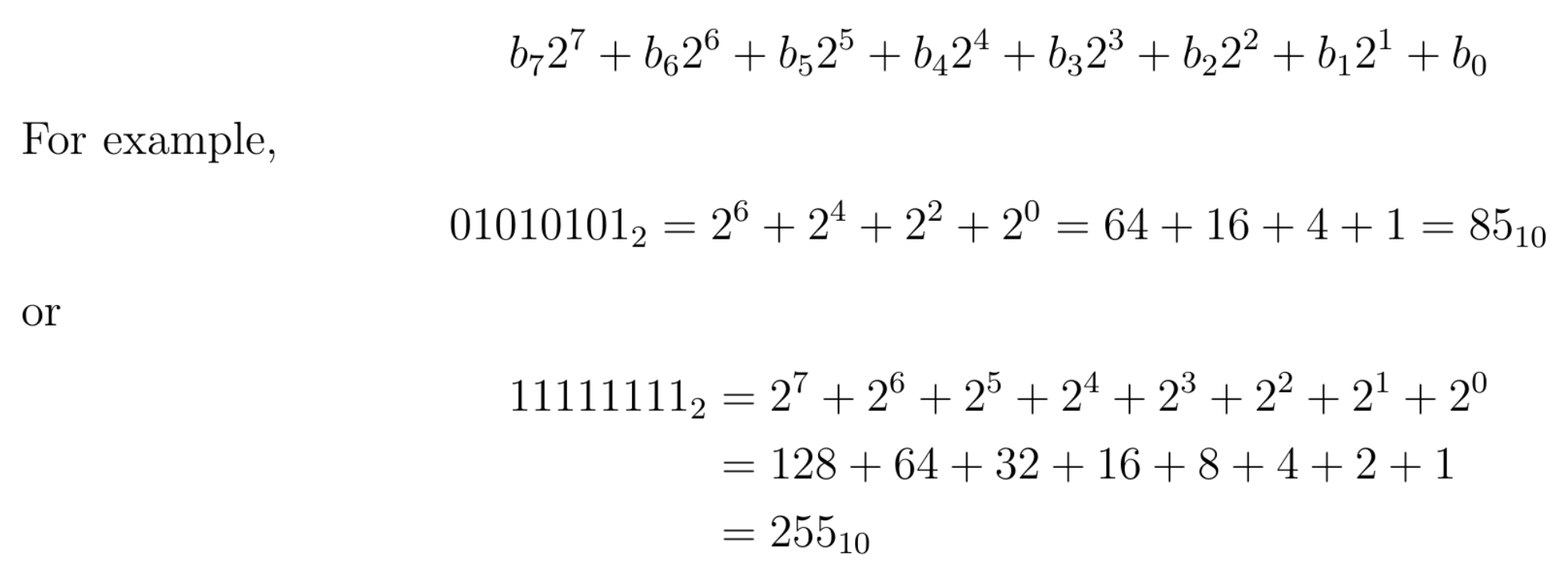
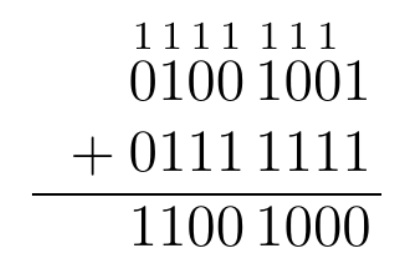
Binary Representation

