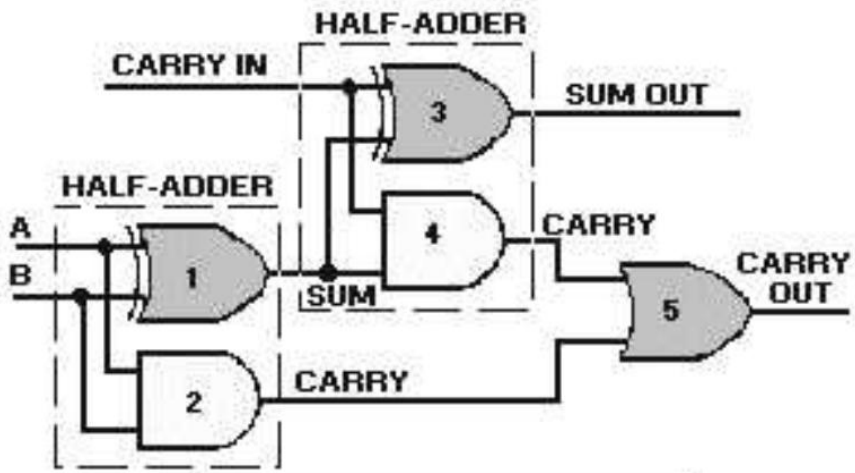
Bit operations



A	B	CARRY IN	SUM OUT	CARRY OUT
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

1 and 3 ➔ exclusive OR (^)
2 and 4 ➔ and (&)
5 ➔ or (|)

01100 carry*
0110 a
0111 b
01101 a+b

* Always start with a carry-in of 0

Did it work?
What is a?
What is b?
What is a+b?
What if 8 bits instead of 4?

Integer Representation

- Different encoding scheme than float
- *Total number of values: 2^w
  - where w is the bit width of the data type
- The left-most bit is the sign bit if using a signed data type (typically… B2T).
- Unsigned → non-neg numbers (>=0)
  - Minimum value: 0
  - *Maximum value: 2^w-1
- Signed → neg, zero, and pos numbers
  - *Minimum value: -2^{w-1}
  - *Maximum value: 2^{w-1}-1

* Where w is the bit width of the data type

Integer Decoding

- Binary to Decimal (mult of powers)
- Unsigned = simple binary = B2U
  - You already know how to do this :o)
    - 0101 = 5, 1111 = F, 1110 = E, 1001 = 9
- Signed = two's complement = B2T*
  - 0 101 = positive number; same as B2U = 5
  - 1 111 = $-1*2^3 + 7 = -8 + 7 = -1$
  - 1 110 = $-1*2^3 + 6 = -8 + 6 = -2$
  - 1 001 = $-1*2^3 + 1 = -8 + 1 = -7$
  - Another way, if sign bit = 1, then it's a negative number and to get the absolute value of that number, you:
    - invert bits and add 1
    - right to left, leave alone to first 1, then invert rest
    - 1010 = lookup says -6… verify both ways

* reminder: left most bit is sign bit

CODE	B2U	B2T
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

B2O & B2S

Note that +0 and −0 return TRUE when tested for zero, FALSE when tested for non-zero.

- One's complement = bit complement of B2U
- Signed Magnitude = left most bit set to 1 with B2U for the remaining bits
- Both include neg values
- Min/max = $-(2^{w-1}-1)$ to $2^{w-1}-1$
- Pos and neg zero
- Difficulties with arithmetic options

CODE	B2U	B2T	B2O	B2S
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	**8**	**-8**	**-7**	**-0**
1001	9	-7	-6	-1
1010	10	-6	-5	-2
1011	11	-5	-4	-3
1100	12	-4	-3	-4
1101	13	-3	-2	-5
1110	14	-2	-1	-6
1111	15	-1	-0	-7

Signed vs Unsigned

sign2unsign.c

- Casting…
- Signed to unsigned…
- Unsigned to signed…

*** Changes the meaning of the value, but not the bit representation

- Notice, the difference of 16 i.e. left most bit →
  - Unsigned = $2^3 = 8$
  - Signed = $-2^3 = -8$

