How can I multiply and divide using only bit shifting and adding?

How can I multiply and divide using only bit shifting and adding?

c assembly bit-manipulation division multiplication

edited Oct 3 '12 at 10:24

ЯegDwight
19.4k 9 34 47

asked May 5 '10 at 19:35

Spidfire
2,345 5 19 34

18 Like you would do it on paper in middle school, only using binary instead of decimal. – Pascal Cuoq May 5 '10 at 19:36

1 @mtk: What is missing from this answer? Are you looking for C or assembly implementation, particular operand widths, a specific division method (e.g. restoring vs non-restoring)? – njuffa Sep 7 '15 at 7:05

Is subtraction OK? All seems to be covered – mksteve Sep 9 '15 at 20:40

What is the need behind this question? CPU's translate multiply and division operations into bitshifting and addition or subtraction already, and if that is if the compiler hasn't already done so. – Kelly S. French Jun 13 '16 at 15:28

@KellyS.French Just curiosity, it's more a way to imagine how a compiler can work with a restricted instruction set. – Spidfire Jun 14 '16 at 8:13

13 Answers

To multiply in terms of adding and shifting you want to decompose one of the numbers by powers of two, like so:

```
21 * 5 = 10101_2 * 101_2             (Initial step)
       = 10101_2 * (1 * 2^2  +  0 * 2^1  +  1 * 2^0)
       = 10101_2 * 2^2 + 10101_2 * 2^0
       = 10101_2 << 2 + 10101_2 << 0 (Decomposed)
       = 10101_2 * 4 + 10101_2 * 1
       = 10101_2 * 5
       = 21 * 5                      (Same as initial expression)
```

(`_2` means base 2)

As you can see, multiplication can be decomposed into adding and shifting and back again. This is also why multiplication takes longer than bit shifts or adding - it's O(n^2) rather than O(n) in the number of bits. Real computer systems (as opposed to theoretical computer systems) have a finite number of bits, so multiplication takes a constant multiple of time compared to addition and shifting. If I recall correctly, modern processors, if pipelined properly, can do multiplication just about as fast as addition, by messing with the utilization of the ALUs (arithmetic units) in the processor.

answered May 5 '10 at 22:31

Andrew Toulouse
833 1 8 10

3 I know it was a while ago, but could you give an example with division? Thanks – GniruT Oct 14 '15 at 13:23

The answer by Andrew Toulouse can be extended to division.

The division by integer constants is considered in details in the book "Hacker's Delight" by Henry S. Warren (ISBN 9780201914658).

The first idea for implementing division is to write the inverse value of the denominator in base two.

E.g., `1/3 = (base-2) 0.0101 0101 0101 0101 0101 0101 0101 0101 .....`

So, `a/3 = (a >> 2) + (a >> 4) + (a >> 6) + ... + (a >> 30)` for 32-bit arithmetics.

By combining the terms in an obvious manner we can reduce the number of operations:

```
b = (a >> 2) + (a >> 4)

b += (b >> 4)

b += (b >> 8)

b += (b >> 16)
```

There are more exciting ways to calculate division and remainders.

EDIT1:

If the OP means multiplication and division of arbitrary numbers, not the division by a constant number, then this thread might be of use: https://stackoverflow.com/a/12699549/1182653

EDIT2:

One of the fastest ways to divide by integer constants is to exploit the modular arithmetics and Montgomery reduction: What's the fastest way to divide an integer by 3?

edited May 23 at 12:02
Community ♦
1 1

answered Apr 25 '12 at 23:37

Viktor Latypov
11.7k 2 24 43

Thanks so much for the Hacker's Delight reference! – alecxe Aug 6 '15 at 17:35

Ehm yes, this answer (division by constant) is only *partially* correct. If you try to do '3/3' you'll end up with 0. In Hacker's Delight, they actually explain that there is an error that you have to compensate for. In this case: `b += r * 11 >> 5` with `r = a - q * 3` . Link: hackersdelight.org/divcMore.pdf page 2+. – atlaste Apr 18 '16 at 8:30

X * 2 = 1 bit shift left
X / 2 = 1 bit shift right
X * 3 = shift left 1 bit and then add X

edited May 5 '10 at 23:34

answered May 5 '10 at 19:38

Kelly S. French
9,943 7 43 84

3 Do you mean `add X` for that last one? – Mark Byers May 5 '10 at 19:39

Fixed typo. Thanks for the correction. – Kelly S. French May 5 '10 at 20:05

1 It's still wrong - last line should read: "X * 3 = shift left 1 bit and then add X" – Paul R May 5 '10 at 21:02

"X / 2 = 1 bit shift right", not entirely, it rounds down to infinity, rather than up to 0 (for negative numbers), which is the usual implementation of division (at least as far as I've seen). – Leif Andersen Aug 27 '11 at 18:26