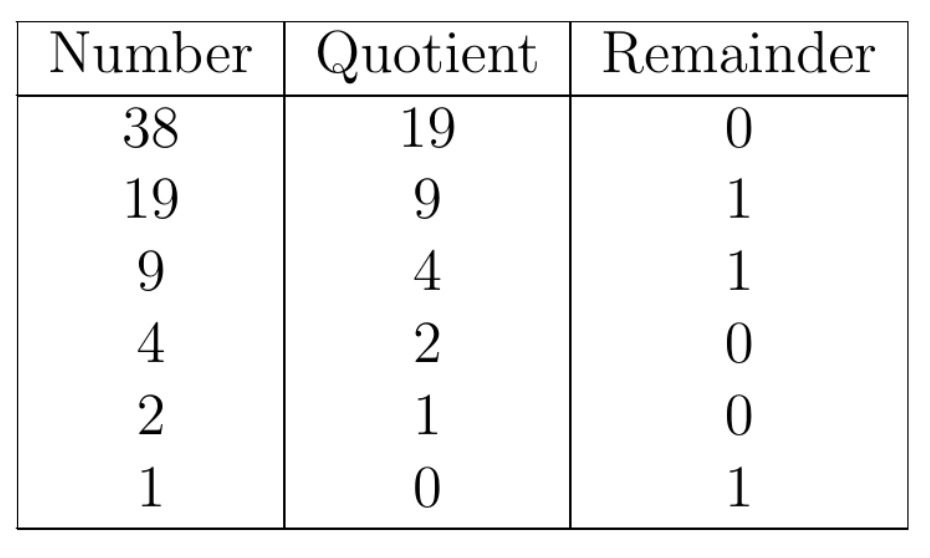
* bit (binary digit): smallest unit of data that can be rep in a computer
  + can be 0 or 1
* nibble: 4 bits
  + e.g. 1001
* byte: 8 bits
  + e.g. 10011101
* word: machine-specific grouping of bytes
  + for this course, a word is 4 bytes, which is 32-bit architecture
  + 8-byte words are more common in modern devices
* most significant bit (MSB): left-most bit
* least significant bit (LSB): right-most bit
* bytes can rep numbers, characters, garbage in memory, instructions, etc.
* 2 types of binary rep: unsigned (non-negative integers) and signed integers
  + can be other types like float point, algebraic, etc.
* for 8-bit unsigned integer, its value in decimal is:



* arithmetic with unsigned binary integers is like:



* + be careful of overflow errors (if there’s a carry when the last pair of bits of are added so the result takes up more than the max number of bits)
    - usually overflowing bits are ignored
* to convert from decimal to binary, repeatedly divide by 2 and keep track of remainders
  + write the remainders found from bottom to top and that’s the binary rep



* + 3810 = 1001102
* for signed binary integers, one approach (not used in practice) is to use sign-magnitude: use first bit to indicate sign
  + 0 is positive and 1 is negative
* use two’s complement rep for signed integers
  + first bit is 0 if non-negative and 1 if negative
  + to negate value (switch to two’s complement or vice versa), take complement of all bits and then add 1
    - can also locate rightmost 1 bit and negate all values to the left of it



