Untitled-1

```c
/*
 * allOddBits - return 1 if all odd-numbered bits in word set to 1
 *   where bits are numbered from 0 (least significant) to 31 (most significant)
 *   Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 2
 */
int allOddBits(int x) {
  // return !((x^0xAA)&((x >> 8) ^ 0xAA)&((x >> 16) ^ 0xAA)&((x >> 24) ^ 0xAA));
  // printf("%d\n", (x^0xAA));
  // return (!(x^0xAA))|(!((x >> 8)^0xAA))|(!((x >> 16)^0xAA))|(!((x >> 24)^0xAA));
  // return ~((x & 0xAA) ^ 0xAA) | ~(((x >> 8) & 0xAA) ^ 0xAA) | ~(((x >> 16) & 0xAA) ^
0xAA) | ~(((x >> 24) & 0xAA) ^ 0xAA);

  // rightmost bit is 0th position
  // 0xAA => 0000 ... 1010 1010
  // the hex value 0xAA has the pattern of all odd-numbered bits set to 1
  // so all we have to do is to bitwise AND the value with x
  // and then shift x to the right by the size of 0xAA, which is 8 bits

  int p1 = (x & 0xAA);
  int p2 = ((x >> 8) & 0xAA);
  int p3 = ((x >> 16) & 0xAA);
  int p4 = ((x >> 24) & 0xAA);

  // p1 to p4 should all be equal to 0xAA, or 1010 1010, if x has all number bits

  // bitwise AND between p1 to p4 would still yield 0xAA if they are all equal
  // use bitwise NOT with 0xAA to compare (similar to ==)
  // because this funciton needs use to return true when it is all odd bits, we simply do
a logical NOT on the result
  return !((p1 & p2 & p3 & p4) ^ 0xAA);
}
/*
 * negate - return -x
 *   Example: negate(1) = -1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 5
 *   Rating: 2
 */
int negate(int x) {
  // very simple as long as we know that ~x = -x - 1
  // so rearranging the equation we get -x = ~x + 1

  return ~x + 1;
}
```