Bit Manipulation Lab — Problem Explanation, Solution Ideas, and Line-by-Line Commentary

This document explains a series of classic bit-level programming exercises from a data lab (like CS:APP's 'bits.c').
Each function must compute a result using restricted bitwise operators and without control overflows or undefined behavior.
We summarize for each function:
• The problem in plain words. • The core idea/derivation. • A fully annotated code listing with comments on (nearly) every line.

Notes:
– Many tasks restrict the allowed operators; where your current code slightly violates the constraints, we point that out and show a compliant variant.
– Shifts are 32-bit two’s complement (int). Right shifts are arithmetic; when we need logical-right behavior, we add masks.
– Example in the handout 'rotateRight(0x87654321,4)' should produce 0x18765432 (9 hex digits in the comment is a common typo).

tmin

Problem. Return the minimum two’s-complement 32-bit integer.

Solution idea. The minimum value has only the sign bit set: 1000...0b. That is 1 shifted left by 31.

```
int tmin(void) {
  // Compute 1 << 31 to set only the sign bit (most significant bit) to 1.
  return 1 << 31;
}
```

bitAnd

Problem. Compute x & y using only ~ and |.

Solution idea. By De Morgan: x & y == ~(~x | ~y).

```
int bitAnd(int x, int y) {
  // By De Morgan's law: AND equals NOT( NOT x OR NOT y ).
  return ~(~x | ~y);
}
```

bitXor

Problem. Compute x ^ y using only ~ and &.

Solution idea. x ^ y == (x & ~y) | (~x & y). Replace the OR with De Morgan: A | B == ~(~A & ~B). After pushing negations, we get: ~(~(x & ~y) & ~(~x & y)).

```
int bitXor(int x, int y) {
  // XOR is (x & ~y) | (~x & y). Replace '|' using De Morgan: A|B == ~(~A & ~B).
  // So x^y == ~(~(x & ~y) & ~(~x & y))
  return ~(~(x & ~y) & ~(~x & y));
}
```

negate

Problem. Return -x.

Solution idea. Two's complement negation is bitwise NOT plus 1: -x == ~x + 1.

```
int negate(int x) {
  // Two's complement negation: invert bits then add 1.
  return ~x + 1;
}
```

isEqual

Problem. Return 1 if x == y else 0 using bitwise operations.

Solution idea. x ^ y == 0 iff x == y. Then !0 => 1 and !nonzero => 0.

```
int isEqual(int x, int y) {
  // x==y iff x^y is 0; logical NOT converts 0 to 1 and nonzero to 0.
  return !(x ^ y);
}
```

satAdd

Problem. Saturating addition of two ints: clamp to Tmin/Tmax on overflow.

Solution idea. Overflow occurs only when x and y share the same sign, but the sum's sign differs. Detect positive overflow (~xsign & ~ysign & ssign) -> return Tmax; negative overflow (xsign & ysign & ~ssign) -> return Tmin; else return sum.

```
int satAdd(int x, int y) {
  int sum = x + y;           // Regular two's-complement addition
  int xsign = x >> 31;       // All 1s if x<0 else 0
  int ysign = y >> 31;       // All 1s if y<0 else 0
  int ssign = sum >> 31;     // Sign of the sum
  int pos_over = (~xsign & ~ysign & ssign);  // + + -> - (overflow to negative)
  int neg_over = (xsign & ysign & ~ssign);   // - - -> + (overflow to positive)
  int Tmax = ~(1 << 31);     // 0x7fffffff
  int Tmin = 1 << 31;        // 0x80000000
  // If positive overflow: return Tmax; if negative overflow: return Tmin; else: sum.
  return (pos_over & Tmax) | (neg_over & Tmin) | (~(pos_over | neg_over) & sum);
}
```

bitMatch

Problem. Create mask marking bit positions where x and y match (both 0 or both 1) using only ~ and &.

Solution idea. Desired: ~(x ^ y). But '^' is disallowed. Use (~(x & ~y) & ~(~x & y)) which equals ~(x ^ y) via De Morgan.

```
int bitMatch(int x, int y) {
  // Bits match where XOR would be 0. Avoid '^' by expanding:
  // ~(x ^ y) == ~( (x & ~y) | (~x & y) ) == ~ (x & ~y) & ~ (~x & y)
  return ~(x & ~y) & ~(~x & y);
}
```

fitsShort

Problem. Return 1 iff x fits in signed 16-bit two's complement.

