CSE 351
Section 4

1/26/12

Agenda

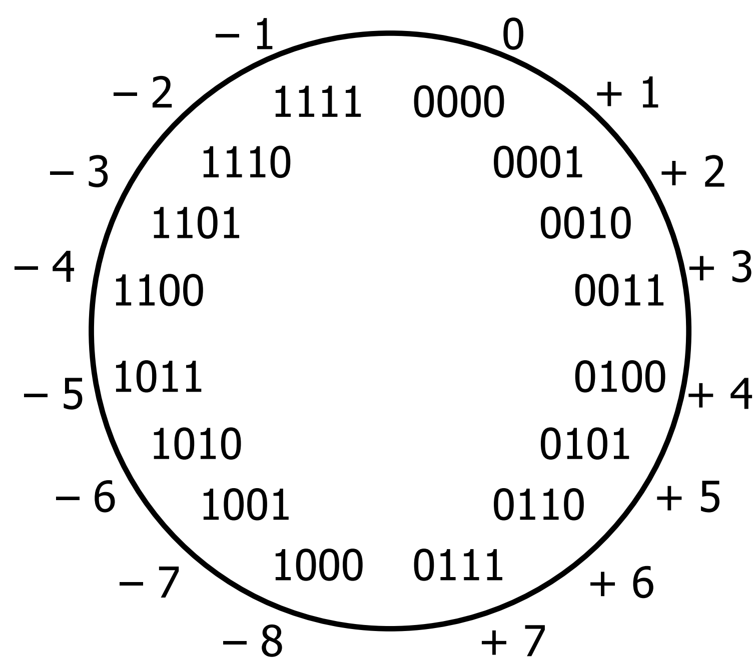
- Review integer representations
  - How they show up in C
- Shifting / Masks
- Sign Extension
- Floating Point – more detail

How can we represent negative numbers?

- Sign-and-Magnitude
  - MSB denotes sign of number, rest of bits denote magnitude
  - E.g. 1001 = -1
  - Two zeros (0000 and 1000) and you need different hardware for + and –
- One’s Complement (a.k.a. Bitwise Complement)
  - Flip all bits to get the negative of a number
  - E.g. 1110 = -1
  - Two zeros (0000 and 11111)
- Two’s Complement
  - To get the negative of a number, flip all bits and add 1
  - E.g. 1111 = -1
  - Only one zero, can use same hardware for + and -, as well as for signed/unsigned

Two’s complement

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



Two's Complement

- Why does it work?
  - The one's complement of a b-bit positive number y is $(2^b - 1) - y$
    - E.g. -3 = 1100_2 in one's complement, which is 12 if it were unsigned
      $(2^4 - 1) - 3 = 12$
  - Two's Complement adds 1 to the one's complement, thus -y is $2^b - y$ (or -x == (~x + 1))
    - $-y$ and $2^b - y$ are equal mod 2^b (have the same remainder when divided by 2^b)
    - Ignoring carries is equivalent to doing arithmetic mod 2^b

Mapping Signed -> Unsigned

Bits	Signed		Unsigned
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	-7		9
1010	-6		10
1011	-5	+16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Signed vs. Unsigned in C

- Constants
  - Default = signed integers
  - Unsigned if they have “U” as a suffix
    - E.g. 0U, 1234567U
  - Size can by typed too
    - E.g. 1234567890123456ULL
- Casting

```
int tx, ty;
unsigned ux, uy;
```

  - Explicit casting

```
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting (careful!)

