set to 1**

  low-order 52 bits of 64-bit word

  - "One-hot" encoding
  - Drawbacks:
    - Hard to compare values and suits
    - Large number of bits required
- **4 bits for suit, 13 bits for card value – 17 bits with two set to 1**
  - Pair of one-hot encoded values
  - Easier to compare suits and values
    - Still an excessive number of bits
- **Can we do better?**

Two better representations

- **Binary encoding of all 52 cards – only 6 bits needed**

  low-order 6 bits of a byte

  - Fits in one byte
  - Smaller than one-hot encodings.
  - How can we make value and suit comparisons easier?

Two better representations

- **Binary encoding of all 52 cards – only 6 bits needed**

  low-order 6 bits of a byte

  - Fits in one byte
  - Smaller than one-hot encodings.
  - How can we make value and suit comparisons easier?
- **Binary encoding of suit (2 bits) and value (4 bits) separately**

  suit value

  - Also fits in one byte, and easy to do comparisons

Compare Card Suits

mask: a bit vector that, when bitwise ANDed with another bit vector *v*, turns all *but* the bits of interest in *v* to 0

```
#define SUIT_MASK  0x30

int sameSuitP(char card1, char card2) {
  return (! (card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

returns int

SUIT_MASK = 0x30 = | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

suit value

equivalent

```
char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( sameSuitP(card1, card2) ) { ... }
```

Compare Card Values

mask: a bit vector that, when bitwise ANDed with another bit vector *v*, turns all *but* the bits of interest in *v* to 0

```
#define VALUE_MASK  0x0F

int greaterValue(char card1, char card2) {
  return ((unsigned int)(card1 & VALUE_MASK) >
          (unsigned int)(card2 & VALUE_MASK));
}
```

works even if value is stored in high bits

VALUE_MASK = 0x0F = | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

suit value

```
char hand[5];        // represents a 5-card hand
char card1, card2;   // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( greaterValue(card1, card2) ) { ... }
```


Encoding Integers

- **The hardware (and C) supports two flavors of integers:**
  - *unsigned* – only the non-negatives
  - *signed* – both negatives and non-negatives
- **There are only 2^W distinct bit patterns of *W* bits, so...**
  - Can not represent all the integers
  - **Unsigned values: 0 ... 2^W-1**
  - **Signed values: -2^{W-1} ... 2^{W-1}-1**
- **Reminder: terminology for binary representations**

"Most-significant" or "high-order" bit(s)

"Least-significant" or "low-order" bit(s)

0110010110101001

Unsigned Integers

- **Unsigned values are just what you expect**
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + b_5 2^5 + \ldots + b_1 2^1 + b_0 2^0$
    - Useful formula: $1+2+4+8+\ldots+2^{N-1} = 2^N - 1$
- **Add and subtract using the normal "carry" and "borrow" rules, just in binary.**

