Storing Information Digitally

Read: Sections 2.4, 3.1, 3.2

Binary Basics:

- Why binary numbers?
  - In a digital computer, all information is represented using the binary number system.
  - It is easy to build fast electronic circuits that operate on the basis of two states: on and off.

- Number systems:
  - Labeling the columns, representing nine hundred forty-two in base 10 (decimal):

Hundreds	Tens	Units
9	4	2
10^2	10^1	10^0
$9 * 10^2$	$4 * 10^1$	$2 * 10^0$

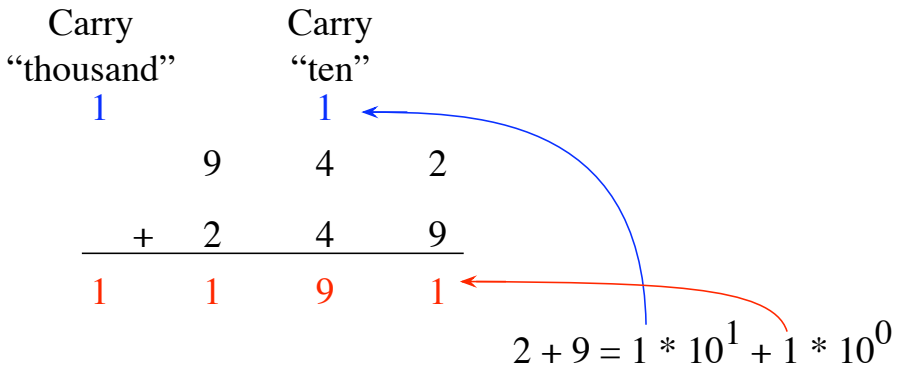
Binary Basics (continued):

- Range of possible values:
  - For a three-digit number, the maximum is:

Hundreds	Tens	Units
9	9	9
10^2	10^1	10^0
$9 * 10^2$	$9 * 10^1$	$9 * 10^0$

  - One less than one thousand (10^3).
  - In general, for an N-digit number, the range is 0 to $10^N - 1$.

- Addition:



Binary Basics (continued):

- Let's consider an 8 digit binary number: 1010 1011
  - Label the columns starting at the right in increasing powers of 2:

1	0	1	0	1	0	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128's	64's	32's	16's	8's	4's	2's	1's

  - To convert to decimal, add the decimal values for each place that has a 1:

    $= 128 + 32 + 8 + 2 + 1 = 171_{ten}$.

  - Range of an N-digit, binary number: $0 .. 2^N - 1$
  - For an 8-digit number, this would be $0000\ 0000_{two}$ to $1111\ 1111_{two}$, or $0 .. 2^8 - 1$, or 0_{ten} to 255_{ten}.
- Terminology:
  - Each *b*inary dig*it* is called a *bit*.
  - Eight binary digits make up a *byte*.
  - Four binary digits thus make a *nibble* (really! :-)
  - The left most bit is called the *most significant bit*, or *MSB*.
  - The right most bit is called the *least significant bit*, or *LSB*.

Binary Addition:

		1	1	1	1	1				
	0	0	1	0	1	0	1	0	$= 1*2^5 + 1*2^3 + 1*2^1$	$= 42_{ten}$
+	0	0	0	1	1	1	1	1	$= 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$	$= 31_{ten}$
	0	1	0	0	1	0	0	1	$= 1*2^6 + 1*2^3 + 1*2^0$	$= 73_{ten}$

$1+1+1 = 1*2^1 + 1*2^0$

- Binary Addition, again:

			1	1	1	1	1			
		0	0	0	0	1	0	1	1	$= 1*2^3 + 1*2^1 + 1*2^0$	$= 11_{ten}$
+	0	0	0	1	0	1	0	1	$= 1*2^4 + 1*2^2 + 1*2^0$	$= 21_{ten}$
	0	0	1	0	0	0	0	0	$= 1*2^5$	$= 32_{ten}$

Overflow:

- Computing devices can only represent binary numbers up to a fixed number of bits.
  - Machines are finite.
- This introduces the problem of *overflow*.
- Binary Addition — if we only have 8 bits (1 byte) to hold each number:

$$\begin{array}{rcl} (1)\ \ 1\ 1\ 1 & & \\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0 & = 1*2^7 + 1*2^3 + 1*2^2 + 1*2^1 & = 142_{ten} \\ +\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1 & = 1*2^7 + 1*2^1 + 1*2^0 & = 131_{ten} \\ \hline 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1 & = 1*2^4 + 1*2^0 & = 17_{ten} \end{array}$$

- We can "solve" the problem by using more than 8 bits.
  - Most systems use (at least) 32 bits, 4 bytes, to represent an integer.
    - The MIPS architecture that we will be using does this.
  - Can still have an overflow problem:

```
  1                                  1
    10000000 10000000 00000000 00000010  = 1*2^31 + 1*2^23 + 1*2^1          = 2,155,872,258
+   10000000 00000000 10000000 00000011  = 1*2^31 + 1*2^15 + 1*2^1 + 1*2^0  = 2,147,516,419
    00000000 10000000 10000000 00000101  = 1*2^23 + 1*2^15 + 1*2^2 + 1*2^0  =     8,421,381
```

Signed and Unsigned Numbers:

- Sometimes we only want to represent positive numbers and zero.
  - Best example: Addresses located in the computer's memory.

- Other times, we need to represent both positive and negative numbers.

- The basic idea: Use one bit to represent the sign.
  - Want to represent a range of positive numbers, and a range of negative numbers, and zero.

- Three methods have been used for representing negative numbers:
  - Sign and Magnitude.
  - One's Complement.
  - Two's Complement.

- None of these is "perfect".

Signed and Unsigned Numbers (continued):

- Sign and Magnitude:
  - Use the left most bit to signal whether the number is positive or negative.
  - Left most bit is then the *sign bit*.
  - Range: $-(2^{N-1} - 1)$ .. $+(2^{N-1} - 1)$
  - Problems:
    - Need separate electronic circuits for manipulating the sign bits.
    - Two distinct representations of zero.
      - Causes problems with comparing a number to zero.
        - Results in two comparisons.
        - Comparison to zero happens a lot!!