CODE	B2U	B2T
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**

What is the largest possible value for short? 16-bits… see <limits.h>
When add 1 to -1 get 0; when add 1 to 4,294,967,295, you get zero. Why?
FFFF + 1 = 0000… overflow warning but okay i.e. -1 + 1 = 0… all is good

Signed vs Unsigned (cont)

- When an operation is performed where one operand is signed and the other is unsigned, *C implicitly casts the signed argument to unsigned* and performs the operations assuming the numbers are nonnegative.
  - Since bit representation does not change, it really doesn't matter arithmetically
  - However… relational operators have issues

EXPRESSION	TYPE	EVALUATION
0 = = 0u	unsigned	1
-1 < 0	signed	1
-1 < 0u	unsigned	0*
2147483647 > -2147483647-1	signed	1
2147483647u > -2147483647-1	unsigned	0*
-1 > -2	signed	1
(unsigned) -1 > -2	unsigned	1

← 1 = TRUE and 0 = FALSE

Sign Extend

- Already have an intuitive sense of this

w = 8
-27 B2T => B2U for +27 = 00011011
=> invert + 1 for -27 = 11100101

w=16 sign extend
+27 00000000 00011011
-27 11111111 11100101

- For unsigned
  - Fill to left with zero
- For signed
  - Repeat sign bit

Truncation

- Drops the high order w-k bytes when truncating a w-bit number to a k-bit number
- 4-bit to 3-bit truncation

HEX		UNSIGNED – B2U		TWO'S COMP – B2T	
orig	trunc	orig	trunc	orig	trunc
0	0	0	0	0	0
2	2	2	2	2	2
9	1	9	1	-7	1
B	3	11	3	-5	3
F	7	15	7	-1	-1

Integer Addition

- Signed or unsigned… that is the question!

- **Unsigned** 4-bit BTU
  - 0 to 15 are valid values
  - Only checking carry-out

- **Signed** 4-bit B2T
  - -8 to 7 are valid values
  - Checking carry-in AND carry-out

x	y	x+y	result
8 1000	5 0101	13 1101	13 ok
8 1000	7 0111	15 1111	15 ok
12 1100	5 0101	17 10001	1 OF

Negative overflow when $x+y < -2^{w-1}$
Positive overflow when $x+y > 2^{w-1}-1$

x	y	x+y	result
-8 1000	-5 1011	-13 1 0011	3 negOF
-8 1000	-8 1000	-16 1 0000	0 negOF
-8 1000	5 0101	-3 1 1101	-3 ok
2 0010	5 0101	7 0 0111	7 ok
5 0101	5 0101	10 0 1010	-6 posOF

B2T integer negation

- How determine a negative value in B2T?
  - Reminder: B2U=B2T for positive values
  - B2U → invert the bits and add one
- Two's complement negation
  - **-2^{w-1} is its own additive inverse**
  - Other values are negated by integer negation
  - Bit patterns generated by two's complement are the same as for unsigned negation

GIVEN			NEGATION		
HEX	binary	base 10	base 10	binary*	HEX
0x00	0b00000000	0	0	0b00000000	0x00
0x40	0b01000000	64	-64	0b11000000	0xC0
0x80	0b10000000	-128	-128	0b10000000	0x80
0x83	0b10000011	-125	125	0b01111101	0x7D
0xFD	0b11111101	-3	3	0b00000011	0x03
0xFF	0b11111111	-1	1	0b00000001	0x01
			*binary = invert the bits and add 1		

See next page: signvsuns.c

Sign/Unsign+Negation example

```
#include <stdio.h>
#include <limits.h>
void main()    {
  int n = 0;
  printf("neg of %d is %d ",n,-n);
  n = 64;
  printf("\nneg of %d is %d ",n,-n);
  n = -64;
  printf("\nneg of %d is %d ",n,-n);
  n = INT_MIN;
  printf("\nneg of %d is %d ",n,-n);

  unsigned int a = 0;
  printf("\n\n0 - 1 unsigned is %u ",a-1);
  printf("\n unsigned max is %u ", UINT_MAX);
  a = 5;
  printf("\nnegof unsigned %d is %u",a, -a);   }
```

```
% signvsuns
negation of 0 is 0
negation of 64 is -64
negation of -64 is 64
negation of -2147483648 is -2147483648


0 - 1 unsigned is 4294967295
 unsigned max is 4294967295
negation of unsigned 5 is 4294967291
```