---

1. A left shift by 1 position is analogous to multiplying by 2. A right shift is analogous to dividing by 2.
2. You can add in a loop to multiply. By picking the loop variable and the addition variable correctly, you can bound performance. Once you've explored that, you should use Peasant Multiplication

answered May 5 '10 at 19:38

Yann Ramin
29.3k 1 43 71

9 +1: But the left shift isn't just analogous to multiplying by 2. It *is* multiplying by 2. At least until overflow... – Don Roby May 5 '10 at 22:20

Shift-division yields incorrect results for negative numbers. – David Feb 5 at 3:57

```
x << k == x multiplied by 2 to the power of k
x >> k == x divided by 2 to the power of k
```

You can use these shifts to do any multiplication operation. For example:

```
x * 14 == x * 16 - x * 2 == (x << 4) - (x << 1)
x * 12 == x * 8 + x * 4 == (x << 3) + (x << 2)
```

To divide a number by a non-power of two, I'm not aware of any easy way, unless you want to implement some low-level logic, use other binary operations and use some form of iteration.

edited Jan 13 '14 at 22:21

answered May 5 '10 at 19:44

IVlad

35k 10 71 158

@IVlad: How would you combine the above operations to perform, say, divide by 3 ? – Paul R May 5 '10 at 21:04

@Paul R - true, that's harder. I've clarified my answer. – IVlad May 5 '10 at 21:30

division by a constant is not too hard (multiply by magic constant and then divide by power of 2), but division by a variable is a little trickier. – Paul R May 6 '10 at 5:57

1 shouldn't x * 14 == x * 16 - x * 2 == (x << 4) - (x << 2) really end up being (x<<4) - (x<<1) since x<<1 is multiplying by x by 2? – Alex Spencer Jan 13 '14 at 20:04

@AlexSpencer - yes, you're correct. Thank you! – IVlad Jan 13 '14 at 22:21

I translated the Python code to C. The example given had a minor flaw. If the dividend value that took up all the 32 bits, the shift would fail. I just used 64-bit variables internally to work around the problem:

```
int No_divide(int nDivisor, int nDividend, int *nRemainder)
{
    int nQuotient = 0;
    int nPos = -1;
    unsigned long long ullDivisor = nDivisor;
    unsigned long long ullDividend = nDividend;

    while (ullDivisor <  ullDividend)
    {
        ullDivisor <<= 1;
        nPos ++;
    }

    ullDivisor >>= 1;

    while (nPos > -1)
    {
        if (ullDividend >= ullDivisor)
        {
            nQuotient += (1 << nPos);
            ullDividend -= ullDivisor;
        }

        ullDivisor >>= 1;
        nPos -= 1;
    }

    *nRemainder = (int) ullDividend;

    return nQuotient;
}
```

edited Aug 8 '15 at 20:58

Peter Mortensen

11.3k 16 78 110

answered Nov 5 '13 at 2:04

user2954726

131 1

What about negative number? I tested -12345 with 10 using eclipse + CDT, but the result was not that good. – kenmux Jun 23 '16 at 2:46

Take two numbers, lets say 9 and 10, write them as binary - 1001 and 1010.

Start with a result, R, of 0.

Take one of the numbers, 1010 in this case, we'll call it A, and shift it right by one bit, if you shift out a one, add the first number, we'll call it B, to R.

Now shift B left by one bit and repeat until all bits have been shifted out of A.

It's easier to see what's going on if you see it written out, this is the example:

```
      0
   0000      0
  10010      1
 000000      0
1001000      1
 ------
1011010
```

answered May 5 '10 at 22:11

Jimmeh
631 5 14

This seems fastest, just requires a little extra coding to loop through the bits of the smallest number and compute the result. – Hellonearthis Jan 15 '12 at 15:28

Taken from here.

This is only for division:

```
int add(int a, int b) {
        int partialSum, carry;
        do {
            partialSum = a ^ b;
            carry = (a & b) << 1;
            a = partialSum;
            b = carry;
        } while (carry != 0);
        return partialSum;
}

int subtract(int a, int b) {
    return add(a, add(~b, 1));
}

int division(int dividend, int divisor) {
        boolean negative = false;
        if ((dividend & (1 << 31)) == (1 << 31)) { // Check for signed bit
            negative = !negative;
            dividend = add(~dividend, 1);  // Negation
        }
        if ((divisor & (1 << 31)) == (1 << 31)) {
            negative = !negative;
            divisor = add(~divisor, 1);  // Negation
        }
        int quotient = 0;
        long r;
        for (int i = 30; i >= 0; i = subtract(i, 1)) {
            r = (divisor << i);
           // Left shift divisor until it's smaller than dividend
            if (r < Integer.MAX_VALUE && r >= 0) { // Avoid cases where comparison
between long and int doesn't make sense
                if (r <= dividend) {
                    quotient |= (1 << i);
                    dividend = subtract(dividend, (int) r);
                }
            }
        }
        if (negative) {
            quotient = add(~quotient, 1);
        }
        return quotient;
}
```