8-bit, Sign and Magnitude Values

Positive Numbers	Negative numbers
0000 0000 = 0_{ten}	1000 0000 = -0_{ten}
0000 0001 = 1_{ten}	1000 0001 = -1_{ten}
0000 0010 = 2_{ten}	1000 0010 = -2_{ten}
0000 0011 = 3_{ten}	1000 0011 = -3_{ten}
0000 0100 = 4_{ten}	1000 0100 = -4_{ten}
0000 0101 = 5_{ten}	1000 0101 = -5_{ten}
0000 0110 = 6_{ten}	1000 0110 = -6_{ten}
0000 0111 = 7_{ten}	1000 0111 = -7_{ten}
0000 1000 = 8_{ten}	1000 1000 = -8_{ten}
0000 1001 = 9_{ten}	1000 1001 = -9_{ten}
...	...
0111 1111 = 127_{ten}	1111 1111 = -127_{ten}

```
  0111 1111 = 127
+          1
  1000 0000 = ???
```

Signed and Unsigned Numbers (continued):

- One's Complement:
  - Any pattern whose sign bit is 1 represents a negative value.
  - Change (complement) every 1 to 0 and every 0 to 1.
  - Range: $-(2^{N-1} - 1)$ .. $+(2^{N-1} - 1)$
    - Same as Sign and Magnitude.
  - Advantage over Sign and Magnitude:
    - Easier to design circuits which perform arithmetic on numbers represented in this form.
    - Especially: easy to convert a positive number to a negative number (and vice versa).
  - Problem:
    - Two distinct representations of zero (still!).

8-bit, One's Complement Values

Positive Numbers	Negative numbers
$00000000 = 0_{ten}$	$11111111 = -0_{ten}$
$00000001 = 1_{ten}$	$11111110 = -1_{ten}$
$00000010 = 2_{ten}$	$11111101 = -2_{ten}$
$00000011 = 3_{ten}$	$11111100 = -3_{ten}$
$00000100 = 4_{ten}$	$11111011 = -4_{ten}$
$00000101 = 5_{ten}$	$11111010 = -5_{ten}$
$00000110 = 6_{ten}$	$11111001 = -6_{ten}$
$00000111 = 7_{ten}$	$11111000 = -7_{ten}$
$00001000 = 8_{ten}$	$11110111 = -8_{ten}$
$00001001 = 9_{ten}$	$11110110 = -9_{ten}$
...	...
$01111111 = 127_{ten}$	$10000000 = -127_{ten}$

```
  0111 1111 = 127
+          1
-----------------
  1000 0000 = ???

  1111 1111 = -0
+          1
-----------------
  0000 0000 = ???
```

Signed and Unsigned Numbers (continued):

- Two's Complement:
  - To switch between the positive and negative, there are two steps:
    - Invert: each 1 to a 0 and each 0 to a 1,
    - Then, add 1 to get the final value.
  - E.g.,

```
1 1 1 1 1 0 1 1 = -5ten
0 0 0 0 0 1 0 0   Invert step
          + 1     Add 1 step
0 0 0 0 0 1 0 1 = 5ten
```

  - E.g.,

```
0 0 0 0 1 0 0 0 = 8ten
1 1 1 1 0 1 1 1   Invert step
          + 1     Add 1 step
1 1 1 1 1 0 0 0 = -8ten
```

8-bit, Two's Complement Values

Positive Numbers	Negative numbers
$00000000 = 0_{ten}$	$00000000 = -0_{ten}$
$00000001 = 1_{ten}$	$11111111 = -1_{ten}$
$00000010 = 2_{ten}$	$11111110 = -2_{ten}$
$00000011 = 3_{ten}$	$11111101 = -3_{ten}$
$00000100 = 4_{ten}$	$11111100 = -4_{ten}$
$00000101 = 5_{ten}$	$11111011 = -5_{ten}$
$00000110 = 6_{ten}$	$11111010 = -6_{ten}$
$00000111 = 7_{ten}$	$11111001 = -7_{ten}$
$00001000 = 8_{ten}$	$11111000 = -8_{ten}$
$00001001 = 9_{ten}$	$11110111 = -9_{ten}$
...	...
$01111111 = 127_{ten}$	$10000000 = -128_{ten}$

- Same as One's Complement, but you add an extra one (hence, *Two's* Complement).

Signed and Unsigned Numbers (continued):

- Only one zero. Apply Two's Complement to zero (16-bit version shown):

```
0  0  0  0  0  0  0  0      0  0  0  0  0  0  0  0
1  1  1  1  1  1  1  1      1  1  1  1  1  1  1  1    Invert step
                                          +     1    Add 1 step
-------------------------------------------------
0  0  0  0  0  0  0  0      0  0  0  0  0  0  0  0
```

  - We "lost" the 1 that carried out past the left side in that addition (did you notice?).

- Solves the problem of two representations of zero.

- However, the negative value 10000000 00000000 does not have an equivalent positive representation:

```
1  0  0  0  0  0  0  0      0  0  0  0  0  0  0  0
0  1  1  1  1  1  1  1      1  1  1  1  1  1  1  1    Invert step
                                          +     1    Add 1 step
-------------------------------------------------
1  0  0  0  0  0  0  0      0  0  0  0  0  0  0  0
```

- Range is therefore: $-(2^{N-1})$ .. $+(2^{N-1} - 1)$
  - For 8-bit values: -128_{ten} .. $+127_{ten}$
  - For 16-bit values: $-32{,}768_{ten}$ .. $+32{,}767_{ten}$
  - For 32-bit values: $-2{,}147{,}483{,}648_{ten}$ .. $+2{,}147{,}483{,}647_{ten}$

Sign Extension:

- Often you will have a number that needs to be padded out to a larger number of bits. For example, an 8-bit value to place in a 16-bit storage location; or, a 16-bit value in a 32-bit location; etc.
- Put the number in the least significant bits of the location and pad to the left by replicating the sign bit.