Rounding

y	round down (towards −∞)	round up (towards +∞)	round towards zero	round away from zero	round to nearest
+23.67	+23	+24	+23	+24	+24
+23.50	+23	+24	+23	+24	+24
+23.35	+23	+24	+23	+24	+23
+23.00	+23	+23	+23	+23	+23
0	0	0	0	0	0
−23.00	−23	−23	−23	−23	−23
−23.35	−24	−23	−23	−24	−23
−23.50	−24	−23	−23	−24	−24
−23.67	−24	−23	−23	−24	−24

Rounding – our system

```
#include <stdio.h>
void main()
{
   float y[9] = {23.67,  23.50,  23.35,  23.00,
                  0, -23, -23.35, -23.5, -23.67};
   int i;
   for (i=0;i<9;i++) {
      printf("y = %.4f %.2f %.1f %.0f\n",
                  y[i],y[i],y[i],y[i]);
   }
}
```

```
//OUTPUT

y = 23.6700   23.67   23.7   24
y = 23.5000   23.50   23.5   24
y = 23.3500   23.35   23.4   23
y = 23.0000   23.00   23.0   23
y = 0.0000   0.00   0.0   0
y = -23.0000   -23.00   -23.0   -23
y = -23.3500   -23.35   -23.4   -23
y = -23.5000   -23.50   -23.5   -24
y = -23.6700   -23.67   -23.7   -24
```

ROUND TO NEAREST

Integer multiplication

- 1 * 1 = 1
- 1 * 0 = 0 * 1 = 0 * 0 = 0

8-bit multiplication

-4 ->				1	1	1	1	1	1	0	0	
9 ->				0	0	0	0	1	0	0	1	
				1	1	1	1	1	1	0	0	
	1	1	1	1	1	1	0	0				
	1	1	1	1	1	1						<- carry
	1	0	0	0	1	1	0	1	1	1	0	0 = -36?

8-bit multiplication

12 ->				0	0	0	0	1	1	0	0	
9 ->				0	0	0	0	1	0	0	1	
				0	0	0	0	1	1	0	0	
	0	0	0	0	1	1	0	0				
												<- carry
				0	1	1	0	1	1	0	0	= 108?

B2T 8-bit range → -128 to 127

B2U 8-bit range → 0 to 255

B2U = 252*9 = 2268 (too big)

B2T = same

Typically, if doing 8-bit multiplication, you want 16-bit product location i.e. 2w bits for the product

Integer multiplication (cont)

- Unsigned i.e. simple binary
  - For x and y, each with the same width (w)
  - x*y yields a w-bit value given by the low-order w bits of the 2w-bit integer product
    - Equivalent to computing the product (x*y) modulo 2^w
    - Result interpreted as an unsigned value
- Signed = similar, but result interpreted signed value

binary	unsigned	two's comp	result
0111*0011	7*3=21=0001**0101**	same	**0101**
	21 mod 16 = 5	same	
1001*0100	9*4=36=0010**0100**	-7*4=-28=1110**0100**	**0100**
fyi	36 mod 16 = 4	-28 mod 16 = 4	
1100*0101	12*5=60=0011**1100**	-4*5=-20=1110**1100**	**1100**
	60 mod 16 = 12	-20 mod 16 = -4	
1101*1110	13*14=182=1011**0110**	-3*-2=6=0000**0110**	**0110**
	182 mod 16 = 6	6 mod 16 = 6	
1111*0001	15*1=15=0000**1111**	-1*1=-1=1111**1111**	**1111**
	15 mod 16 = 15	-1 mod 16 = -1	