```
tx = ux;
uy = ty;
```

Casting Surprises

- If you mix unsigned and signed in a single expression, signed values are implicitly cast to unsigned
  - Including comparison operations <, >, ==, <=, >=

Examples for 32-bit: TMIN = -2,147,483,648 TMAX = 2,147,483,647

Constant 1	Constant 2	Evaluated As	Relation between C1 and C2
0	0U	Unsigned	==
-1	0	Signed	<
-1	0U	Unsigned	>
2147483647	-2147483648	Signed	>
2147483647U	-2147483648	Unsigned	<
-1	-2	Signed	>
0U – 1	-2	Unsigned	>
2147483647	2147483648U	Unsigned	<
2147483647	(int) 2147483648U	Signed	>

Shift Operations

- Left Shift: x << y
  - Shift bit-vector x left by y positions
  - Throw away extra bits on left, fill with 0's on right
  - Each shift left by 1 bit is the same as multiplying by 2
    - So x << y is the same as $x * 2^y$

Argument x	01100010
<< 3	00010000
Logical >> 2	00011000
Arithmetic >> 2	00011000

- Right Shift: x >> y
  - Shift bit-vector x right by y positions
  - Throw away extra bits on right
    - Logical shift (for unsigned): Fill with 0's on left
    - Arithmetic shift (for signed): Fill with whatever was MSB on left – Maintain the sign of x
  - Each shift right by 1 is the same as dividing by 2

Argument x	10100010
<< 3	00010000
Logical >> 2	00101000
Arithmetic >> 2	11101000

Masking

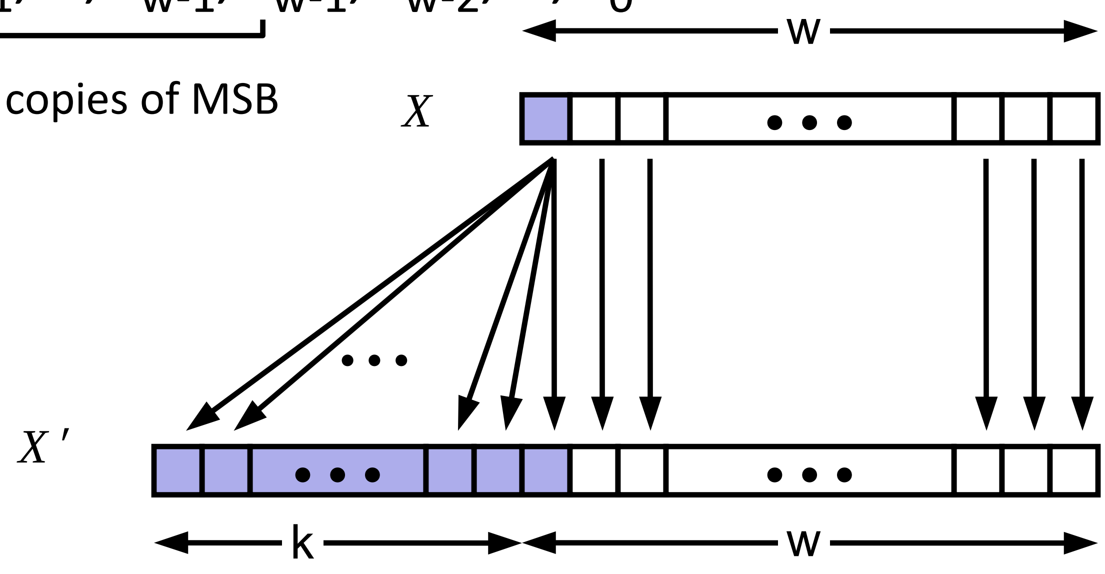
- What if you need to extract the 2nd most significant byte of an integer (i.e. bits 16 through 23)?
  - First shift: x >> 16
  - Then mask: (x >> 16) & 0xff

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

- Extracting the sign bit
  - (x >> 31) & 1
  - Need the "& 1" to clear out all other bits except the LSB

Sign Extension

- Given a w-bit signed integer x, convert to a (w+k)-bit signed integer with the same value
- Rule: Make k copies of sign bit
  - $X2 = \underbrace{X_{w-1},\ldots,X_{w-1}}_{\text{k copies of MSB}},X_{w-1},X_{w-2},\ldots,X_0$



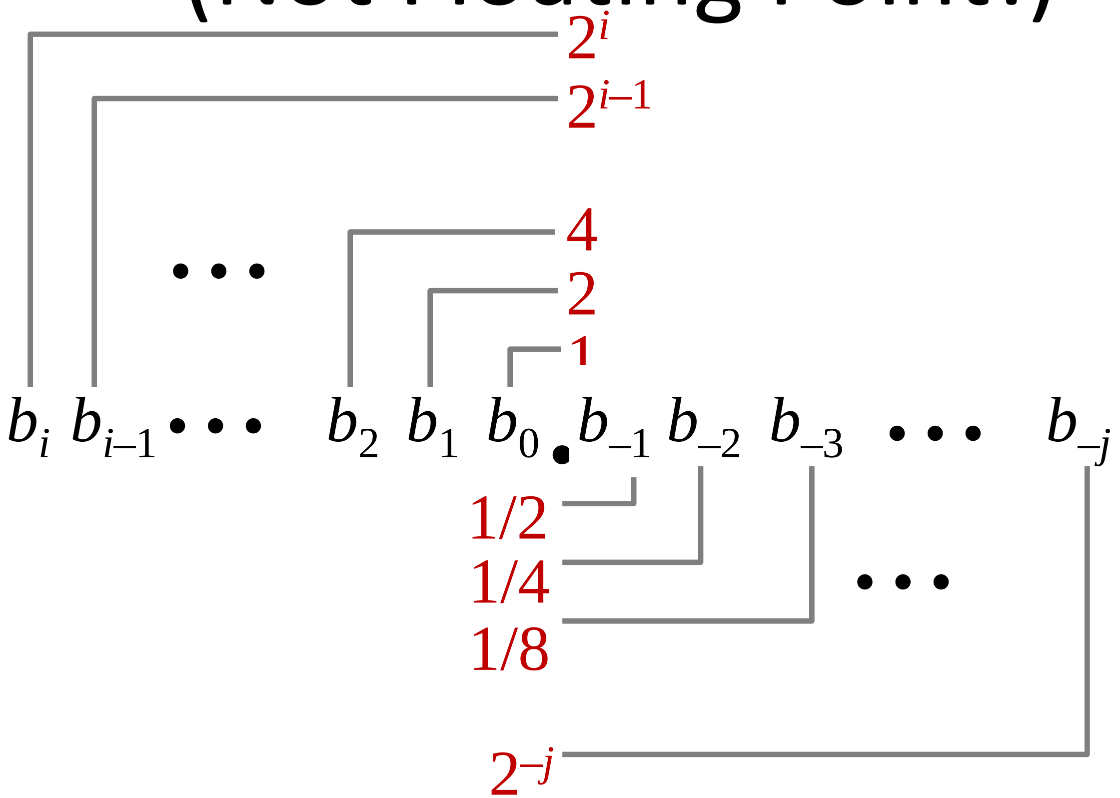
Sign Extension Example