```
  00111111      63
+ 00001000    +  8
  01000111      71
```

- **How would you make *signed* integers?**

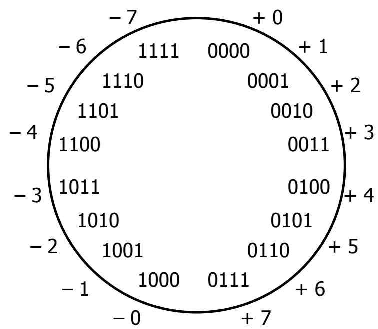
Signed Integers: Sign-and-Magnitude

- **Let's do the natural thing for the positives**
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, …, 0x7F = 127
- **But, we need to let about half of them be negative**
  - Use the **high-order bit** to indicate *negative*: call it the **"sign bit"**
    - Call this a "sign-and-magnitude" representation
  - Examples (8 bits):
    - 0x00 = 00000000_2 is non-negative, because the sign bit is 0
    - 0x7F = 01111111_2 is non-negative
    - 0x85 = 10000101_2 is negative
    - 0x80 = 10000000_2 is negative...

Signed Integers: Sign-and-Magnitude

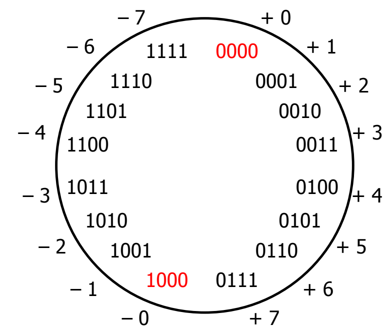
- **How should we represent -1 in binary?**
  - **10000001_2**
    Use the MSB for + or -, and the other bits to give magnitude.

Most Significant Bit



Sign-and-Magnitude Negatives

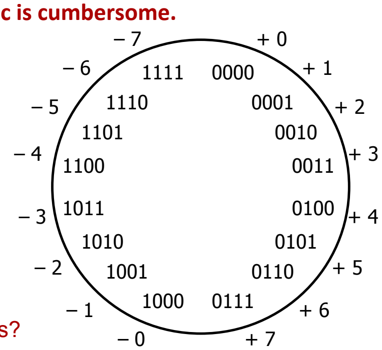
- **How should we represent -1 in binary?**
  - **10000001_2**
    Use the MSB for + or -, and the other bits to give magnitude.
    (Unfortunate side effect: there are **two representations of 0**!)



Sign-and-Magnitude Negatives

- **How should we represent -1 in binary?**
  - **10000001_2**
    Use the MSB for + or -, and the other bits to give magnitude.
    (Unfortunate side effect: there are two representations of 0!)
  - Another problem: **arithmetic is cumbersome.**
    - Example:
      4 - 3 != 4 + (-3)

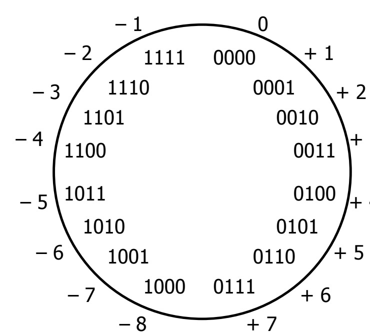
```
 0100
+1011
 1111
```



How do we solve these problems?

Two's Complement Negatives

- **How should we represent -1 in binary?**

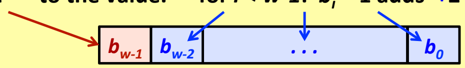


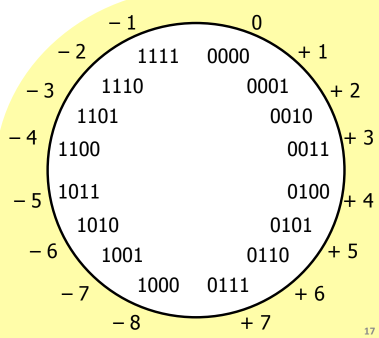
Two's Complement Negatives

- **How should we represent -1 in binary?**

  Rather than a sign bit, let MSB have same value, but ***negative weight.***

  b_{w-1} = 1 adds -2^{w-1} to the value. **for $i < w-1$: b_i = 1 adds +2^i to the value.**





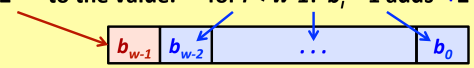
University of Washington

Two's Complement Negatives

- **How should we represent -1 in binary?**

  Rather than a sign bit, let MSB have same value, but ***negative weight.***

  b_{w-1} = 1 adds -2^{w-1} to the value. **for $i < w-1$: b_i = 1 adds +2^i to the value.**

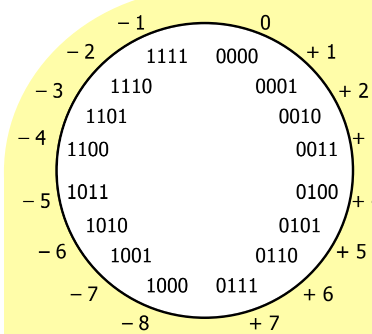


e.g. **unsigned** 1010_2:

$1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10}$

2's compl. 1010_2:

$-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10}$



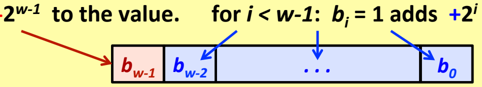
University of Washington

Two's Complement Negatives

- **How should we represent -1 in binary?**

  Rather than a sign bit, let MSB have same value, but ***negative weight.***

  b_{w-1} = 1 adds -2^{w-1} to the value. **for $i < w-1$: b_i = 1 adds +2^i to the value.**



  e.g. **unsigned** 1010_2:

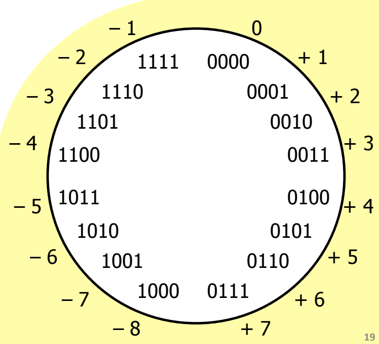
  $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10}$

  2's compl. 1010_2:

  $-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10}$

- -1 is represented as $\mathbf{1111_2} = -2^3 + (2^3 - 1)$
  All negative integers still have MSB = 1.
- **Advantages:** single zero, simple arithmetic
- **To get negative representation of any integer, take bitwise complement and then add one!**

`~x + 1 == -x`



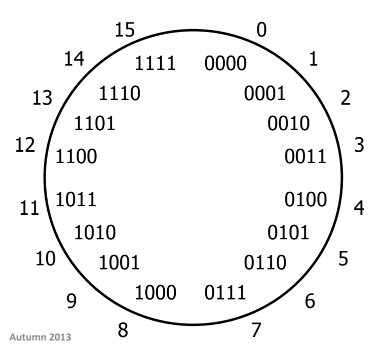
University of Washington

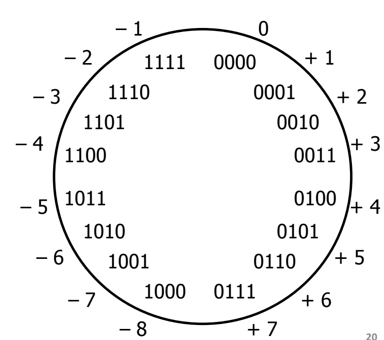
4-bit Unsigned vs. Two's Complement

1 0 1 1

$2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$

$-2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$





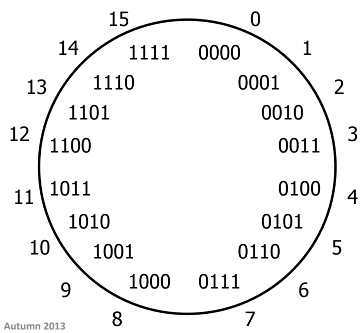
4-bit Unsigned vs. Two's Complement

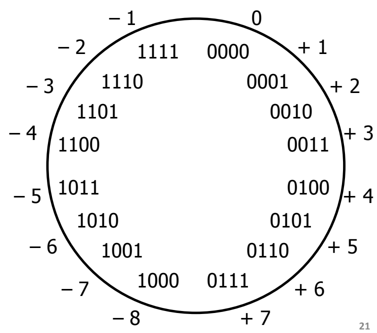
1 0 1 1

$2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$

$-2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$

11 ← (math) difference = 16 = 2^4 → -5





University of Washington

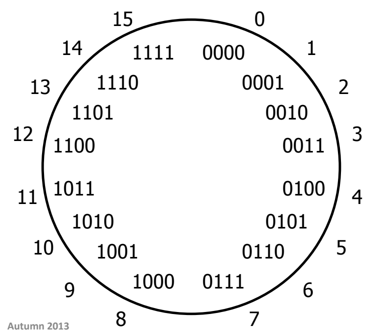
4-bit Unsigned vs. Two's Complement

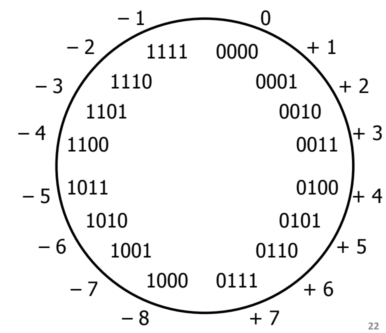
1 0 1 1

$2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$

$-2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$

11 (math) difference = 16 = 2^4 -5





University of Washington

Two's Complement Arithmetic

- **The same addition procedure works for both unsigned and two's complement integers**
  - Simplifies hardware: only one algorithm for addition
  - Algorithm: simple addition, discard the highest carry bit
    - Called "modular" addition: result is sum *modulo* 2^W
- **Examples:**

4 + 3	0100 + 0011	4 – 3	0100 + 1101	– 4 + 3	1100 + 0011
= 7	= 0111	= 1 drop carry	1 0001 = 0001	– 1	1111

University of Washington

Two's Complement

- **Why does it work?**
  - Put another way, for all positive integers x, we want:
    - *bits(x) + bits(–x) = 0* (ignoring the carry-out bit)
  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
 00000001
+????????   (we want whichever bit string gives the right result)
 00000000

 00000010       00000011
+????????      +????????
 00000000       00000000
```