Multiply by constants

- First case: Multiplying by a power of 2
  - Power of 2 represented by k
  - So k zeroes added in to the right side of x
    - Shift left by k: x<<k
  - Overflow issues the same as x*y
- General case
  - Every binary value is an addition of powers of 2
  - Has to be a run of one's to work
    - Where n = position of leftmost 1 bit in the run and m=the rightmost
  - "multiplication of powers" where K = 7 = 0111 = $2^2+2^1+2^0$
    - **(x<<n)+(x<<n-1) + … + (x<<m)**
  - Also equal to $2^3 - 2^0 = 8 - 1 = 7$
    - **(x<<n+1) - (x<<m)**… subtracting
- Why looking at it this way?
  - Shifting, adds and subtracts are quicker calculations than multiplication (2.41)
  - Optimization for C compiler

What is x*4 where x = 5?
x = 5 = 00000101
4 = 2^k, so k = 2
x<<k = 00010100 = 20
What if x = -5?

x = 5, **n = 2 and m = 0**
x*7 = 35?
x<<2 + x<<1+x<<0
00010100
00001010
<u>00000101</u>
00100011 = 35? YES
OR
00101000
<u>11111011</u>
00100011

Multiply by constants (cont)

Assume x = 5

- What if the bit position n is the most significant bit? **assuming 4 bits for this part (easier!)**
  - Since (x<<n+1) – (x<<m) and shifting n+1 times gives zero, then formula is –(x<<m)
- What if K is negative? i.e. K = constant = -6
  - **back to 8-bits**
  - 0000 0110 = +6
  - Using (x<<n+1) – (x<<m) where n=2 and m=1
  - If want -6 then negate formula which gives:
    - (x<<m) – (x<<n+1)
    - 2^1 - 2^3 which equals 2 – 8 = -6 (good) so…
    - (x<<1) – (x<<3) = 0000 1010 – 0010 1000
    - To add, negate the 2nd value → 1101 1000
    - And 0000 1010 + 1101 1000 = 1110 0010 = -30? YES!

Dividing by powers of 2

- Even slower than integer multiplication
- Dividing by powers of 2 → right shifting
  - Logical - unsigned
  - Arithmetic – two's complement
- Integer division always rounds toward zero (i.e. truncates)
  - C float-to-integer casts round towards zero.
  - These rounding errors generally accumulate

8÷2

		1	0	0
10	1	0	0	0
	1	0		
			0	0
			0	0
			0	0

9÷2

		1	0	0
10	1	0	0	1
	1	0		
			0	1
			0	0
				1

12÷4

			1	1
100	1	1	0	0
	1	0	0	
		1	0	0
		1	0	0
				0

15÷4

			1	1
100	1	1	1	1
	1	0	0	
		1	1	1
		1	0	0
			1	1

What if the binary numbers are B2T?

Unsigned Integer Division by 2^k

- Logical right shift by k (x>>k)
  - x divided by 2^k and then rounding toward zero
  - Pull in zeroes
  - Example x = 12
    - 1100>>0110>>0011>>0001>>0000
    - 12/2 = 6/2 = 3/2 = 1/2 = 0
    - $12/2^1 = 12/2^2 = 12/2^3 = 12/2^4$
  - Example x = 15
    - 1111>>0111>>0011>>0001>>0000
    - 15/2 = 7/2 = 3/2 = 1/2 = 0
    - $15/2^1 = 15/2^2 = 15/2^3 = 15/2^4$
  - Example
    - $12/2^2 = 3$
    - $10/2^2 = 2$
    - $12/2^3 = 1$

Signed Integer Division by 2^k

- Two's complement
  - Sign extend for arithmetic shift
  - Examples
  - Rounds down (rather than towards zero)
    - -7/2 should yield -3 rather than -4

$x = -82$
$y = 2^k$

k	Binary								Decimal	$-82/2^k$
0	1	0	1	0	1	1	1	0	-82	-82.000000
1	1	1	0	1	0	1	1	1	-41	-41.000000
2	1	1	1	0	1	0	1	1	-21	-20.500000
3	1	1	1	1	0	1	0	1	-11	-10.250000
4	1	1	1	1	1	0	1	0	-6	-5.125000
5	1	1	1	1	1	1	0	1	-3	-2.562500

For now…

y	same binary	Dec (rd)	(x+y-1)/y
1		-82	-82.000000
2		-41	-40.500000
4		-20	-19.750000
8		-10	-9.375000
16		-5	-4.187500
32		-2	-1.593750