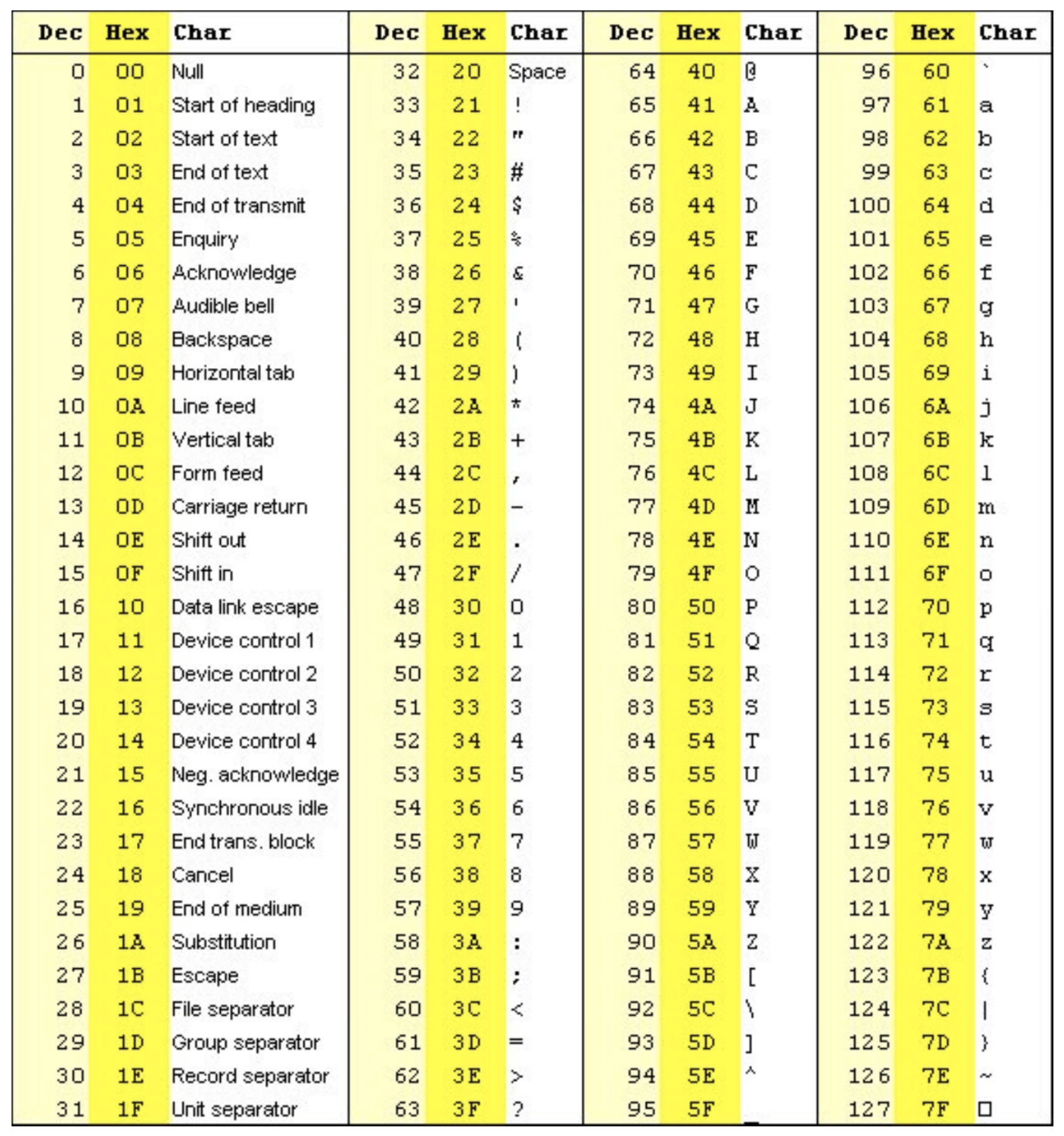
* two’s complement method works because using modular arithmetic, unsigned 8-bit integers can range from 0-255 but we’re simply adjusting the range so that half of the numbers are negative
  + reminder: in modular arithmetic, once there’s overflow, we loop back to the beginning of the number range
* for addition and subtraction, there’s no difference between signed and unsigned arithmetic
  + reminder: the answer will be in whichever form the values being added/subtracted are
* for multiplication and division, there's a difference in handling the operations to make sure the signs work out properly

Hexadecimal Representation

* hexadecimal system: base-16 rep system and consists of numbers from 0 to 9 and letters A to F
* it’s popular because it’s easy to convert from hex to binary and vice versa
  + each hexadecimal digit is a nibble so binary numbers can be converted by being grouped into 4 bits
* a more common rep of hex is to add 0x in front of the number instead of a 16 subscript

ASCII Representation

* ASCII stands for American Standard Code for Information Exchange and it’s based on the English alphabet
* can access this table from Unix command line by typing “man ascii”



* 0-31 are control characters
* 48-57 are the numbers 0-9
* 65-90 are the letters A-Z
* 97-122 are the letters a-z
  + ‘A’ and ‘a’ are 32 away from each other

Bit-Wise Operations

* bit-wise operators are used on binary numbers
  + bitwise NOT (~): applies logical NOT operator to each pair of bits in the operands
  + bitwise AND (&)
  + bitwise OR (|)
  + bitwise XOR (^)
  + left bit shift (<<): x << n is equivalent to x \* 2n
    - shift all bits to left and pad additional 0s on the right
    - overflowing bits are discarded
  + right bit shift (>>): x >> n is equivalent to x / 2n
    - shift all bits to right and pad additional 0s on the left
    - overflowing bits are discarded
    - arithmetic shift: if MSB is 1, then must know if the value is unsigned or signed
      * if unsigned, 0s are padded on the left like usual
      * if signed, 1 at MSB indicates it’s a negative number so 1s must be padded on the left
* bit-wise operators can also be combined with assignment operator: C &= 5;
* below C program demonstrates some of these bit-wise operations:

