**Cheat Sheet**

/\*

\* bang - Compute !x without using !

\* Examples: bang(3) = 0, bang(0) = 1

\*/

int bang(int x) { //done.

return ((((~x + 1) | x) >> 31) + 1);

}

/\*

\* bitCount - returns count of number of 1's in word

\* Examples: bitCount(5) = 2, bitCount(7) = 3

\*/

int bitCount(int x) {//done.

int x1 = 0x11 |(0x11<<8);

int x2 = x1 | (x1 << 16);

int res = x&x2;

res = res+ ((x >> 1)&x2);

res = res+ ((x >> 2)&x2);

res = res+ ((x >> 3)&x2);

res = res + (res >> 16);

x1 = 0xF|(0xF << 8);

res = (res&x1) + ((res >> 4)&x1);

return ((res + (res >> 8)) & 0x3F);

}

/\*

\* bitOr - x|y using only ~ and &

\* Example: bitOr(6, 5) = 7

\*/

int bitOr(int x, int y) {//done.

return ~(~x&~y);

}

/\*

\* bitRepeat - repeat x's low-order n bits until word is full.

\*/

int bitRepeat(int x, int n) {

unsigned ux = x;

int shift = 32 - n;

int repeat = ((ux << shift) >> shift);

repeat = repeat | (repeat << n&(~((31 - n) >> 31)));

n=n << 1;

repeat = repeat | (repeat << n&(~((31 - n) >> 31)));

n=n << 1;

repeat = repeat | (repeat << n&(~((31 - n) >> 31)));

n=n << 1;

repeat = repeat | (repeat << n&(~((31 - n) >> 31)));

n=n << 1;

repeat = repeat | (repeat << n&(~((31 - n) >> 31)));

n=n << 1;

return repeat;

}

/\*

\* fitsBits - return 1 if x can be represented as an

\* n-bit, two's complement integer.

\* 1 <= n <= 32

\*/

int fitsBits(int x, int n) {//done.

return !(((~x & (x >> 31)) + (x & ~(x >> 31))) >> (n + ~0));

}

/\*

\* getByte - Extract byte n from word x

\* Bytes numbered from 0 (LSB) to 3 (MSB)

\* Examples: getByte(0x12345678,1) = 0x56

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 6

\* Rating: 2

\*/

int getByte(int x, int n) { //done.

int x1 = 0xff;

return ((x&(x1 << (n << 3))) >> (n << 3))& x1;

}

/\*

\* isLessOrEqual - if x <= y then return 1, else return 0

\*/

int isLessOrEqual(int x, int y) {//done.

/\*/int x2 = !(~(x + ~(1 << 31))); //== 1 when x = tmin, 0 otherwise

//int y2 = !(~(y + (1 << 31))); //==1 when y = tmax, 0 otherwise

int x3 = !(!(~(x + (1 << 31)))); //==0 when x=tmax, 1 otherwise

int signx = !(x >> 31);

//int signy = !(y >> 31);

//int signdiff = signy + (~signx + 1);

int k = (x^y)&signx;

int sign\_x = x >> 31;

int sign\_y = y >> 31;

int equal = !(sign\_x ^ sign\_y) & ((~y + x) >> 31);

int notEqual = sign\_x & !sign\_y;

return !(!(equal | notEqual));

/\*/return (x2 |( ((!((y + ((~x) + 1)) >> 31)) & 1) & x3) & ( | y2);

return ((((!((y + ((~x) + 1)) >> 31)) & 1) & x3) & (k));

\*/

}

/\*

\* isPositive - return 1 if x > 0, return 0 otherwise

\*/

int isPositive(int x) {//done.

return !((x&(1 << 31)) | !x);

}

/\*

\* logicalShift - shift x to the right by n, using a logical shift

\*/

int logicalShift(int x, int n) {//done.

int op = (1 << 31) >> 31;

int t = (~n + 1);

int shift = ((0x20)&t);

op = (op << shift);

return (x >> n) + (op);

}

/\*

\* tmin - return minimum two's complement integer

\*/

int tmin(void) {//done.

return 1<<31;

}

/\*

\* bitNor - ~(x|y) using only ~ and &

\*/

int bitNor(int x, int y)

{

// Applied DeMorgan's Law to expression:

return (~x & ~y);

}

/\*

\* bitXor - x^y using only ~ and &

\* Rating: 2

\*/

int bitXor(int x, int y)

{

// Applied DeMorgan's Law to expression:

// (~x & y) | (x & ~y)

return (~(~(~x & y) & ~(x & ~y)));

}

/\*

\* copyLSB - set all bits of result to least significant bit of x

\* Example: copyLSB(5) = 0xFFFFFFFF, copyLSB(6) = 0x00000000

\*/

int copyLSB(int x)

{

return ((x << 31) >> 31);

}

/\*

\* leastBitPos - return a mask that marks the position of the

\* least significant 1 bit. If x == 0, return 0

\*/

int leastBitPos(int x)

{

// The logical negative value of x.

int negative\_x = ~x + 1;

// x and the logical negative value of x combined with the & operator properly produce the desired mask.

return(x & negative\_x);

}

/\*

\* TMax - return maximum two's complement integer

\*/

int tmax(void)

{

int x = 0x80;

return(~(x << 24));

}

/\*

\* isNonNegative - return 1 if x >= 0, return 0 otherwise

\* Example: isNonNegative(-1) = 0. isNonNegative(0) = 1.