Corrected…

Signed Integer Division by 2^k (cont)

- By adding a "bias" of y-1 before performing the arithmetic right shift causes the result to be correctly rounded "upward" i.e. towards zero
- So using the given property, $\lceil x / y \rceil = \lfloor (x + y - 1) / y \rfloor$
  - For integers x and y such that y > 0
  - Example: x=-30, y=4
    - x+y-1=-27 and -30/4 (ru) = -7 = -27/4 (rd)
  - Example x=-32 and y=4
    - x+y-1=-29 and -32/4 (ru) = -8=-29/4 (rd)
- Equivalent to $x/2^k$
  - y = 1<<k
    - $4=2^k$, so if y = 4 then k = 2, so shift 1 left by 2 = 100 is 4
  - In C ➔ (x<0 ? x+(1<<k)-1 : x) >> k

Bit-level operations in C

- Can be applied to any "integral" data type
  - One declared as type char or int
    - with or without qualifiers (ex. short, long, unsigned)
- How use?
  - Expand hex arguments to their binary representations
  - Perform binary operation
  - Convert back to hex
- NOTE: the expression ~0 will yield a mask of all ones, regardless of the word size of the machine; same as 0xFFFFFFFF for a 32-bit machine, but such code is not portable.

value of x	machine rep	mask	type of x and mask	c expr	result	note
153 (base 10)	0b10011001 == 0x99	0b10000000 == 0x80	char	x & mask	0b10000000 == 0x80	2^7 = 128
				mask >> 1	0b01000000 == 0x40	2^6 = 64 (etc)
		0b01000000 == 0x40		x & mask	0b00000000 == 0x00	
153 (base 10)	0b10011001 == 0x99	0b10000000 == 0x80	int	x & mask	0b10000000 == 0x80	same
				x << 1	???	

Shift operations

- Shifting bit patterns
  - What if shift > sizeof variable type? ←"undefined"
    - "warning: right shift count >= width of type"
    - k (shift) mod w (width)
  - What if shift using a negative number? ←"undefined"
    - "warning: right shift count is negative"
    - w (width) – k (neg shift)
  - Left shift
    - Always brings in zero, but
      - What if left shift a signed value? Oh well…
  - Right shift
    - Logical – is unsigned; always brings in zeros
    - Arithmetic – is signed; repeats left most bit (sign bit)
      - sign specification default in C is "signed"
      - Almost all compilers/machines do repeat sign (most)
- Examples (shiftex.c)
  - Note: difference between 0xF0F0 and 0xF0F0F0F0?

Boolean Algebra

- Boolean algebra has many of the same properties as arithmetic over integers
  - *+ and &|
    - Multiplication distributes over addition
      - a*(b+c) = (a*b) + (a*c)
    - Boolean operation & distributes over |
      - a&(b|c) = (a&b) | (a&c)
    - Boolean operation | distributes over &
      - a|(b&c) = (a|b) & (a|c)
    - CANNOT distribute addition over multiplication
      - a+(b*c) <> (a+b) * (a+c)… for all integers
- Boolean ring – commonalities with integers
  - Every value has an additive inverse –x, such that
    x + -x = 0
  - a^a = 0 each element is its own additive inverse
    - (a^b)^a = b above holds even in different ordering
    - Consider (swap):
      - *y = *x ^ *y;
      - *x = *x ^ *y;
      - *y = *x ^ *y;

Don't worry about dereferencing issues… just substitute
If *y = *x^*y, then the next line is equal to
*x = *x ^ (*x ^ *y) so *x = *y
And *y = (*x^*x^*y) ^ *x^*y = *x

IEEE floating point

- IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms
- Limited range and precision (finite space)
- Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers.
- Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Floating Point