- For example, starting with an 8-bit representation of -6_{ten}:

```
8-bits:                                                 1 1 1 1 1 0 1 0
16-bits:                                        1 1 1 1 1 1 1 1   1 1 1 1 1 0 1 0
32-bits:   1 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1   1 1 1 1 1 0 1 0
```

- Analogously for positive numbers: extend the zero on the left.
- For example, starting with a 16-bit representation of 312_{ten}:

```
16-bits:                                        0 0 0 0 0 0 0 1   0 0 1 1 1 0 0 0
32-bits:   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 1   0 0 1 1 1 0 0 0
```

Binary Arithmetic:

- Addition:

```
      1 1 1 1 1
    0 0 0 1 1 1 1 1 =        31ten
  + 1 1 0 1 0 1 1 1 =    + -41ten
    1 1 1 1 0 1 1 0 =       -10ten
```

- Subtraction:

```
                    10
                  0 0 10
    1   0   0   0   1 1 1 0  =     -114ten
-   1   0   0   0   0 0 1 1  =   - -125ten
    0   0   0   0   1 0 1 1  =      11ten
```

Binary Arithmetic (continued):

- Easier approach: Change the subtrahend by finding its Two's Complement, then Add the two values:

```
   1  0  0  0  1  1  1  0  =     -114ten
-  1  0  0  0  0  0  1  1  =   - -125ten
```

Find the two's complement of the subtrahend:

```
   1  0  0  0  0  0  1  1
   0  1  1  1  1  1  0  0   Invert step
+                     1    Add 1 step
   0  1  1  1  1  1  0  1   Two's complement of subtrahend
```

Now, add the two numbers:

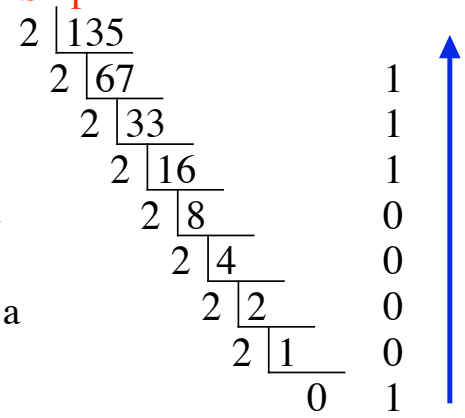
```
   1  1  1  1  1  1
   1  0  0  0  1  1  1  0
+  0  1  1  1  1  1  0  1
   0  0  0  0  1  0  1  1
```

Ignore the carry out

Converting Decimal to Binary:

- Successive Division:
  - Keep dividing by two.
  - The remainders form the binary result.
  - Note the order in which the remainders are collected!
  - (16-bit example, since 135_{ten} will not fit in a signed 8-bit number):

Step 1:

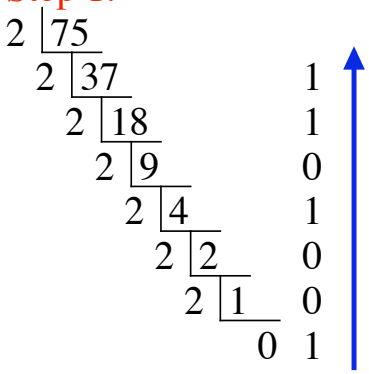


Step 2:
= 1 0 0 0 0 1 1 1

Step 3:
pad with 0's to 16-bits
sign-bit for positive number:
0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1

- For a negative number, find its positive representation, then find its Two's Complement:

Step 1:



Step 2:
= 1 0 0 1 0 1 1

Step 3:
pad to 16-bits with zero
sign-bit for positive number:
0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1

Step 4:
Two's complement to get negative number:

1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0
+ 1
1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 1

Converting Decimal to Binary (continued):

- Successive Subtraction: Subtract powers of two (16-bit example):
  - Convert 135_{ten} to binary:
  - For 16-bit, signed value, we start with $2^{14} = 16{,}384$:

135	we can't	0	subtract 16,384
135	we can't	0	subtract 8,192
135	we can't	0	subtract 4,096
135	we can't	0	subtract 2,048
135	we can't	0	subtract 1,024
135	we can't	0	subtract 512
135	we can't	0	subtract 256
135	we can	1	subtract 128 (remainder 7)
7	we can't	0	subtract 64
7	we can't	0	subtract 32
7	we can't	0	subtract 16
7	we can't	0	subtract 8
7	we can	1	subtract 4 (remainder 3)
3	we can	1	subtract 2 (remainder 1)
1	we can	1	subtract 1 (remainder 0)

$= 0000\ 0000\ 1000\ 0111$

Multiplication in Binary:

- 4-bit example:

```
          1  0  0  0
        * 0  1  1  0
        ------------
          0  0  0  0
       1  0  0  0
    1  0  0  0
    ------------------
0   0  0  0  0
0  1  1  0  0  0  0
```

- Note: we ended up with more than a 4-bit answer.
  - In general, to multiply two N-bit binary numbers, you need to allow for up to 2 * N bits in the answer. For example:

```
             1  1  0  0
          * 1  1  0  1
          ------------
             1  1  0  0
          0  0  0  0
       1  1  0  0
    1  1  0  0
    ------------------
1  0  0  1  1  1  0  0
```

Multiplication in Binary (continued):

- Multiplying by two is a "shift left" operation:

```
      0  0  1  1  0  0  1  1
*                     1  0
-------------------------------
      0  0  0  0  0  0  0  0
   0  0  1  1  0  0  1  1
-------------------------------
   0  0  1  1  0  0  1  1  0
```

- Works with both signed and unsigned integers.
  - Unsigned:

$$0\ 0\ 1\ 0\ 1\ 1\ 0\ 1 = 2^5 + 2^3 + 2^2 + 2^0 = 32 + 8 + 4 + 1 = 45_{ten}$$

$$0\ 1\ 0\ 1\ 1\ 0\ 1\ 0 = 2^6 + 2^4 + 2^3 + 2^1 = 64 + 16 + 8 + 2 = 90_{ten}$$

$$1\ 0\ 1\ 1\ 0\ 1\ 0\ 0 = 2^7 + 2^5 + 2^4 + 2^2 = 128 + 32 + 16 + 4 = 180_{ten}$$

    - What happens when we shift left again?
  - Signed:

$$1\ 1\ 1\ 0\ 0\ 0\ 0\ 0 = -32_{ten}$$

$$1\ 1\ 0\ 0\ 0\ 0\ 0\ 0 = -64_{ten}$$

$$1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 = ??$$

    - If we shift left again?

Division in Binary:

- $85_{ten} / 6_{ten} = 14\ 1/6$

```
                1 1 1 0   quotient
          ---------------
1 1 0 ) 1 0 1 0 1 0 1
        - 1 1 0
          1 0 0 1
        - 1 1 0
            1 1 0
          - 1 1 0
                1   remainder
```

$1110 = 2^3 + 2^2 + 2^1$
$= 8 + 4 + 2$
$= 14_{ten}$

- Dividing by two is a "shift right" operation:

```
              1 1 0 0 1 quotient
          -------------
1 0 ) 0 0 1 1 0 0 1 1
      - 1 0
          1 0
        - 1 0
            0 0 1 1
          - 1 0
              1
```

Division in Binary (continued):

- Signed number:
  - How does shift right maintain the correct result for dividing by 2?
  - Arithmetic shift right: replicate the sign bit when shifting:
    - $\mathbf{-31_{ten} = 1110\ 0001_{two}}$
    - Shift right: $\mathbf{1111\ 0000_{two}}$

      0000 1111 Invert step

      + 1 Add one

      0001 0000 = 2^4 = 16_{ten}
    - -16_{ten} is the correct result.

Signed Numbers — Detecting Overflow:

- No overflow possible when adding a positive and a negative number: positive + negative negative + positive
- No overflow possible when the signs are the same for subtraction: positive - positive negative - negative
- Overflow occurs when the answer makes the sign wrong:
  - Examples using 8-bit, Two's Complement integers:
    - Overflow: adding two positives yields a "negative" (answer should be positive)

```
    1         1  1
    0  1  1   0  1  1  1  0        110ten
+   0  1  0   0  0  1  0  0    +    68ten
-------------------------------------------
    1  0  1   1  0  0  1  0        -78ten
```

    - Overflow: adding two negatives gives a "positive" (answer should be negative)

```
 1
    1  0  0   1  0  0  1  1       -109ten
+   1  1  0   0  1  1  0  0   + -   52ten
-------------------------------------------
    0  1  0   1  1  1  1  1         95ten
```

Note: With signed numbers, whether there is, or is not, a carry-out on the left tells you *nothing* about whether overflow did or did not occur!

Signed Numbers — Detecting Overflow (continued):

- Overflow: subtract a negative from a positive and get a negative (should be positive).

```
   0  1  1  1  1  1  1  1          127ten
-  1  1  1  1  1  1  1  1      -    -1ten
   1  0  0  0  0  0  0  0  -> Two's complement for -128, but answer should be +128.
```

- Overflow: subtract a positive from a negative and get a positive (should be negative).

```
   1  0  0  0  0  0  0  1        -127ten
-  0  0  0  0  0  1  0  1      -   +5ten
   0  1  1  1  1  1  0  0  -> Two's complement for +124ten, but answer should be -132ten
```

- Figure 3.2, page 226:

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Signed Numbers — Detecting Overflow (continued):

- Okay, so do we really care?
  - Sometimes, we do not!
    - Ignore any overflows that occur.
    - Java example:

```
public class Zap {
   public static void main (String args[]) {
      int i, j;
      for (i = 2147483640, j = 0; j < 15; i++, j++)
         System.out.println("i = " + i + "    j = " + j);
    } // main
} // class Zap
```

- Sometimes, we do
  - Duplicate versions of assembly language instructions that cause an "exception" when overflow occurs.
  - MIPS uses this approach:
    - **`add, addi, sub`** cause exceptions on overflow.
    - **`addu, addiu, subu`** do not cause exceptions on overflow.

Hexadecimal and Octal:

- Binary can be difficult to read and write (for us humans :-).
- An alternative is to use octal notation.
  - Octal = base 8.
  - 3 binary digits = 1 octal digit.

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

  - Counting exercise: What ten numbers follow:
    - 75, 76,
    - 376,

Hexadecimal and Octal (continued):

- Converting octal to/from binary:
  - Group the binary number into groups of three digits, starting from the right.
    - Examples:

$00\ \ 110\ \ 111_{two} = 067_{octal}$

$10\ \ 101\ \ 001_{two} = 251$

$11\ \ 111\ \ 111_{two} = 377$

$01\ 110\ 100\ 010\ 101\ 011\ 111\ 010\ 010\ 101\ 100_{two} =$

$1\ \ 6\ \ \ 4\ \ \ 2\ \ \ 5\ \ \ 3\ \ \ 7\ \ \ 2\ \ \ 2\ \ \ 5\ \ \ 4$

- Converting octal to/from decimal:
  - My preference: convert to binary first:

$117_{octal} = 001\ 001\ 111 = 64 + 8 + 4 + 2 + 1 = 79_{ten}$

  - But, you can do it directly:

$2{,}143_{octal} = 010\ 001\ 100\ 011 = 2\text{^}10 + 2\text{^}6 + 2\text{^}5 + 2\text{^}1 + 2\text{^}0 = 1024 + 64 + 32 + 2 + 1 = 1123_{ten}$

$2{,}143_{octal} = 2 * 8\text{^}3 + 1 * 8\text{^}2 + 4 * 8\text{^}1 + 3 * 8\text{^}0 = 2 * 512 + 64 + 32 + 3 = 1{,}123_{ten}$

Hexadecimal and Octal (continued):

- Converting octal to/from decimal (continued):
  - Convert $1{,}293_{ten}$ to octal

My preference:
Convert to binary, then to octal:

2 | 1293		
2 | 646	1	
2 | 323	0	
2 | 161	1	
2 | 80	1	
2 | 40	0	
2 | 20	0	
2 | 10	0	
2 | 5	0	
2 | 2	1	
2 | 1	0	
0	1	

$= 10100001101_{two}$

$= 10\ 100\ 001\ 101_{two}$

$= 2\quad 4\quad 1\quad 5_{octal}$

Can be done directly
using successive division:

8 | 1293	
8 | 161	5
8 | 20	1
8 | 2	4
0	2

$= 2{,}415_{octal}$

Conversion Exercises:

1. Convert $3{,}724_{octal}$ to decimal.
2. Convert $1{,}562_{ten}$ to octal.

Hexadecimal and Octal (continued):

- More common today: base 16 (hexadecimal):

Binary	Hexadecimal	Binary	Hexadecimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

- Counting exercise: What ten hexadecimal numbers follow:
  - 97, 98, 99, 9A, 9B, 9C, …, 9F, A0, A1, A2, …, FD, FE, FF, 100
  - 2F9, 2FA, 2FB, 2FC, 2FD, 2FE, 2FF, 300,
  - largest hexadecimal value that will fit in an unsigned, 8-bit byte: FF

Hexadecimal and Octal (continued):

- Convert hexadecimal to/from binary:
- Group the binary number in groups of four digits, starting from the right.
- Examples:

1001 = 8 + 1; 1111 = 8 + 4 + 2 + 1

0100 $1001_{two} = 49_{hex}$

1111 $1111_{two} = FF_{hex}$

0110 $1100_{two} = ??_{hex}$

$1001\ 1000\ 1101\ 1011\ 1100\ 0100\ 0101\ 0011_{two}$ =

$2F\ 37\ A0\ C6_{hex}$ =

2 F 3 7 A 0 C 6 =

Hexadecimal and Octal (continued):

- Converting hexadecimal to/from decimal:
  - My preference: convert to binary first:

$$AC8E_{hex} = 1010\ 1100\ 1000\ 1110_{two} = 2^{15} + 2^{13} + 2^{11} + 2^{10} + 2^{7} + 2^{3} + 2^{2} + 2^{1} =$$

32,768	2^{15}
8,192	2^{13}
2,048	2^{11}
1,024	2^{10}
128	2^{7}
8	2^{3}
4	2^{2}
+ 2	2^{1}
$44,174_{ten}$	

  - But, you can do it directly:

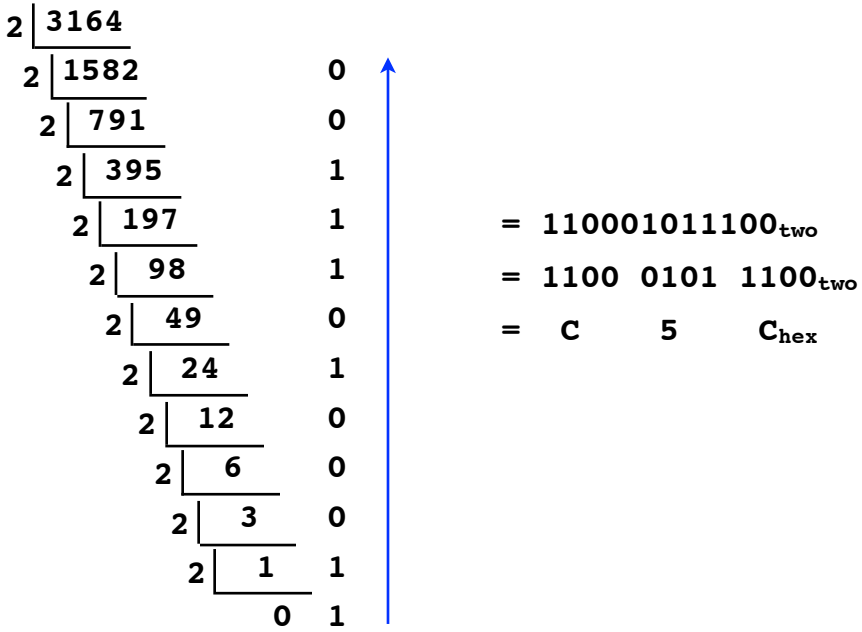
$$AC8E_{hex} = 10 * 16^{3} + 12 * 16^{2} + 8 * 16^{1} + 14 * 16^{0} =$$

40,960	10 * 4,096
3,072	12 * 256
128	8 * 16
+ 14	14 * 1
$44,174_{ten}$	

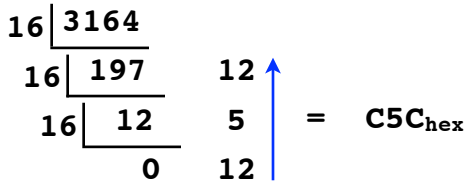
Hexadecimal and Octal (continued):

- Converting hexadecimal to/from decimal (continued):
  - Convert $3{,}164_{ten}$ to hexadecimal.

My preference:
Convert to binary, then to hexadecimal:



Can be done directly
using successive division:



Conversion Exercises:

1. Convert 5B $C7_{hex}$ to decimal.
2. Convert $9{,}127_{ten}$ to hexadecimal.

Hexadecimal and Octal (continued):

- Multiplication table for hexadecimal:
  - Complete the rows for **4** and **8** (they are useful) and **F** (for fun :-)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2														
3	0	3														
4	0	4														
5	0	5														
6	0	6														
7	0	7														
8	0	8														
9	0	9														
A	0	A														
B	0	B														
C	0	C														
D	0	D														
E	0	E														
F	0	F														

Floating Point (a brief look).

- Read: Section 3.5 — pages 196-222 (5th ed.), 242-254 (4th ed.)
- Need a way to represent:
  - Numbers with fractions: 3.1415926536
  - Very small numbers: 0.000000000000132 = $1.32 * 10^{-13}$
  - Very large numbers: $6.023 * 10^{23}$
  - Negative versions of these as well.
- Review:
  - $6.023 * 10^{23}$
  - 6.023 is the *mantissa*.
    - .023 is the *fraction*.
  - 10 is the *base* (or *radix*).
  - 23 is the *exponent*.
- In binary, it is exactly the same, except the base is *two*, not ten
- Computer arithmetic that supports this form is called *floating point* arithmetic.