Two's Complement

- **Why does it work?**
  - Put another way, for all positive integers x, we want:
    - *bits(x) + bits(–x) = 0* (ignoring the carry-out bit)

  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
 00000001
+11111111   (we want whichever bit string gives the right result)
100000000

 00000010     00000011
+????????    +????????
 00000000     00000000
```


Two's Complement

- **Why does it work?**
  - Put another way, for all positive integers x, we want:
    - *bits(x) + bits(–x) = 0* (ignoring the carry-out bit)

  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
 00000001
+11111111   (we want whichever bit string gives the right result)
100000000

 00000010     00000011
+11111110    +11111101
100000000    100000000
```


Unsigned & Signed Numeric Values

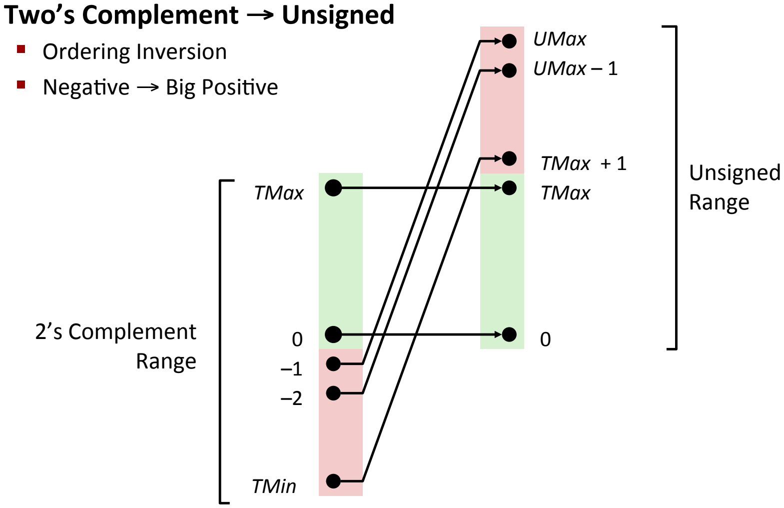
bits	Unsigned	Signed
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Signed and unsigned integers have limits.**
  - If you compute a number that is too big (positive), it wraps:
    6 + 4 = ? 15U + 2U = ?
  - If you compute a number that is too small (negative), it wraps:
    -7 - 3 = ? 0U - 2U = ?
  - Answers are only correct mod 2^b

- **The CPU may be capable of "throwing an exception" for overflow on signed values.**
  - It won't for unsigned.
- **But C and Java just cruise along silently when overflow occurs... Oops.**

Conversion Visualized

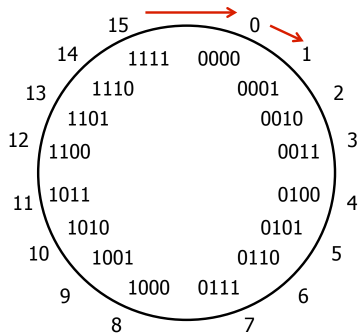
- **Two's Complement → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



Overflow/Wrapping: Unsigned

addition: drop the carry bit

15	1111
+ 2	+ 0010
~~17~~	10001
1	



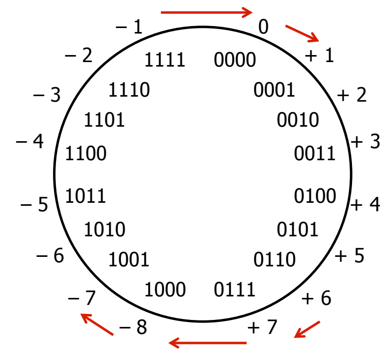
Modular Arithmetic

Overflow/Wrapping: Two's Complement

addition: drop the carry bit

-1	1111
+ 2	+ 0010
1	10001

6	0110
+ 3	+ 0011
~~9~~	1001
-7	



Modular Arithmetic

Values To Remember

- **Unsigned Values**
  - **UMin = 0**
    - 000…0
  - **UMax = $2^w - 1$**
    - 111…1
- **Two's Complement Values**
  - **TMin = -2^{w-1}**
    - 100…0
  - **TMax = $2^{w-1} - 1$**
    - 011…1
  - **Negative one**
    - 111…1 0xF…F