- "real numbers" having a decimal portion != 0
- Example: 123.14 base 10
  - Meaning:
    - $1*10^2 + 2*10^1 + 3*10^0 + 1*10^{-1} + 4*10^{-2}$
  - Digit format: $d_m d_{m-1} \ldots d_1 d_0 \,.\, d_{-1} d_{-2} \ldots d_{-n}$
  - dnum → summation_of(i = -n to m) $d_i * 10^i$
- Example: 110.11 base 2
  - Meaning:
    - $1*2^2 + 1*2^1 + 0*2^0 + 1*2^{-1} + 1*2^{-2}$
  - Digit format: $b_m b_{m-1} \ldots b_1 b_0 \,.\, b_{-1} b_{-2} \ldots b_{-n}$
  - bnum → summation_of(i = -n to m) $b_i * 2^i$
- "." now a "binary point"
- In both cases, digits on the left of the "point" are weighted by positive power and those on the right are weighted by negative powers

Floating Point

- Shifting the binary point one position left
  - Divides the number by 2
  - Compare 101.11 base 2 with 10.111 base 2
- Shifting the binary point one position right
  - Multiplies the number by 2
  - Compare 101.11 base 2 with 1011.1 base 2
- Numbers 0.111…11 base 2 represent numbers just below 1 → 0.111111 base 2 = 63/64
- Only finite-length encodings
  - 1/3 and 5/7 cannot be represented exactly
- Fractional binary notation can only represent numbers that can be written **$x * 2^y$** i.e. $63/64 = 63*2^{-6}$
  - Otherwise, approximated
  - Increasing accuracy = lengthening the binary representation but still have finite space

Practice Page

- Fractional value of the following binary values:
  - .01 =
  - .010 =
  - 1.00110 =
  - 11.001101 =
- 123.45 base 10
  - Binary value =
  - FYI also equals:
    - 1.2345×10^{2} is normalized form
    - 12345×10^{-2} uses significand/mantissa/coeefficient and exponent

Floating point example

- Put the decimal number 64.2 into the IEEE standard single precision floating point representation…

  SEE HANDOUT

IEEE standard floating point representation

- The bit representation is divided into 3 fields
  - The single sign bit s directly encodes the sign s
  - The k-bit exponent field encodes the exponent
    - exp = $e_{k-1}\ldots e_1 e_0$
  - The n-bit fraction field encodes the significand M (but the value encoded also depends on whether or not the exponent field equals 0… later)
    - frac = $f_{n-1}\ldots f_1 f_0$
- Two most common formats
  - Single precision (float)
  - Double-Precision (double)

	Sign	Exponent	Fraction	Bias
Single Precision (4 bytes)	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision (8 bytes)	1 [63]	11 [62-52]	52 [51-00]	1023

The sign bit and the exponent

- The sign bit is as simple as it gets.
  - 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

- The exponent field needs to represent both positive and negative exponents.
  - A *bias* is added to the actual exponent in order to get the stored exponent.
  - For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.
  - For double precision, the exponent field is 11 bits, and has a bias of 1023.

More on the “bias”

- In IEEE 754 floating point numbers, the exponent is biased in the engineering sense of the word – the value stored is offset from the actual value by the exponent bias.
- **Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder.**
- To solve this problem the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison.
- By arranging the fields so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits, the resulting value will be ordered properly, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating point numbers using fixed point hardware.
- When interpreting the floating-point number, the bias is subtracted to retrieve the actual exponent.
- For a single-precision number, an exponent in the range −126 .. +127 is biased by adding 127 to get a value in the range 1 .. 254 (0 and 255 have special meanings).
- For a double-precision number, an exponent in the range −1022 .. +1023 is biased by adding 1023 to get a value in the range 1 .. 2046 (0 and 2047 have special meanings).

The fraction

- Typically called the “significand”
- Represents the precision bits of the number.
- It is composed of an implicit (i.e. hidden) leading bit and the fraction bits.
- In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in *normalized* form.
  - This basically puts the radix point after the first non-zero digit (see previous example)

FYI: A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa/significand has effectively 24 bits of resolution, by way of 23 fraction bits.

Putting it all together

- So, to sum up:
  - The sign bit is 0 for positive, 1 for negative.
  - The exponent's base is two.
  - The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
  - The first bit of the mantissa/significand is typically assumed to be 1.*f*, where *f* is the field of fraction bits.

