Decimal to Binary Conversion

- Use an algorithm: repeatedly divide by 2 and take the remainder
- Mentally: recognize powers of 2!

Sign Extension:

- What if we have a signed representation and we get more bits? Can we

Bitstrings

- Let's be explicit about how we manipulate strings of bits
- The positions of bits are labeled from 0 to n-1 for an n-bit string (reading from right to left)
- For example, if X = 0101, X[0] = 1; X[1] = 0; X[2] = 1; X[3] = 0
- We can specify ranges (substrings) like so X[2:1]. This is inclusive, X[2:1] = 10
- We have a few more basic operations
- Let's state our goals this way: given an arbitrary bit string, we want to be able to know whether or not bit 6 is 1. We'll need a few operations for that

- **Shifting:** We drop bits on left and right, and fill up the holes on left, right
- Example: 1100 left shifted by one is 1000 (or perhaps 1001, depending on the shifting scheme you invent!)
- These are just bitstrings, but what if we interpret them as numbers? What does shifting mean? (division/multiplication by 2)
- What if it's a negative number? What should our policy be? We want the leading 1s preserved on a right shift correct?
- So we'll just invent a scheme that fills the holes with 1s instead of 0s. This is called arithmetic shift. Zero fill is called logical shift
- Left shifting does not matter whether signed or unsigned, so there's only zero fill
- Shifting in C is done with the "\(>>\)" operator
- C will pick the right one for you based on the type (e.g. int or unsigned int)

- **Bitwise Operators:**
- How do I flip all the bits? Use a bitwise NOT. (in C, the ~ operator, e.g. ~x). For \(x[i] = \sim x[i]\)
- NOTE: this is how you can explicitly take 1s complement!
- Bitwise AND: for all \(x[i]\) in \(x\) and \(y[i]\) in \(y\), \(x \land y = x[i] \& y[i]\). In C, represented with '\&'
- Similarly, we have bitwise OR and bitwise XOR. In C, '\(|\)' and '\(^\sim\)' respectively
**Logical Operators:**
- C also has &&, ||, !, and != (logical and, or, not, xor, respectively)
- in C, integer 0 represent false (0), and any nonzero integer represents true (1).
- x && y is true if x = 27 and y = -32
- ! (x && y) is 0
- !!(x && y) is 1

**Hexadecimal**
- A base 16 format. Not an actual representation, just shorthand for arbitrary bitstrings!
- group bitstrings into 4-byte chunks (called “nibbles”). Each of these chunks takes on one of 16 values
- Assign symbols to each value. We can use 0-9 for the first 10 values. For the rest we use the letters A through F
  - A = 1010, B = 1011, C = 1100, D = 1101, E = 1110, F = 1111
- We usually denote that a number is in hex by prefacing it with “0x” or sometimes just “x”. E.g. 0xdeadbeef or x07F3
- We use it because it’s easier to read than binary.
- Play around with the hexdump program on Linux and you should be able to notice some interesting things about certain files

**Manipulating Bits**
- Back to our question, how do we get bit 6 (whether its 0 or 1?)
- We can use shifts first. Right shift by 6. Bit 6 is now in bit 0’s former position.
- Now all we have to do is bitwise and with 1. If the result is 1, that means bit 6 was set to 1. Otherwise, it wasn’t
- In general, we can use **Bit Masks**, strings of bits used to help isolate parts of bitstrings
- We’ll rely on some properties: (assume b is a bit)

<table>
<thead>
<tr>
<th>X</th>
<th>b ⊕ X</th>
<th>b v X</th>
<th>b ⊙ X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>1</td>
<td>~b</td>
</tr>
</tbody>
</table>

- OK, so given a bit string X, and we want to inspect bit k (X[k]), how do we do it?
- We saw one technique, right shift by k, & with 1
- Another technique, build a mask. The mask has all 0s except in position k. How do we construct such a mask?
- Well, bit k is just 2^k. So we can say our mask M = 2^k. We can then use X & M = S. If S is 0 then bit k was 0, otherwise bit k was 1
- How do we get 2k? In C, we can just left shift 1 by k, e.g. (1 << k)
- How do we set a bit to 1?
- We use the same technique to get an isolated 1 in position k \((1 << k)\)
- Instead of using AND, we can use OR. \(S = X \mid M\) where M again is constructed with \((1 << k)\)
- How do we clear a bit? (AND with \(\sim(1<<k)\))
- How do we flip a bit? (XOR with \((1<<k)\))

- What about for multiple bits?
- Here's where the mask really comes in handy. We need to construct a mask with k bits set to 1 (we don't care about where they are now)
- Let's just assume they'll be in bit positions 0 through k-1
- So, for 8 bits and k=4, we'd like to have 00001111
- We really only know how to set the fourth bit though! \((1 << k)\) or \((1 << 4)\). Could we somehow get our string from that?
- Do the algebra! \(2^k + x = 00001111\) or \(00010000 + x = 00001111\)
- Turns out x here is -1. To build a full mask in bits 0 through k-1, we can just use \(2^k - 1\) or \((1<<k) - 1\)
- Say we want to flip the top four bits of some bit string. Can we use shifts? (yes, with XOR)
- In general: \(X \& M\) gives us the last k bits
- \(X <- X \mid M\) sets the last k bits to 1
- \(X <- X \& \sim M\) clears the last k bits to 0
- \(X <- X \^ M\) flips the last k bits

**NOTE**
: I will post some example C code for bit shifts and bitwise operations and masks

**Multi-bit Masking**

- Please go through the program I posted on fourier or come see me in office hours. I won't have time to cover this again in lecture

**Binary Fractions**

- Binary Fractions: Note decimal notation for non-whole numbers
- 3.75 is \(3*10^0 + 7*10^{-1} + 5*10^{-2}\)
- Dividing by \(10^n\) shifts the decimal point n digits to the left
- What about binary? Same thing, but with powers of 2
- 1.0111 is \(1*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3} + 1*2^{-4} = 1 + 0 + 1/4 + 1/8 + 1/16 = 1\)
  
  
- 7/16 or 21/16 or 1.3125
- How do we convert from decimal?
- We split the number into two parts (left of point and right of point)
- we know how to do left of point already
- For right of point, we subtract out negative powers of 2 repeatedly
E.g., for 1.375, first can we subtract out 1/2? .0
No, so we subtract 1/4. 1.125 ==> .01
Then we subtract 1/8 ==> .011
If it’s in fractional form (a rational number), there is a special case
If the denominator is a power of 2 \( \left(2^n\right)\) we can first write the number as an improper fraction, write the numerator in binary, then shift the (implicit) decimal point left by \( n \)
For example, take 7 5/16. We see that 16 is a power of 2 \( \left(2^4\right)\). We then convert to an improper fraction \((117/16)\).
Then, write 117 in binary and shift the implicit decimal left by 4
1110101 => 111.0101 = 7.3125 = 7 5/16