```
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

C automatically performs sign extension

Fractional Binary Numbers (Not Floating Point!)



Bits to right of "binary point" represent fractional powers of 2

Fractional Binary Numbers Examples

- What are these numbers in binary?
  - 5 and ¾ 101.11_2
  - 2 and 7/8 10.111_2
  - 63/64 0.111111_2
- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0

Representable Numbers

- Limitation
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations
- Value Representation

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Fixed Point Representation

- Pick where you want to put the decimal point
- The position of the binary point affects the <u>range</u> and <u>precision</u>
  - Range: difference between the largest and smallest representable numbers
  - Precision: smallest possible difference between any two numbers
- Pro
  - Simple: The same hardware that does integer arithmetic can do fixed point arithmetic
    - In fact, the programmer can use ints with an implicit fixed point
      - E.g. int balance; // number of pennies in the account
    - ints are just fixed point numbers with the binary point to the right of the LSB
- Con
  - There is no good way to pick where the fixed point should be
    - Sometimes you need range, sometimes you need precision
    - More range = less precision and vice versa

Floating Point Representation

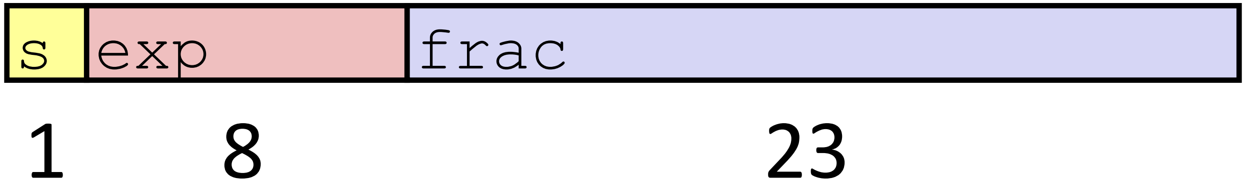
$$(-1)^S * M * 2^E$$

- Sign bit S determines whether number is negative or positive
- Mantissa M (aka Significand aka “Frac”) normally a fractional value in range [1.0,2.0).
- Exponent E weights value by power of two
- Encoding
  - MSB is sign bit S
  - frac field encodes M (but is not equal to M)
  - exp field encodes E (but is not equal to E)

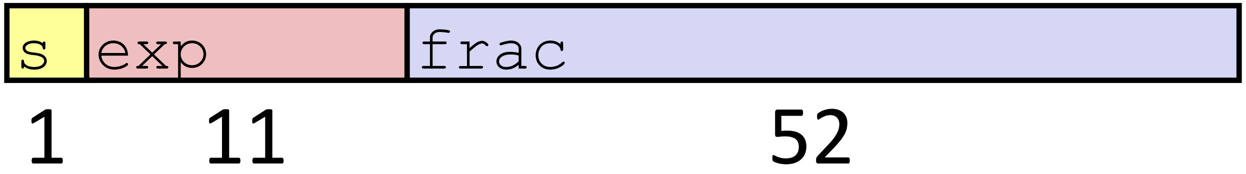
s	exp	frac

Precisions

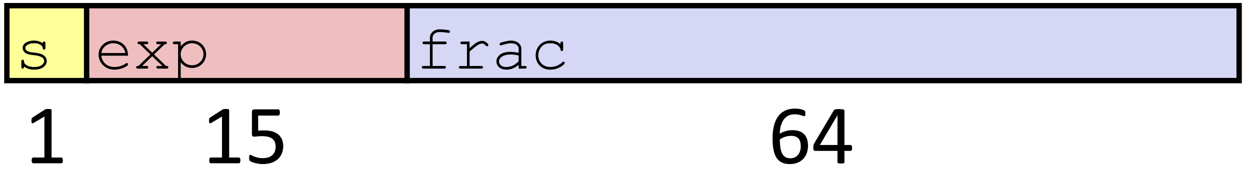
- Single precision (float): 32 bits



- Double Precision (double): 64 bits



- Extended Precision: 80 bits (Intel only)



Normalization, Bias and Special Values

- "Normalized" means mantissa has form 1.xxxx
  - $0.011 * 2^5$ and $1.1 * 2^3$ represent the same number, but the latter makes better use of available bits
  - Since we know the mantissa starts with a 1, don't bother to store it
  - Therefore, when the mantissa is 1.xxxxx, M (i.e. `frac`) contains xxxxx
- The exponent field does not contain the exponent of the number, but the offset from a bias
  - `exp` = E + Bias
  - Bias = $2^{|\texttt{exp}|-1} - 1$ where |`exp`| = size of `exp` field
  - (e.g. 127 is the bias for an 8 bit `exp`)
- Special Values
  - The float value 00….0 represents zero
  - Exp = 11…1 and Mantissa = 00…0 represents infinity
    - E.g. 10.0 / 0.0
  - Exp = 11…1 and Mantissa != 00…0 represents NaN
    - E.g. 0 * Infinity

Floating Point Example

s	exp (8)	frac (23)

- How is float 12345.0 represented?
- Value

$$12345.0_{10} = 11000000111001_2$$

$$= 1.1000000111001_2 * 2^{13}$$

Floating Point Example

s	exp (8)	frac (23)

- How is float 12345.0 represented?
- Value

  $12345.0_{10} = 11000000111001_2$

  $= 1.1000000111001_2 * 2^{13}$

- Mantissa

  $M = 1.\underline{1000000111001}_2$

  frac = $\underline{1000000111001}0000000000_2$ (Need to extend to fill all 23 bits)