Solution idea. Arithmetic shift left 16 then right 16; if value unchanged, top bits were sign extension only.

```
int fitsShort(int x) {
  // If shifting out and back preserves x, it fits into 16-bit two's complement.
  return !(((x << 16) >> 16) ^ x);
}
```

rotateRight

Problem. Rotate x right by n (0<=n<=31).

Solution idea. Right-rotate takes low n bits to the top while shifting the rest down. Because >> is arithmetic, mask to emulate logical right shift. Compute left = x << (32-n) and right = (x >> n) & ((1<<(32-n))-1); then OR.

```
int rotateRight(int x, int n) {
  // Compute (32 - n) in a way that stays within 0..31 for shifts
  int r = (32 + (~n + 1)) & 31;       // r = (32 - n) & 31
  int left = x << r;                   // Move low n bits into high positions
  int mask = (1 << r) + ~0;            // (1<<r) - 1 : ones in the low r positions
  int right = (x >> n) & mask;         // Arithmetic >>, then mask to logical
  return left | right;                 // Combine
}
```

byteSwap

Problem. Swap the n-th and m-th bytes (0-based) of x.

Solution idea. Extract bytes with shifts & 0xFF, clear their slots with a mask, then place them swapped.

```
int byteSwap(int x, int n, int m) {
  int nshift = n << 3;              // n * 8 to target the byte
  int mshift = m << 3;              // m * 8
  int nbyte  = (x >> nshift) & 0xFF; // extract n-th byte
```

```
  int mbyte  = (x >> mshift) & 0xFF; // extract m-th byte
  int mask   = (0xFF << nshift) | (0xFF << mshift);  // bits to clear
  int rest   = x & ~mask;            // zero-out those two byte positions
  int nput   = mbyte << nshift;      // put m's byte into n's slot
  int mput   = nbyte << mshift;      // put n's byte into m's slot
  return rest | nput | mput;         // merge
}
```

floatAbsVal

Problem. Return the IEEE-754 bit-level absolute value of uf, unless uf is NaN (then return uf).

Solution idea. Clear sign bit (mask 0x7fffffff). If result >= 0x7f800001, it's NaN (exp all ones and mantissa nonzero).

```
unsigned floatAbsVal(unsigned uf) {
  unsigned mask = 0x7FFFFFFF;  // clear sign bit
  unsigned abs  = uf & mask;   // absolute value bits
  unsigned nan  = 0x7F800001;  // smallest NaN: exp=all ones, mantissa>=1
  if (abs >= nan) return uf;   // NaN: return argument unchanged
  return abs;                  // otherwise, absolute value
}
```

floatScale2

Problem. Return bit-level representation of 2*f for single-precision uf. Preserve NaNs.

Solution idea. If exp==0 (denormals/zero): shift fraction left by 1 (keep sign). If exp==255: NaN/inf -> return uf. Else increment exponent; if it overflows to 255, return signed infinity (sign and exp).

```
unsigned floatScale2(unsigned uf) {
  unsigned sign = uf & 0x80000000;      // preserve sign
  unsigned exp  = (uf >> 23) & 0xFF;    // exponent
  unsigned frac = uf & 0x7FFFFF;        // fraction (mantissa)
  if (exp == 0) {                       // denormal or zero
    // shift fraction; note that if frac overflows into hidden 1, it becomes normal,
    // but this path keeps it denormal per typical Datalab spec.
    return sign | (frac << 1);
  }
  if (exp == 0xFF) return uf;           // NaN or infinity
  exp = exp + 1;                        // multiply by 2 => increment exponent
  if (exp == 0xFF) {                    // overflow to infinity
    return sign | (0xFF << 23);
  }
  return sign | (exp << 23) | frac;     // recombine
}
```

Caveats & Checks

• bitMatch: Your original used '^', but the legal ops were only '~' and '&'. The rewritten version complies.

• rotateRight example: Correct 32-bit result for rotateRight(0x87654321,4) is 0x18765432.

• floatScale2: For denormals, many lab specs accept simply shifting the fraction; some variants normalize when the top bit crosses. This variant matches common Datalab grading.