Values for *W* = 32

	Decimal	Hex	Binary
UMax	**4,294,967,296**	FF FF FF FF	11111111 11111111 11111111 11111111
TMax	**2,147,483,647**	7F FF FF FF	01111111 11111111 11111111 11111111
TMin	**-2,147,483,648**	80 00 00 00	10000000 00000000 00000000 00000000
-1	**-1**	FF FF FF FF	11111111 11111111 11111111 11111111
0	**0**	00 00 00 00	00000000 00000000 00000000 00000000

Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Use "U" suffix to force unsigned:
    - `0U`, `4294967259U`

Signed vs. Unsigned in C

!!!

- **Casting**
  - `int tx, ty;`
  - `unsigned ux, uy;`
  - **Explicit casting between signed & unsigned:**
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
  - **Implicit casting also occurs via assignments and function calls:**
    - `tx = ux;`
    - `uy = ty;`
    - The gcc flag -*Wsign-conversion* produces warnings for implicit casts, but -*Wall* does not!
  - **How does casting between signed and unsigned work?**
  - **What values are going to be produced?**

Signed vs. Unsigned in C

!!!

- **Casting**
  - `int tx, ty;`
  - `unsigned ux, uy;`
  - **Explicit casting between signed & unsigned:**
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
  - **Implicit casting also occurs via assignments and function calls:**
    - `tx = ux;`
    - `uy = ty;`
    - The gcc flag -*Wsign-conversion* produces warnings for implicit casts, but -*Wall* does not!
  - **How does casting between signed and unsigned work?**
  - **What values are going to be produced?**
    - ***Bits are unchanged*, just interpreted differently!**

Casting Surprises

!!!

- **Expression Evaluation**
  - If you mix unsigned and signed in a single expression, then ***signed values are implicitly cast to unsigned***.
  - Including comparison operations **<, >, ==, <=, >=**
  - Examples for *W* = 32: **TMIN = -2,147,483,648** **TMAX = 2,147,483,647**

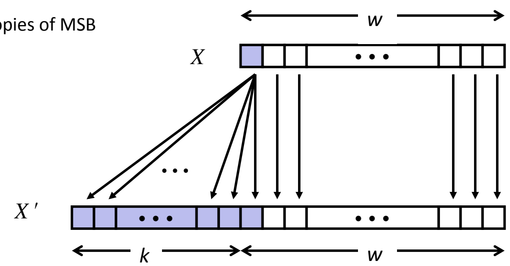
Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483648	>	signed
2147483647U	-2147483648	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Sign Extension

- **What happens if you convert a 32-bit signed integer to a 64-bit signed integer?**

Sign Extension

- **Task:**
  - Given w-bit signed integer x
  - Convert it to w+k-bit integer *with same value*
- **Rule:**
  - Make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



8-bit representations

0 0 0 0 1 0 0 1

1 0 0 0 0 0 0 1

1 1 1 1 1 1 1 1

0 0 1 0 0 1 1 1

C: casting between unsigned and signed just reinterprets the same bits.

Sign Extension

0 0 1 0 4-bit 2

0 0 0 0 0 0 1 0 8-bit 2

1 1 0 0 4-bit -4

? ? ? ? 1 1 0 0 8-bit -4

Sign Extension

0 0 1 0 4-bit 2

0 0 0 0 0 0 1 0 8-bit 2

1 1 0 0 4-bit -4

0 0 0 0 1 1 0 0 8-bit 12

Sign Extension

0 0 1 0	4-bit 2
0 0 0 0 0 0 1 0	8-bit 2
1 1 0 0	4-bit -4
1 0 0 0 1 1 0 0	8-bit -116

Sign Extension

0 0 1 0	4-bit 2
0 0 0 0 0 0 1 0	8-bit 2
1 1 0 0	4-bit -4
1 1 1 1 1 1 0 0	8-bit -4

Sign Extension Example

- **Converting from smaller to larger integer data type**
- **C automatically performs sign extension (Java too)**

```
short int x =  12345;
int       ix = (int) x;
short int y = -12345;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	12345	30 39	00110000 01101101
ix	12345	00 00 30 39	00000000 00000000 00110000 01101101
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

Shift Operations

- **Left shift: x << y**
  - Shift bit vector x left by y positions
    - Throw away extra bits on left
    - Fill with 0s on right
- **Right shift: x >> y**
  - Shift bit-vector x right by y positions
    - Throw away extra bits on right
  - Logical shift (for unsigned values)
    - Fill with 0s on left
  - Arithmetic shift (for signed values)
    - Replicate most significant bit on left
    - Maintains sign of x

Argument x	01100010
<< 3	00010*000*
Logical >> 2	*00*011000
Arithmetic >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Logical >> 2	*00*101000
Arithmetic >> 2	*11*101000

The behavior of >> in C depends on the compiler! It is *arithmetic* shift right in GCC.
Java: >>> is logical shift right; >> is arithmetic shift right.

Shift Operations

- **Left shift:** **x << y**
  - Shift bit vector x left by y positions
    - Throw away extra bits on left
    - Fill with 0s on right
- **Right shift:** **x >> y**
  - Shift bit-vector x right by y positions
    - Throw away extra bits on right
  - Logical shift (for unsigned values)
    - Fill with 0s on left
  - Arithmetic shift (for signed values)
    - Replicate most significant bit on left
    - Maintains sign of x
    - ***Why is this useful?***

Argument x	01100010
<< 3	
Logical >> 2	
Arithmetic >> 2	

Argument x	10100010
<< 3	
Logical >> 2	
Arithmetic >> 2	

x >> 9?

The behavior of >> in C depends on the compiler! It is *arithmetic* shift right in GCC.
Java: >>> is logical shift right; >> is arithmetic shift right.

What happens when…

- **x >> n?**

- **x << m?**

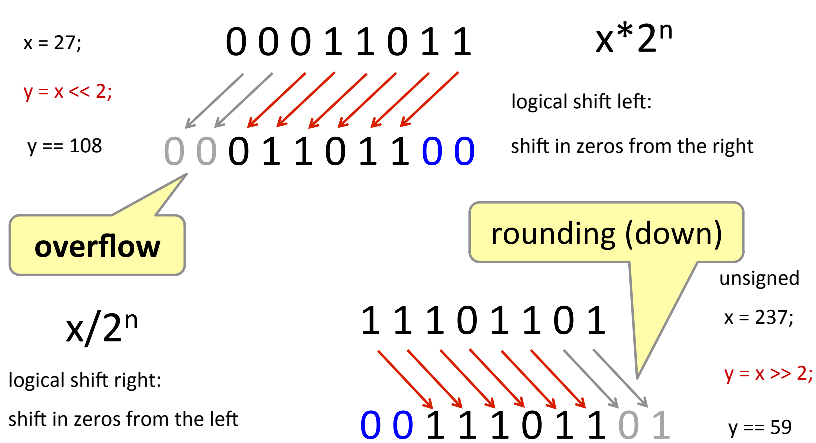
What happens when…

- **x >> n: divide by 2^n**

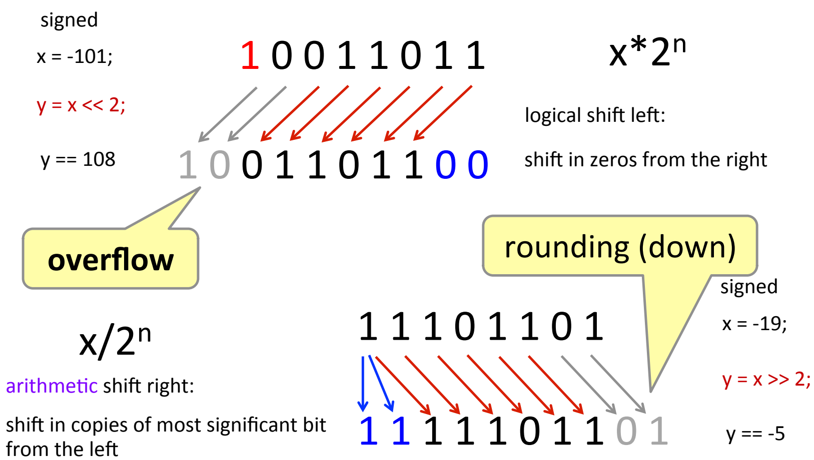
- **x << m: multiply by 2^m**

faster than general multiple or divide operations

Shifting and Arithmetic



Shifting and Arithmetic



clarification from Mon.: shifts by n < 0 or n >= word size are undefined

Using Shifts and Masks

- **Extract the 2nd most significant byte of an integer?**

x	01100001 01100010 01100011 01100100

Using Shifts and Masks

- **Extract the 2nd most significant byte of an integer:**
  - First shift, then mask: (x >> 16) & 0xFF

x	01100001 01100010 01100011 01100100
x >> 16	00000000 00000000 01100001 01100010
(x >> 16) & 0xFF	*00000000 00000000 00000000 11111111* 00000000 00000000 00000000 01100010

- **Extract the sign bit of a signed integer?**

Using Shifts and Masks

- **Extract the 2nd most significant byte of an integer:**
  - First shift, then mask: (x >> 16) & 0xFF

x	01100001 01100010 01100011 01100100
x >> 16	00000000 00000000 01100001 01100010
(x >> 16) & 0xFF	*00000000 00000000 00000000 11111111* 00000000 00000000 00000000 01100010

- **Extract the sign bit of a signed integer:**
  - (x >> 31) & 1 - need the “& 1” to clear out all other bits except LSB
- **Conditionals as Boolean expressions (assuming x is 0 or 1)**
  - if (x) a=y else a=z; which is the same as a = x ? y : z;
  - Can be re-written (assuming arithmetic right shift) as:
    a = (((x << 31) >> 31) & y) | (((!x) << 31) >> 31) & z);

Multiplication

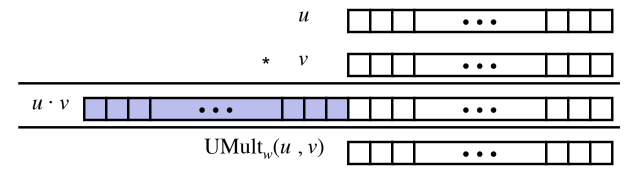
- **What do you get when you multiply 9 x 9?**
- **What about 2^{30} x 3?**
- **2^{30} x 5?**
- **-2^{31} x -2^{31}?**

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- **Standard Multiplication Function**
  - Ignores high order *w* bits
- **Implements Modular Arithmetic**

  $\text{UMult}_w(u, v) = u \cdot v \mod 2^w$

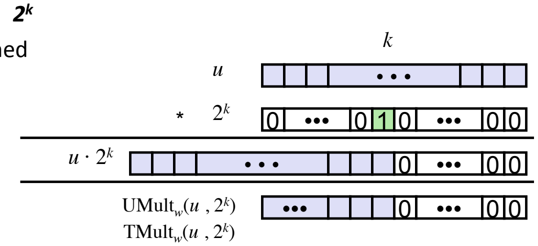
Power-of-2 Multiply with Shift

- **Operation**
  - `u << k` gives `u * 2^k`
  - Both signed and unsigned

Operands: *w* bits

True Product: *w*+*k* bits

Discard *k* bits: *w* bits



- **Examples**
  - `u << 3` == `u * 8`
  - `u << 5 - u << 3` == `u * 24`
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void* user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious Usage