Another Example

- π, rounded to 24 bits of precision, has:
  - sign = 0 ;
  - *e* = 1 ;
  - *f* = 11001001000011111011011 (including the hidden bit)
  - The sum of the exponent bias (127) and the exponent (1) is 128, so this is represented in single precision format as

0 10000000 10010010000111111011011
(excluding the hidden bit) = 0x40490FDB

- In binary single-precision floating-point, this is represented as *s* = 1.10010010000111111011011 with *e* = 1. This has a decimal value of
- **3.141592**7410125732421875, whereas a more accurate approximation of the true value of π is
- **3.14159265358979323846264338327950**... The result of rounding differs from the true value by about 0.03 parts per million, and matches the decimal representation of π in the first 7 digits. **The difference is the discretization error and is limited by the machine epsilon.**

Why are we doing this?

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

Special values

- The hardware that does arithmetic on floating point numbers must be constantly checking to see if it needs to use a hidden bit of a 1 or a hidden bit of a 0 (for 0.0)
- Zero could be 0x00000000 or 0x80000000
  - What number(s) cannot be represented because of this?

	s	e	f	hidden bit
0.0	0 or 1	all zero	all zero	0
subnormal	0 or 1	all zero	not all zero	0
normalized	0 or 1	>0	any bit pattern	1
+infinity	0	11111111	00000… (0x7f80 0000)	
-infinity	1	11111111	00000… (0xff80 0000)	
NaN*	0 or 1	0xff	anything but all zeros	

*Not a Number

5-bit floating point representation with one sign bit, two exponent bits (k=2) and two fraction bits (n=2 so 2^n=4); the exponent bias is 2^{2-1}-1 = 1

bits	e	E	2^E	f	M	2^ExM	V	Decimal
0 00 00	0	0	1	0/4	0/4	0/4	0	0.00
0 00 01	0	0	1	1/4	1/4	1/4	1/4	0.25
0 00 10	0	0	1	2/4	2/4	2/4	1/2	0.50
0 00 11	0	0	1	3/4	3/4	3/4	3/4	0.75
0 01 00	1	0	1	0/4	4/4	4/4	1	1.00
0 01 01	1	0	1	1/4	5/4	5/4	5/4	1.25
0 01 10	1	0	1	2/4	6/4	6/4	3/2	1.50
0 01 11	1	0	1	3/4	7/4	7/4	7/4	1.75
0 10 00	2	1	2	0/4	0/4	8/4	2	2.00
0 10 01	2	1	2	1/4	1/4	10/4	5/2	2.50
0 10 10	2	1	2	2/4	2/4	12/4	3	3.00
0 10 11	2	1	2	3/4	3/4	14/4	7/2	3.50
0 11 00	--	--	--	--	--	--	inf	--
0 11 01	--	--	--	--	--	--	NaN	--
0 11 10	--	--	--	--	--	--	NaN	--
0 11 11	--	--	--	--	--	--	NaN	--

Note the transition between denormalized and normalized
Have to always check for the hidden bit

e: the value represented by considering the exponent field to be an unsigned integer

E: the value of the exponent after biasing = e - bias … 1-bias when e = 0

2^E: numeric weight of the exponent

f: the value of the fraction

M: the value of the significand =1+f ==1.f

2^ExM: the (unreduced) fractional value of the number

V: the reduced fractional value of the number

Decimal: the decimal representation of the number

Denormalized values

- Also called denormal or subnormal numbers
- Values that are very close to zero
- Fill the "underflow" gap around zero
- Any number with magnitude smaller than the smallest normal number (i.e. values == $x*2^y$)
- When the exponent field is all zeros
  - **E = 1-bias**
- Significand M = f without implied leading 1
  - **h = 0 (hidden bit)**
- Two purposes
  - Provide a way to represent numeric value 0
    - -0.0 and +0.0 are considered different in some ways and the same in others
  - Represents numbers that are very close to 0.0
    - Gradual underflow = possible numeric values are spaced evenly near 0.0

Denormal numbers (cont)

- In a normal floating point value there are no leading zeros in the significand, instead leading zeros are moved to the exponent.
- For example:
  - 0.0123 would be written as $1.23 * 10^{-2}$.
- Denormal numbers are numbers where this representation would result in an exponent that is too small (the exponent usually having a limited range). Such numbers are represented using leading zeros in the significand.