Floating Point Representation:

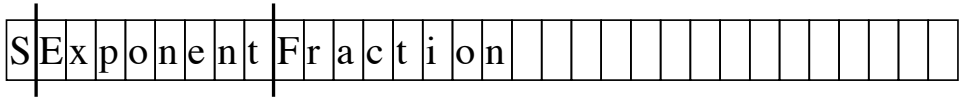
- Floating point numbers are stored in binary, not decimal!
- Example: $176{,}128.75_{ten}$
  - $176{,}128_{ten} = 131{,}072 + 32{,}768 + 8{,}192 + 4{,}096 = 2^{17} + 2^{15} + 2^{13} + 2^{12} = 10\ 1011\ 0000\ 0000\ 0000_{two}$
  - $0.75_{ten} = ½ + ¼ = 2^{-1} + 2^{-2} = 0.11_{two}$
  - $176{,}128.75_{ten} = 10\ 1011\ 0000\ 0000\ 0000.11_{two}$
- Normal format is $1.m * 2^e$
  - $176{,}128.75_{ten} = 10\ 1011\ 0000\ 0000\ 0000.11_{two} = 1.0\ 1011\ 0000\ 0000\ 0000\ 11 * 2^{17}$
  - $= 1.0\ 1011\ 0000\ 0000\ 0000\ 11 * 2^{10001}$
- If we have a fixed amount of space (and we always do!), we have to trade off the number of bits among *m* and *e*:
  - More bits for *m* = greater accuracy.
  - More bits for *e* = greater range.
  - Historically, there have been many different representations
    - They differ (mainly) in the values of *m* and *e*.

Floating Point Representation (continued):

- IEEE 754 — A Standard for Floating Point Numbers.
  - Bits for $m = 23$. Bits for $e = 8$.
  - Plus one bit for the Sign.
- Exponent.
  - With 8 bits for the exponent we can represent numbers from approximately 2^{-127} .. 2^{127}
    - $2^{-127} \approx 10^{-38}$; $2^{127} \approx 10^{38}$.
- Mantissa.
  - Since the first digit of the Mantissa is always 1 in binary (what is it in decimal?), we can optimize (save space) by not storing it (but remembering it is there).

Floating Point Representation (continued):

- Laying out Floating Point Numbers.
  - Basic layout:



    - S = 1 bit, Exponent = 8 bits, Fraction = 23 bits.
  - There is a strong motivation to enable comparisons to be made using integer compares:
    - Compare the signs.
    - Compare the exponents.
    - Compare the fractions.
  - Problem with comparing exponents: -127 appears to be larger than +127 (when both are written as Two's Complement binary numbers).
  - Solution: *bias* the exponent so positive exponents occupy the upper-half of the exponent range.
  - In IEEE 754, 127_{ten} is used to bias the exponent.
    - For an exponent of $+1_{ten}$ use $1 + 127 = 128_{ten} = 1000\ 0000_{two}$
    - For an exponent of $+12_{ten}$ use $12 + 127 = 139_{ten} = 1000\ 1011_{two}$
    - For -4_{ten}: $-4 + 127 = 123_{ten} = 0111\ 1011_{two}$
    - For -33_{ten}: $-33 + 127 = 94_{ten} = 0101\ 1110_{two}$

Floating Point Representation (continued):

- Completing the example:
  - $176{,}128.75_{ten} = 1.0\ 1011\ 0000\ 0000\ 0000\ 11 * 2^{17}$
  - Sign: 0 (positive)
  - Exponent:
    - 17
    - Add the bias: 17 + 127 = 144 = 128 + 16 = $1001\ 0000_{two}$
    - Suppose the exponent was 12 + 127 = 139 = $1000\ 1011_{two}$
    - Suppose the exponent was -17: -17 + 127 = 110_{ten} = 0110 1110
  - Fraction (without the 1 to the left of the decimal): 010 1100 0000 0000 0011 0000
  - Result:

```
S Expo nent Fra ctio n
0 1001 0000 010 1100 0000 0000 0011 0000
```

- When converting **from decimal to binary**:
  - **Add** the bias (always).
  - Does not matter whether the exponent in decimal is positive or negative, always add.

Floating Point Representation (continued):

- Example of a Floating Point number:

```
0  01101000  101 0000 0000 0000 0000 0000
S  Exponent  Fra ctio n
```

  - Sign bit is **0**, so the number is positive.
  - Exponent: 8 digits
    - **0110 1000**$_{two}$ = 104_{ten}
    - Biased: 104_{ten} - 127_{ten} = -23_{ten}
  - Fraction: 23 digits
    - **101 0000 0000 0000 0000 0000**
    - 1.101_{two} * 2^{-23} .0000 0000 0000 0000 0000 0011 01
    - 1.101_{two} = 1 + (1*2^{-1} + 0*2^{-2} + 1*2^{-3}) = 1 + ½ + ⅛ = 1 + ⅝ = 1.625_{ten}
  - Number: 1.625 * 2^{-23}, which is approximately 1.937 * 10^{-7}.

- When converting **from binary to decimal**:
  - **Subtract** the bias (always).
  - Does not matter whether the exponent in binary starts with **1** or **0**, always subtract.

Floating Point Representation (continued):

- There are three problems to handle:
  - Zero: The standard uses all `0`'s for the exponent and all `0`'s for the fraction:

    ```
    0 0000 0000 000 0000 0000 0000 0000 0000
    S Expo nent Fra ctio n
    ```

    - The sign bit can be `0` or `1`.
  - Infinity: The standard uses all `1`'s for the exponent and all `0`'s for the fraction:

    ```
    0 1111 1111 000 0000 0000 0000 0000 0000
    S Expo nent Fra ctio n
    ```

    - The sign bit can be `0` or `1`. Supports positive and negative infinity.
  - NaN: Not a Number. For operations such as `0 / 0` or subtracting infinity - infinity.
    - The standard uses all `1`'s for the exponent and some (any) nonzero value for the fraction.

      `0 1111 1111 101 0000 0000 0000 0000 0000` (as one example).

      ```
      S Expo nent Fra ctio n
      ```

    - The sign bit can be `0` or `1`.