```
/* Declaration of library function memcpy */
void* memcpy(void* dest, void* src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void* user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

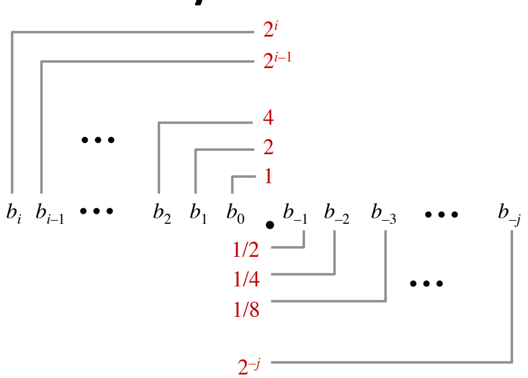
```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```


Floating point topics

- Background: fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won't cover
  - It's a 58-page standard…

Fractional Binary Numbers



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

Fractional Binary Numbers

- **Value** / **Representation**
  - 5 and 3/4 — 101.11_2
  - 2 and 7/8 — 10.111_2
  - 47/64 — 0.101111_2

- **Observations**
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form `0.111111`$…_2$ are just below 1.0
- **Limitations:**
  - Exact representation possible only for numbers of the form $x * 2^y$
  - Other rational numbers have repeating bit representations
    - $1/3 = 0.333333…_{10} = 0.01010101[01]…_2$

Fixed Point Representation

- **Implied binary point.** **Examples:**

  #1: the binary point is between bits 2 and 3

  $b_7\ b_6\ b_5\ b_4\ b_3$ [.] $b_2\ b_1\ b_0$

  #2: the binary point is between bits 4 and 5

  $b_7\ b_6\ b_5$ [.] $b_4\ b_3\ b_2\ b_1\ b_0$

- **Same hardware as for integer arithmetic.**

  #3: integers! the binary point is after bit 0

  $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ [.]

- **Fixed point = fixed *range* and fixed *precision***
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers

IEEE Floating Point

- **Analogous to scientific notation**
  - 12000000 1.2×10^{7} C: 1.2e7
  - 0.0000012 1.2×10^{-6} C: 1.2e-6

- **IEEE Standard 754 used by all major CPUs today**

- **Driven by numerical concerns**
  - Rounding, overflow, underflow
  - Numerically well-behaved, but hard to make fast in hardware

Floating Point Representation

- **Numerical form:**

$$V_{10} = (-1)^{S} * M * 2^{E}$$

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

Floating Point Representation

- **Numerical form:**

$$V_{10} = (-1)^{S} * M * 2^{E}$$

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

- **Representation in memory:**
  - 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)

Precisions

- **Single precision: 32 bits**

s	exp	frac
1 bit	8 bits	23 bits

- **Double precision: 64 bits**

s	exp	frac
1 bit	11 bits	52 bits

- **Finite representation means not all values can be represented exactly. Some will be approximated.**

Normalization and Special Values

$V = (-1)^S * M * 2^E$

s	exp	frac

- **"Normalized" = M has the form 1.xxxxx**
  - As in scientific notation, but in binary
  - 0.011×2^5 and 1.1×2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it
- **How do we represent 0.0? Or special / undefined values like 1.0/0.0?**

Normalization and Special Values

$V = (-1)^S * M * 2^E$

s	exp	frac

- **"Normalized" = M has the form 1.xxxxx**
  - As in scientific notation, but in binary
  - 0.011×2^5 and 1.1×2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it.
- **Special values:**
  - **zero**: **s** == 0 **exp** == 00…0 **frac** == 00…0
  - **+∞, -∞:** **exp** == 11…1 **frac** == 00…0

    $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -1.0/0.0 = -\infty$
  - **NaN** ("Not a Number"): **exp** == 11…1 **frac** != 00…0

    Results from operations with undefined result: sqrt(-1), $\infty - \infty$, $\infty * 0$, etc.
  - note: exp=11…1 and exp=00…0 are reserved, limiting exp range…

Floating Point Operations: Basic Idea

$V = (-1)^S * M * 2^E$

s	exp	frac

- **x +$_f$ y = *Round*(x + y)**
- **x *$_f$ y = *Round*(x * y)**
- **Basic idea for floating point operations:**
  - First, compute the exact result
  - Then, *round* the result to make it fit into desired precision:
    - Possibly overflow if exponent too large
    - Possibly drop least-significant bits of significand to fit into **frac**

Floating Point Multiplication

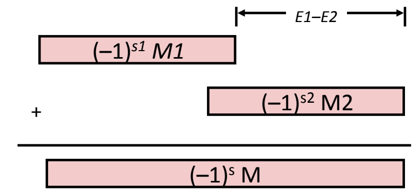
$(-1)^{s1}\ M1\ \ 2^{E1}\ \ *\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

- **Exact Result: $(-1)^s\ M\ \ 2^E$**
  - Sign *s*: *s1 ^ s2*
  - Significand *M*: *M1 * M2*
  - Exponent *E*: *E1 + E2*
- **Fixing**
  - If *M* ≥ 2, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit `frac` precision

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* ≥ 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

Rounding modes

- **Possible rounding modes (illustrate with dollar rounding):**

	\$1.40	**\$1.60**	**\$1.50**	**\$2.50**	**–\$1.50**
Round-toward-zero	\$1	\$1	\$1	\$2	–\$1
Round-down (-∞)	\$1	\$1	\$1	\$2	–\$2
Round-up (+∞)	\$2	\$2	\$2	\$3	–\$1
Round-to-nearest	\$1	\$2	??	??	??
Round-to-even	\$1	\$2	\$2	\$2	–\$2

- **Round-to-even avoids statistical bias in repeated rounding.**
  - Rounds up about half the time, down about half the time.
  - Default rounding mode for IEEE floating-point

Mathematical Properties of FP Operations

- **Exponent overflow yields +∞ or -∞**
- **Floats with value +∞, -∞, and NaN can be used in operations**
  - Result usually still +∞, -∞, or NaN; sometimes intuitive, sometimes not
- **Floating point operations are not always associative or distributive, due to rounding!**
  - (3.14 + 1e10) - 1e10 != 3.14 + (1e10 - 1e10)
  - 1e20 * (1e20 - 1e20) != (1e20 * 1e20) - (1e20 * 1e20)

Floating Point in C

!!!

- **C offers two levels of precision**
  - `float` single precision (32-bit)
  - `double` double precision (64-bit)
- **`#include <math.h>` to get `INFINITY` and `NAN` constants**
- **Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results**
  - Just avoid them!

Floating Point in C

!!!

- **Conversions between data types:**
  - Casting between `int`, `float`, and `double` **changes** the bit representation.
  - `int` → `float`
    - May be rounded; overflow not possible
  - `int` → `double` or `float` → `double`
    - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
  - `long int` → `double`
    - Rounded or exact, depending on word size
  - `double` or `float` → `int`
    - Truncates fractional part (rounded toward zero)
    - Not defined when out of range or NaN: generally sets to Tmin

Number Representation Really Matters

!!!

- **1991: Patriot missile targeting error**
  - clock skew due to conversion from integer to floating point
- **1996: Ariane 5 rocket exploded ($1 billion)**
  - overflow converting 64-bit floating point to 16-bit integer
- **2000: Y2K problem**
  - limited (decimal) representation: overflow, wrap-around
- **2038: Unix epoch rollover**
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to *TMin* in 2038
- **other related bugs**
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown "smart" warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for ( i=0; i<10; i++ ) {
    f2 += 1.0/10.0;
  }

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.8f\n", f1);
  printf("f2 = %10.8f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf ("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Memory Referencing Bug

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  -> 3.14
fun(1)  -> 3.14
fun(2)  -> 3.1399998664856
fun(3)  -> 2.00000061035156
fun(4)  -> 3.14, then segmentation fault
```

Explanation:

Saved State	4	Location accessed by fun(i)
d7 … d4	3	
d3 … d0	2	
a[1]	1	
a[0]	0	

Representing 3.14 as a Double FP Number

- **1073741824 = 0100 0000 0000 0000 0000 0000 0000 0000**
- **3.14 = 11.0010 0011 1101 0111 0000 1010 000…**
- **$(-1)^s\ M\ 2^E$**
  - S = 0 encoded as 0
  - M = 1.1001 0001 1110 1011 1000 0101 000…. (leading 1 left out)
  - E = 1 encoded as 1024 (with bias)

s	exp (11)	frac (first 20 bits)
0	100 0000 0000	1001 0001 1110 1011 1000

frac (the other 32 bits)
0101 0000 …

Memory Referencing Bug (Revisited)

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  -> 3.14
fun(1)  -> 3.14
fun(2)  -> 3.1399998664856
fun(3)  -> 2.00000061035156
fun(4)  -> 3.14, then segmentation fault
```

Saved State		4	Location accessed by fun(i)
d7 … d4	0100 0000 0000 1001 0001 1110 1011 1000	3	
d3 … d0	0101 0000 …	2	
a[1]		1	
a[0]		0	

Memory Referencing Bug (Revisited)

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  -> 3.14
fun(1)  -> 3.14
fun(2)  -> 3.1399998664856
fun(3)  -> 2.00000061035156
fun(4)  -> 3.14, then segmentation fault
```

Saved State		4	Location accessed by fun(i)
d7 … d4	0100 0000 0000 1001 0001 1110 1011 1000	3	
d3 … d0	0100 0000 0000 0000 0000 0000 0000 0000	2	
a[1]		1	
a[0]		0	

Memory Referencing Bug (Revisited)

