|  |  |  |
| --- | --- | --- |
| char | 1 | 1 |
| short int | 2 | 2 |
| int | 4 | 4 |
| long int | 4 | 8 |
| long long int | 8 | 8 |
| pointer | 4 | 8 |
| float | 4 | 4 |
| double | 8 | 8 |

|  |  |  |
| --- | --- | --- |
| not | ~ | ! |
| and | & | && |
| or | | | || |
| xor | ^ |  |

|  |  |  |
| --- | --- | --- |
| cast | overfl | round |
| int>fl | no | yes |
| fl>dbl | no | no |
| dbl>fl | ±∞ | yes |
| fl/dbl>int | yes | yes, to 0 |

|  |  |  |
| --- | --- | --- |
| %eax | %ax | %al |
| %ecx | %cx | %cl |
| %edx | %dx | %dl |
| %ebx | %bx | %bl |
| %esi | %si |  |
| %edi | %di |  |
| %esp | %sp | <-stack |
| %ebp | %bp | <-frame ptr |

|  |  |
| --- | --- |
| inc D | D<D+1 |
| dec D | D<D-1 |
| neg D | D<-D |
| not D | D<~D |
| add S,D | D<D+S |
| sub S,D | D<D-S |
| imul S,D | D<D\*S |
| xor S,D | D<D^S |
| or S,D | D<D|S |
| and S,D | D<D&S |
| sal K,D | D<<K |
| shl K,D | D<<K |
| sar K,D | D>>Ka |
| shr K,D | D>>Kl |
| CF-bel. unsign | carry |
| ZF-equa | zero |
| SF-neg | sign |
| OF | overfl |

|  |  |
| --- | --- |
| sete D | D<ZF |
| setne D | D<~ZF |
| sets D | D<SF |
| setns D | D<~SF |
| setge D | ~(SF^OF) |
| setl D | D<SF^OF |
| seta D | D<~CF&~ZF |
| setae D | (above unsi) |
| setb D | below |
| setbe D |  |
| jmp lbl |  |
| je label | ZF |
| jns | ~SF |
| jl | SF^OF |
| jb | CF |
| leal S,D | D<&S |

|  |  |
| --- | --- |
| leal | (%eax, %eax, 2), %edx  x+2x = 3x |

print(“v = %d, uv=%u\n”, v, uv);

Right shifting:

logical (0’s) – unsigned

arith (most sig bit) - signed

2’s complement negation

fixed precision unsigned **integer arithmetic**:

9+12 = 21 (4 bits = 5) (9+12)mod16 = 5

determine **overflow**: sum < both #’s

ie. w=4, x=7, y=3. x+y = -6 (positive overflow)

If adding 2 ints of same sign, result must be same sign, otherwise overflow

**multiply** by constants:

shift left and add (power of 2)

**dividing** by powers of 2:

unsigned integer division: right shift (logical)

signed integer division: right shift (arithmetic)

Floating Point Rep:

v= (-1)^s \* M \* 2^E

1. Normalized:

E = e – bias

Bias = 2^(k-1) – 1

2. Denormalized:

E = 1 – bias

Bias = (given?)

3. Specialized

1.1 | 000..000 = +- infin

1.1 | != 0 = NaN

**Operand Poss.**

Memory:

absolute: 0x100

indirect: (%eax)

base + displacement: 4(%eax)

indexed: (%eax, %eac)

indexed: 4(%eax, %eac)

scaled index: (, %eax,2)

**Gen Form Loops**

T=test\_expr

If(!t)

goto false

then\_statement;

goto done;

false:

else statement;

done:

if(t)

goto true;

else

goto done;

/\*

\* logicalNeg - implement the ! operator, using all of

\* the legal operators except !

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

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

\* Max ops: 12

\* Rating: 4

\*/

int logicalNeg(int x) {

/\*

if x or -x has a 1 in the sign place, it isn't 0

that xor'd 1 will be 1 if x is 0, and 0 otherwise

\*/

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

}

/\*

\* negate - return -x

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

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

\* Max ops: 5

\* Rating: 2

\*/

int negate(int x) {

/\*

by definition of negate

\*/

return ~x + 1;

}

/\*

\* multFiveEights - multiplies by 5/8 rounding toward 0.