Floating Point Representation (continued):

- Exponent:
  - -126_{ten} to $+127_{ten}$ can be represented.
    - $-126 + 127 = 1_{ten} = 0000\ 0001_{two}$
    - $+127 + 127 = 254_{ten} = 1111\ 1110_{two}$
  - When the exponent is all **0**'s:
    - If the fraction is all **0**'s, the bit pattern represents $\mathbf{0_{ten}}$.
    - If the fraction is nonzero, the fraction does not have an implied leading **1**.
      - Called "denormalized" values.
      - Allows representation of values closer to zero.
- Figure 3.13, page 199 (partial), Single Precision:

Exponent	Fraction	Object represented
0	0	0
0	nonzero	+/- denormalized number
1 - 254	anything	+/- floating-point number
255	0	+/- infinity
255	nonzero	NaN (Not a Number)

Exponent: 2^{-127} to 2^{+127}

$1.1 * 2^{-127} =$

-127 + 127 = 0000 0000

0 0000 0000 100 0000 0000 0000 0000 0000

$1.101 * 2^{-143} =$

$0.0000\ 0000\ 0000\ 0001\ 101 * 2^{-127} =$

0 0000 0000 0000 0000 0000 0001 1010 000

$1.101 * 2^{-137} =$

$0.0000\ 0000\ 0110\ 1 * 2^{-127} =$

0 0000 0000 0000 0000 0110 1000 0000 000

$1.101 * 2^{-148} =$

$0.\ 0000\ 0000\ 0000\ 0000\ 0000\ 01101 * 2^{-127} =$

0 0000 0000 0000 0000 0000 0000 0000 011~~01~~

$1.101 * 2^{-150} =$

$0.\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 101 * 2^{-127} =$

0 0000 0000 0000 0000 0000 0000 0000 000~~1101~~

Floating Point Representation (continued):

- Example:
  - Decimal: -0.75 = - 3/4 = - (1/2 + 1/4) = - ($1*2^{-1} + 1*2^{-2}$)
  - Binary: -0.11 = $-1.1 * 2^{-1}$
  - Floating point:
    - Sign bit for negative value: 1
    - Exponent:
      - -1_{ten} + bias = $-1_{ten} + 127_{ten} = 126_{ten} = 0111\ 1110_{two}$
    - Fraction: 1.1
      - 1.100 0000 0000 0000 0000 0000
      - Implicit 1 at the start not part of the stored number.
      - Fraction is thus: 100 0000 0000 0000 0000 0000
  - IEEE single precision result: 1 0111 1110 100 0000 0000 0000 0000 0000

```
1  01111110  10000000000000000000000
S  Exponent  Fraction
```

Floating Point Representation (continued):

- Example:
  - $4.2 * 10^3$ into binary (IEEE format).
  - $4.2 * 10^3 = 4200_{ten}$
  - $4200_{ten} = 1\ 0000\ 0110\ 1000_{two}$
  - "Normalize it"
    - Need to move the binary decimal point 12 places:
      - $1.000\ 0011\ 0100\ 0 * 2^{12}$
  - Result:
    - Sign bit for positive number: **0**
    - Exponent:
      - $12_{ten} + 127_{ten} = 139_{ten} = 128_{ten} + 8_{ten} + 2_{ten} + 1_{ten} = \mathbf{1000\ 1011}_{two}$
    - Fraction: 1.000001101000, we drop the implicit 1, to get:
      - **000 0011 0100 0000 0000 0000**, extra zero's needed on the right to fill out the 23 digits

```
0   1000 1011   000  0011  0100  0000  0000  0000
S   Expo nent   Fra  ctio  n
```

Floating Point Addition:

- Have to make the exponents the same (align the floating decimal point).
- Base 10 example:
  - $4.2*10^3 + 4.2*10^5 = 0.042*10^5 + 4.2*10^5 = 4.242*10^5$
- Binary example:
  - We converted $4.2 * 10^3$ earlier (see previous slide): **`0  1000 1011  000 0011 0100 0000 0000 0000`**
  - $4.2 * 10^5 = 420{,}000_{ten} = 110\ \ 0110\ \ 1000\ \ 1010\ \ 0000_{two} = 1.1001101000101*2^{18}$
    - Sign, number is positive: **`0`**
    - Exponent: $18_{ten} + 127_{ten} = 145_{ten} =$ **`1001 0001`**$_{two}$
    - Fraction: **`100 1101 0001 0100 0000 0000`**
  - Result:

```
0  1001 0001  100 1101 0001 0100 0000 0000
S  Expo nent  Fra ctio n
```

Floating Point Addition (continued):

- Adding:

```
  0  1001 0001  100 1101 0001 0100 0000 0000    2^18    4.2 * 10^5
+ 0  1000 1011  000 0011 0100 0000 0000 0000    2^12  + 4.2 * 10^3
```

  - Shift the second number six places to the right to align the exponents:

```
  0  1001 0001  100 1101 0001 0100 0000 0000    2^18      4.2   * 10^5
+ 0  1001 0001  000 0010 0000 1101 0000 0000    2^18    + 0.042 * 10^5
```

  - Have to put the implicit one into the second number when we shift it right!
  - Result:

```
  0  1001 0001  100 1101 0001 0100 0000 0000    2^18      4.2   * 10^5
+ 0  1001 0001  000 0010 0000 1101 0000 0000    2^18    + 0.042 * 10^5

  0  1001 0001  100 1111 0010 0001 0000 0000

  S  Exponent   Fra ctio n
```

  - Checking the result to see if it matches (approximates) the decimal result:

Exponent: 1001 0001 = 128 + 16 + 1 = 145_{ten} - 127_{ten} = 18_{ten}

1.100 1111 0010 0001 * 2^{18} = 110 0111 1001 0000 1000_{two} = $67,908_{hex}$ =

$6*16^4 + 7*16^3 + 9*16^2 + 8*16^0$ = 6*65,536 + 7*4,096 + 9*256 + 8 =
393,216 + 28,672 + 2,304 + 8 = 424,200

Floating Point (continued):

- IEEE 754 Floating Point standard:
  - Single precision:
    - Sign bit.
    - Exponent: 8 bits, bias 127.
    - Fraction: 23 bits.
  - Double precision:
    - Sign bit.
    - Exponent: 11 bits, bias 1023.
    - Fraction: 52 bits.