\*/

int isNonNegative(int x)

{

int sign\_x = x >> 31;

// The negation of the sign bit of value x computes the appropriate boolean return value.

return (!(sign\_x));

}

/\*

\* isGreater - if x > y then return 1, else return 0

\*/

int isGreater(int x, int y)

{

int sign\_x = x >> 31;

int sign\_y = y >> 31;

// if the signs are equal, then

// if x is larger, sign bit of (~y + x) is 0

// if y is larger, sign bit of (~y + x) is 1

int equal = !(sign\_x ^ sign\_y) & ((~y + x) >> 31);

// if signs are not equal, these principles are reversed.

int notEqual = sign\_x & !sign\_y;

// this | returns 0 when it is x is greater, so you have to negate it.

return !( equal | notEqual);

}

/\*

\* divpwr2 - Compute x/(2^n), for 0 <= n <= 30

\*/

int divpwr2(int x, int n)

{

// Something is needed to account for x >> n if positive and x >> n + 1 if negative

// Subtract 1 from 2^n

// This accounts for the need to + 1

int mask = (1 << n) + ~0;

// Use & operator on mask and sign bit of x

int equalizer = (x >> 31) & mask;

// Adds 1 if x was originally negative

// Adds 0 if x was originally positive

return (x + equalizer) >> n;

}

/\*

\* abs - absolute value of x (except returns TMin for TMin)

\*/

int abs(int x)

{

// Boolean value indicating sign of x

int sign\_x = x >> 31;

// XOR of value x and the sign bit of value x plus 1 plus the compliment of the sign bit of x returns the absolute value of x.

return((x ^ (sign\_x)) + (1 + ( ~(sign\_x))));

}

/\*

\* addOK - Determine if can compute x+y without overflow

\* Example: addOK(0x80000000,0x80000000) = 0,

\* addOK(0x80000000,0x70000000) = 1,

\*/

int addOK(int x, int y)

{

int xy\_sum = x + y;

int sign\_x = x >> 31;

// Boolean value indicating sign of y

// 1 = Negative

int sign\_y = y >> 31;

int sign\_sum\_xy = xy\_sum >> 31;

// An overflow occurs when the sign of x and y are the same, but the sign of the sum of x and y is different

return !(~(sign\_x ^ sign\_y) & (sign\_x ^ sign\_sum\_xy));

}

\*

\* allOddBits - return 1 if all odd-numbered bits in word set to 1

\* Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1

\*/

int allOddBits(int x) {

// build mask: 0xAAAAAAAA

int mask = (0xAA << 8) + 0xAA;

mask = (mask << 16) + mask;

return !((x & mask) ^ mask);

}

/\*

\* reverseBytes - reverse the bytes of x

\* Example: reverseBytes(0x01020304) = 0x04030201

\*/

int reverseBytes(int x) {

// get each byte by right shift

int mask = 0xFF;

int byte0 = x & mask;

int byte1 = (x >> 8) & mask;

int byte2 = (x >> 16) & mask;

int byte3 = (x >> 24) & mask;

// assembly each byte reversely

return byte3 | (byte2 << 8) | (byte1 << 16) | (byte0 << 24);

}

int bitMask(int highbit, int lowbit) {

/\* build two sub-masks based on highbit and lowbit

\* highPart is to right shift -1 to (highbit + 1) bits

\* lowPart is to right shift -1 to lowbit bits

\*/

int negOne = ~0;

int highPart = (negOne << highbit) << 1;

int lowPart = negOne << lowbit;

// using xor to get the different part

// using lowPart as a mask when lowbit > highbit

return (highPart ^ lowPart) & lowPart;

}

/\*

\* isNegative - return 1 if x < 0, return 0 otherwise

s\*/

int isNegative(int x) {

return (x >> 31) & 1;

}

/\*

\* negate - return -x

\* Example: negate(1) = -1.

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 5

\* Rating: 2

\*/

int negate(int x) {

return ~x + 1;

}

/\*

\* subOK - Determine if can compute x-y without overflow

\* Example: subOK(0x80000000,0x80000000) = 1,

\* subOK(0x80000000,0x70000000) = 0,

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 20

\* Rating: 3

\*/

int subOK(int x, int y) {

/\*

\* overflow of sub happens iff

\* 1) x and y have different signs

\* 2) res = x - y has different sign with x

\*/

int res = x + (~y + 1);

int sameSign = x ^ y;

int resSign = res ^ x;

return !((sameSign & resSign) >> 31);

}

/\*

\* isLess - if x < y then return 1, else return 0

\* Example: isLess(4,5) = 1.

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 24

\* Rating: 3

\*/

int isLess(int x, int y) {

/\*

\* there are 2 cases can lead to x < y

\* 1) x < 0 and y >= 0

\* 2) x and y have the same sign,

\* and x - y is negative

\*/

int res = x + (~y + 1);

int flag1 = x & (~y);

int flag2 = (~(x ^ y)) & res;

return (flag1 | flag2) >> 31 & 1;

}