Floating Point Example

s	exp (8)	frac (23)

- How is float 12345.0 represented?
- Value

  $12345.0_{10} = 11000000111001_2$

  $= 1.1000000111001_2 * 2^{13}$

- Mantissa

  $M = 1.\underline{1000000111001}_2$

  $\text{frac} = \underline{1000000111001}0000000000_2$ (Need to extend to fill all 23 bits)

- Exponent

  $E = 13$

  $\text{Bias} = 2^7 - 1 = 127$

  $\text{exp} = 140_{10} = 10001100_2$

Floating Point Operations

- Basic Idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent is too large
    - Possibly round to fit into `frac`
- $\mathbf{x} +_{\mathbf{f}} \mathbf{y} = \mathbf{Round(x + y)}$
- $\mathbf{x} *_{\mathbf{f}} \mathbf{y} = \mathbf{Round(x * y)}$

Floating Point Multiplication

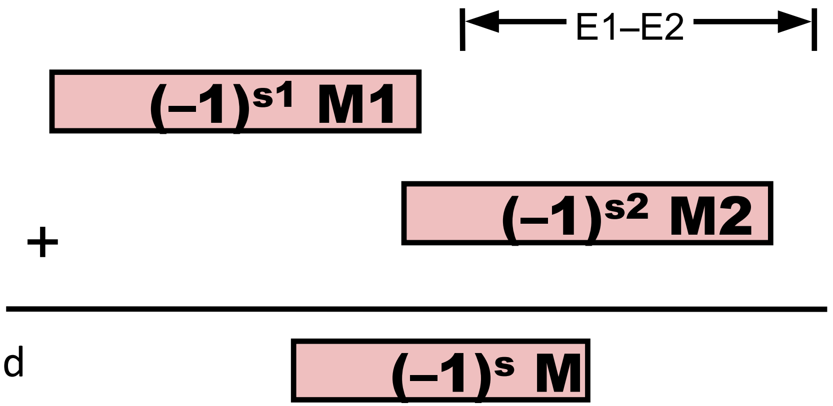
$(-1)^{S1}\ M_1\ 2^{E1} * (-1)^{S2}\ M_2\ 2^{E2}$

- Exact Result
  - Sign = S_1 ^ S_2
  - Mantissa: $M_1 * M_2$
  - Exponent: $E_1 + E_2$
- Fixing
  - If $M \geq 2$, $M = M >> 1$, $E = E + 1$
  - If E is out of range, overflow
  - Round M to fit `frac` precision

Floating Point Addition

$$(-1)^{S1}\, M_1\, 2^{E1} + (-1)^{S2}\, M_2\, 2^{E2}$$

Assume $E_1 > E_2$



- Exact Result
  - Sign S, Mantissa M:
    - Shift sign and mantissa of first value left by the difference of the exponents. This makes exponents equal, so you can add the signed mantissas.
  - Exponent E: E_1
- Fixing
  - If $M \geq 2$, $M = M >> 1$, $E = E + 1$
  - If $M < 1$, $M = M << k$, $E = E - k$
  - Overflow if E is out of range
  - Round M to fit `frac` precision

Rounding Errors

- Since we round on every operation, the operations are not really associate or distributive
  - Let a = 1.52342, b = 6.2342342, c = 2.2523555
    - (a + b) + c = 10.01000**9700000001**

      a + (b + c) = 10.01000**9699999999**
    - a * (b + c) = 12.92864048077**4000**

      a * b + a * c = 12.92864048077**4002**

Floating Point Values 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 == f2? %s\n”, f1 == f2 ? “yes” : “no”);
    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;
}
```

Floating Point Values 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 == f2? %s\n", f1 == f2 ? "yes" : "no");
    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 == f2? no
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

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. 0.1)
  - Can also lose precision, unlike ints
    - "Every operation gets a slightly wrong result"
- Mathematically equivalent ways of writing an expression may compute different results
- NEVER test floating point values for equality!