Floating Point Complexities:

- Operations are somewhat more complicated than integer operations (see text).
- In addition to overflow, we can have "underflow":
  - Value too close to zero to be represented.
- Accuracy can be a **big** problem.
  - IEEE 754 uses two extra bits during operations: **`guard`** and **`round`** — to round intermediate answers.
  - Four rounding modes.
  - Positive divided by zero yields "infinity".
  - Zero divided by zero yields "Not a Number".
  - Other complexities…
- Implementing the standard can be tricky.
- MIPS uses separate floating-point registers (not part of the 32 general-purpose registers that are used for integer operations) — see discussion in the text.
  - We will come back to this point later.
- Not using the standard can be even worse.
  - See the text (4th ed: pages 272-279; 5th ed: pages 231-232) for description of the 80x86 floating-pt operations and the Pentium divide bug!

Representation of Characters:

- Characters such as 'A' '?' 'p' 'u' '5' (and the characters in this document :-) must also be represented.
- Standard representation (ASCII code) — 4th edition: Figure 2.15, page 122.

```
0 1 0 0 0 0 0 1  = A
0 1 0 0 0 0 1 0  = B
0 1 1 0 0 0 0 1  = a
0 1 1 0 0 0 1 0  = b
```

- Also have to represent numbers as characters; e.g. '5', but note these are not the same as 5.
  - 5 (the number, as an 8-bit, signed integer) = **0 0 0 0 0 1 0 1**
  - '5' (the character) = **0 0 1 1 0 1 0 1**
  - 55 (the number, as an 8-bit, signed integer) = **0 0 1 1 0 1 1 1**
  - '5' '5' (two characters, need 16-bits, 8 bits for each '5') = **0 0 1 1 0 1 0 1 0 0 1 1 0 1 0 1**
- How many bits would it take to represent $1{,}000{,}000_{ten}$ in ASCII?
- How many bits would it take to represent $1{,}000{,}000_{ten}$ as an unsigned integer?

Representation of Characters (continued):

- ASCII in various bases (taken from "man ascii" in OS X):

```
   The decimal set:
  0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack    7 bel
  8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so    15 si
 16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn   23 etb
 24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs    31 us
 32 sp    33  !    34  "    35  #    36  $    37  %    38  &    39  '
 40  (    41  )    42  *    43  +    44  ,    45  -    46  .    47  /
 48  0    49  1    50  2    51  3    52  4    53  5    54  6    55  7
 56  8    57  9    58  :    59  ;    60  <    61  =    62  >    63  ?
 64  @    65  A    66  B    67  C    68  D    69  E    70  F    71  G
 72  H    73  I    74  J    75  K    76  L    77  M    78  N    79  O
 80  P    81  Q    82  R    83  S    84  T    85  U    86  V    87  W
 88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^    95  _
 96  `    97  a    98  b    99  c   100  d   101  e   102  f   103  g
104  h   105  i   106  j   107  k   108  l   109  m   110  n   111  o
112  p   113  q   114  r   115  s   116  t   117  u   118  v   119  w
120  x   121  y   122  z   123  {   124  |   125  }   126  ~   127 del
```

Representation of Characters (continued):

- ASCII in various bases (continued):

The hexadecimal set:

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0a nl	0b vt	0c np	0d cr	0e so	0f si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2a *	2b +	2c ,	2d -	2e .	2f /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3a :	3b ;	3c <	3d =	3e >	3f ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4a J	4b K	4c L	4d M	4e N	4f O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5a Z	5b [	5c \	5d]	5e ^	5f _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6a j	6b k	6c l	6d m	6e n	6f o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7a z	7b {	7c |	7d }	7e ~	7f del

Representation of Characters (continued):

- ASCII in various bases (continued):

The octal set:

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	ht	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042	"	043	#	044	$	045	%	046	&	047	'
050	(	051	)	052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	`	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	152	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174	|	175	}	176	~	177	del

Representation of Characters (continued):

- ASCII printable and non-printable characters:
- Printable:
  - Characters from the blank space to the tilde character.
    - Decimal: values between 32 and 126, inclusive.
- Non-Printable:
  - Characters from the **`nul`** character (ascii value 0) to the **`us`** character (**`us`** = unit separator, ascii value 31).
  - And, the **`del`** character: (ascii value 127).
- Interesting non-printable characters:
  - Ascii values 1 to 26 are the control characters: control-A through control-Z.
  - Some are tied to keys:
    - Control-I, ascii 9, **`ht`**, "horizontal tab", is the tab key.
    - Control-H, ascii 8, **`bs`**, "backspace", is the backspace key.
    - Ascii 27, "escape", is the **`ESC`** key.
  - Networking use, in low-level message protocols:
    - **`soh`**, "start of heading", **`stx`**, "start of text", **`etx`**, "end of text".

Representation of Characters (continued):

- Interesting non-printable characters (continued):
  - There are two ascii characters that can control advancing to the next line:
    - Control-J, ascii 10, **`lf`**, "line feed".
    - Control-M, ascii 13, **`cr`**, "carriage return".
    - Windows puts both characters at the end of a line. In Java, when you use '**`\n`**', you get both **`cr`** and **`lf`**.
    - Unix puts only the **`lf`** character at the end of a line. In Java, when you use '**`\n`**', you get only **`lf`**.
      - Linux and Mac OS X follow the Unix convention.

Representation of Characters (continued):

- Unicode.

  - Of course, 128 is not very many possibilities when you have to represent all the characters in the world.
    - Think different alphabets here.
    - East Asian, Cyrillic, Arabic, Greek, Tibetan, Cherokee, …

  - Many systems use Unicode. 16 bits for each character.
    - Java specifies the use of Unicode.

  - 2^{16} allows for 65,536 characters.

  - The first 128 characters in Unicode are the ASCII characters, of course. (Why "of course"?)

Summary:

- Numbers and characters may be represented using binary.
  - And other items as well (assembly language instructions, for example).
  - We always have to know how to interpret a bit pattern -- the pattern alone is not sufficient!
- Variety of techniques for representing negative numbers, floating point numbers, etc.
- Can perform arithmetic on binary numbers.
- Can (easily?) convert between more readable hexadecimal (or octal) and binary notation.
- How do we tell from looking at a bit pattern what it represents?
  - Often cannot tell from the pattern alone (see the homework).
  - Can sometimes eliminate some possibilities.
  - Need context to determine what the bit pattern means…