\* Examples: multFiveEights(77) = 48

\* multFiveEights(-22) = -13

\* You can assume |x| < (1 << 29)

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

\* Max ops: 12

\* Rating: 3

\*/

int multFiveEights(int x) {

x = (x << 2) + x; //x\*5

return (x + ((x >> 31) & 7)) >> 3; //x/8

}

/\*

\* isPower2 - returns 1 if x is a power of 2, and 0 otherwise

\* Examples: isPower2(5) = 0, isPower2(8) = 1, isPower2(0) = 0

\* Note that no negative number is a power of 2.

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

\* Max ops: 60

\* Rating: 4

\*/

int isPower2(int x) {

/\*

boundary cases are 0 and all negative numbers

if x and x-1 have any bits in common, then x is not a power of 2

to solve x = 0,

the above will result in 1, so xor'ing with !x will result in 0

if x =0 or x if x != 0. and'ing the combined result with the not of the

sign of x, 0 would result if x is neg and 1 if x is pos, will eliminate

the need about neg #'s

\*/

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

}

/\*

\* conditional - same as x ? y : z

\* Example: conditional(2,4,5) = 4

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

\* Max ops: 16

\* Rating: 3

\*/

int conditional(int x, int y, int z) {

/\*

turn x into a mask - 1's if x is false, 0's if true

x negated to be all 1's if true or 0's if false

then and'ed with y and z

\*/

x = ((!x) << 31) >> 31;

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

}

/\*

\* 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) {

/\*

f computes absolute difference between x and y

x and y must have opposite signs

f (which is 0 or a 1) determines which # is closer to 0,

if f = 1, y is farther from 0 and the answer is opposite sign of y

if f = 0, x is closer to 0, and the answer depends on the sign of x

\*/

unsigned int xSign = x >> 31;

unsigned int ySign = y >> 31;

//1...1 if x&y are same signs

unsigned int changeSign = xSign ^ ~ySign;

unsigned int f;

y = ((~y + 1) & changeSign) | (y & ~changeSign);

//f is the absolute difference between x and y

f = (((y + x) ^ xSign) >> 31) & 1;

return ((f & !ySign) | ((!f) & xSign));

}

/\*

\* isNonZero - Check whether x is nonzero using

\* the legal operators except !

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

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

\* Max ops: 10

\* Rating: 4

\*/

int isNonZero(int x) {

/\*

if x or -x has a 1 in the sign place, it isn't 0

\*/

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

}

/\*

\* 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) {

/\*

and'ing with 0xFF to get a byte with 1's in that byte and 0's

everywhere else

shift by 8\*n to get the correct byte in x

\*/

return 0xFF&(x >>(n<<3));

}

/\*

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

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

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

\* Max ops: 5

\* Rating: 2

\*/

int copyLSB(int x) {

/\*

add -1 to opposite of the last digit of x

if xmod2 = 0, ~xmod2 = 1, and -1 + 1 = 0

if xmod2 = 1, ~xmod2 = 0, and -1 + 0 = -1 (1's)

\*/

return (~0) + ((x & 1) ^ 1);

}

/\*

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

\* Example: bitXor(4, 5) = 1

\* Legal ops: ~ &

\* Max ops: 14

\* Rating: 2

\*/

int bitXor(int x, int y) {

/\*

xor: A^B = (~A&B)|(A&~B)

demorgan's law to change | to &

\*/

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

}

/\*

\* bitParity - returns 1 if x contains an odd number of 0's

\* Examples: bitParity(5) = 0, bitParity(7) = 1

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

\* Max ops: 20

\* Rating: 4

\*/

int bitParity(int x) {

/\*

split the set into 2 and eliminate double 1's

keep going until only 2 bits are compared

\*/

x = (x >> 16) ^ x;

x = (x >> 8) ^ x;

x = (x >> 4) ^ x;

x = (x >> 2) ^ x;

x = (x >> 1) ^ x;

return x&1;

}

/\*

\* bitMask - Generate a mask consisting of all 1's

\* lowbit and highbit

\* Examples: bitMask(5,3) = 0x38

\* Assume 0 <= lowbit <= 31, and 0 <= highbit <= 31

\* If lowbit > highbit, then mask should be all 0's

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

\*/

int bitMask(int highbit, int lowbit) {

/\*

Shift set of 1's left by highbit + 1

Negate it to include correct bits as 1's

the right bits become 1's

next, shift set of 1's left by lowbit

the set will have correct bits and to the left

and'ing the two will get 1's in needed spots

\*/

return ~((~0 << (highbit)) << 1) & ((~0) << lowbit);

}