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  -> 3.14
fun(1)  -> 3.14
fun(2)  -> 3.1399998664856
fun(3)  -> 2.00000061035156
fun(4)  -> 3.14, then segmentation fault
```

Saved State		4	Location accessed by `fun(i)`
d7 … d4	0100 0000 0000 0000 0000 0000 0000 0000	3	
d3 … d0	0101 0000 …	2	
a[1]		1	
a[0]		0	

Summary

- **As with integers, floats suffer from the fixed number of bits available to represent them**
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”
- **Mathematically equivalent ways of writing an expression may compute different results**
  - Violates associativity/distributivity
- **Never test floating point values for equality!**
- **Careful when converting between ints and floats!**

Many more details for the curious...

- **Exponent bias**
- **Denormalized values – to get finer precision near zero**
- **Distribution of representable values**
- **Floating point multiplication & addition algorithms**
- **Rounding strategies**

- **We won’t be using or testing you on any of these extras in 351.**

Normalized Values

$V = (-1)^S * M * 2^E$

s	exp	frac
	k	n

- **Condition: exp ≠ 000…0 and exp ≠ 111…1**
- **Exponent coded as *biased* value: *E* = *exp* - *Bias***
  - **exp** is an *unsigned* value ranging from 1 to 2^k-2 (k == # bits in **exp**)
  - *Bias* = 2^{k-1} - 1
    - Single precision: 127 (so *exp*: 1…254, *E*: -126…127)
    - Double precision: 1023 (so *exp*: 1…2046, *E*: -1022…1023)
  - These enable negative values for E, for representing very small values
- **Significand coded with implied leading 1: *M* = 1.xxx…x$_2$**
  - **xxx…x**: the n bits of **frac**
  - Minimum when 000…0 (*M* = 1.0)
  - Maximum when 111…1 (*M* = 2.0 – ε)
  - Get extra leading bit for "free"

Normalized Encoding Example

$V = (-1)^S * M * 2^E$

s	exp	frac
	k	n

- **Value: `float f = 12345.0;`**
  - $12345_{10} = 11000000111001_2$
    $= 1.1000000111001_2 \times 2^{13}$ (normalized form)
- **Significand:**
  - *M* = 1.$\underline{1000000111001}_2$
  - **frac**= $\underline{1000000111001}0000000000_2$
- **Exponent:** E = exp - Bias, so exp = E + Bias
  - *E* = 13
  - *Bias* = 127
  - **exp** = 140 = 10001100_2
- **Result:**

0	10001100	10000001110010000000000
s	exp	frac

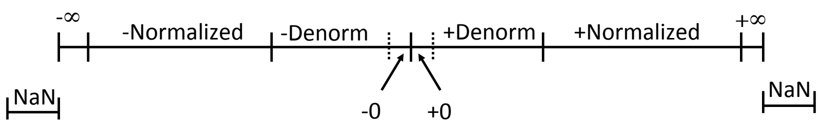
Denormalized Values

- **Condition: exp = 000…0**
- **Exponent value: *E* = exp – *Bias* + 1 (instead of E = exp – Bias)**
- **Significand coded with implied leading 0: *M* = 0.xxx…x$_2$**
  - **xxx…x**: bits of **frac**
- **Cases**
  - **exp** = 000…0, **frac** = 000…0
    - Represents value 0
    - Note distinct values: +0 and –0 (why?)
  - **exp** = 000…0, **frac** ≠ 000…0
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

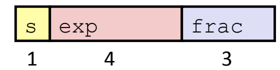
Special Values

- **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 = +∞, 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), $\infty - \infty$, $\infty * 0$

Visualization: Floating Point Encodings



Tiny Floating Point Example



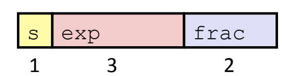
- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit.
  - the next four bits are the exponent, with a bias of 7.
  - the last three bits are the `frac`
- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

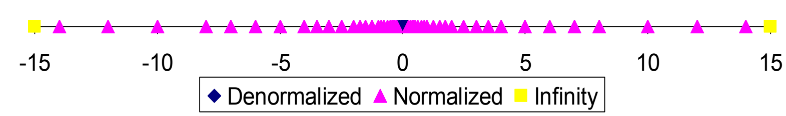
	s	exp	frac	*E*	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

Distribution of Values

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$



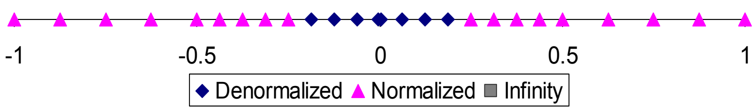
- **Notice how the distribution gets denser toward zero.**



Distribution of Values (close-up view)

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

s	exp	frac
1	3	2



University of Washington

Interesting Numbers

{single,double}

Description	exp	frac	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} * 2^{-\{126,1022\}}$
Single ≈ $1.4 * 10^{-45}$			
Double ≈ $4.9 * 10^{-324}$			
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) * 2^{-\{126,1022\}}$
Single ≈ $1.18 * 10^{-38}$			
Double ≈ $2.2 * 10^{-308}$			
Smallest Pos. Norm.	00…01	00…00	$1.0 * 2^{-\{126,1022\}}$
Just larger than largest denormalized			
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) * 2^{\{127,1023\}}$
Single ≈ $3.4 * 10^{38}$			
Double ≈ $1.8 * 10^{308}$			

University of Washington

Special Properties of Encoding

- **Floating point zero (0^+) exactly the same bits as integer zero**
  - All bits = 0
- **Can (Almost) Use Unsigned Integer Comparison**
  - Must first compare sign bits
  - Must consider $0^- = 0^+ = 0$
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

University of Washington

Floating Point Multiplication

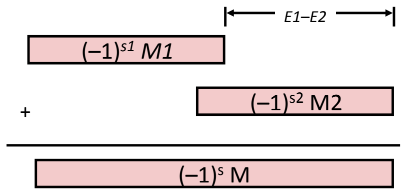
$$(-1)^{s1}\ M1\ 2^{E1} \quad * \quad (-1)^{s2}\ M2\ 2^{E2}$$

- **Exact Result: $(-1)^s\ M\ 2^E$**
  - Sign s: s1 ^ s2 // xor of s1 and s2
  - Significand M: M1 * M2
  - Exponent E: E1 + E2
- **Fixing**
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit frac precision

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 ≥ 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

Closer Look at Round-To-Even

- **Default Rounding Mode**
  - Hard to get any other kind without dropping into assembly
  - All others 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

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

- **Binary Fractional Numbers**
  - "Half way" when bits to right of rounding position = $100\ldots_2$
- **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