answered Feb 28 '16 at 12:26

Ashish Ahuja
3,719 8 25 57

This should work for multiplication:

```
.data

.text
.globl  main

main:

# $4 * $5 = $2

    addi $4, $0, 0x9
    addi $5, $0, 0x6
```

```
    add  $2, $0, $0 # initialize product to zero

Loop:
    beq  $5, $0, Exit # if multiplier is 0,terminate loop
    andi $3, $5, 1 # mask out the 0th bit in multiplier
    beq  $3, $0, Shift # if the bit is 0, skip add
    addu $2, $2, $4 # add (shifted) multiplicand to product

Shift:
    sll $4, $4, 1 # shift up the multiplicand 1 bit
    srl $5, $5, 1 # shift down the multiplier 1 bit
    j Loop # go for next

Exit: #


EXIT:
li $v0,10
syscall
```

answered Sep 13 '12 at 23:44

Melsi
1,239 11 17

What flavor of assembly? – Keith Pinson Apr 1 '13 at 15:36

1 It is MIPS assembly, if this is what you are asking. I think I used MARS to write/run it. – Melsi Apr 3 '13 at 9:34

The below method is the implementation of binary divide considering both numbers are positive. If subtraction is a concern we can implement that as well using binary operators.

Code

```
-(int)binaryDivide:(int)numerator with:(int)denominator
{
    if (numerator == 0 || denominator == 1) {
        return numerator;
    }

    if (denominator == 0) {

        #ifdef DEBUG
            NSAssert(denominator==0, @"denominator should be greater then 0");
        #endif
        return INFINITY;
    }

    // if (numerator <0) {
    //     numerator = abs(numerator);
    // }

    int maxBitDenom = [self getMaxBit:denominator];
    int maxBitNumerator = [self getMaxBit:numerator];
    int msbNumber = [self getMSB:maxBitDenom ofNumber:numerator];

    int qoutient = 0;

    int subResult = 0;

    int remainingBits = maxBitNumerator-maxBitDenom;

    if (msbNumber >= denominator) {
        qoutient |=1;
        subResult = msbNumber - denominator;
    }
    else {
        subResult = msbNumber;
    }

    while (remainingBits > 0) {
        int msbBit = (numerator & (1 << (remainingBits-1)))>0?1:0;
        subResult = (subResult << 1) | msbBit;
        if(subResult >= denominator) {
            subResult = subResult - denominator;
            qoutient= (qoutient << 1) | 1;
        }
        else{
            qoutient = qoutient << 1;
        }
        remainingBits--;

    }
    return qoutient;
}

-(int)getMaxBit:(int)inputNumber
{
```

```
    int maxBit = 0;
    BOOL isMaxBitSet = NO;
    for (int i=0; i<sizeof(inputNumber)*8; i++) {
        if (inputNumber & (1<<i)) {
            maxBit = i;
            isMaxBitSet=YES;
        }
    }
    if (isMaxBitSet) {
        maxBit+=1;
    }
    return maxBit;
}


-(int)getMSB:(int)bits ofNumber:(int)number
{
    int numbeMaxBit = [self getMaxBit:number];
    return number >> (numbeMaxBit - bits);
}
```

For multiplication:

```
-(int)multiplyNumber:(int)num1 withNumber:(int)num2
{
    int mulResult = 0;
    int ithBit;

    BOOL isNegativeSign = (num1<0 && num2>0) || (num1>0 && num2<0);
    num1 = abs(num1);
    num2 = abs(num2);


    for (int i=0; i<sizeof(num2)*8; i++)
    {
        ithBit =  num2 & (1<<i);
        if (ithBit>0) {
            mulResult += (num1 << i);
        }

    }

    if (isNegativeSign) {
        mulResult =  ((~mulResult)+1);
    }

    return mulResult;
}
```

edited Aug 8 '15 at 21:04 — Peter Mortensen — 11.3k 16 78 110

answered Feb 12 '14 at 20:37 — muzz — 3,546 2 18 14

A procedure for dividing integers that uses shifts and adds can be derived in straightforward fashion from decimal longhand division as taught in elementary school. The selection of each quotient digit is simplified, as the digit is either 0 and 1: if the current remainder is greater than or equal to the divisor, the least significant bit of the partial quotient is 1.

Just as with decimal longhand division, the digits of the dividend are considered from most significant to least significant, one digit at a time. This is easily accomplished by a left shift in binary division. Also, quotient bits are gathered by left shifting the current quotient bits by one position, then appending the new quotient bit.

In a classical arrangement, these two left shifts are combined into left shifting of one register pair. The upper half holds the current remainder, the lower half initial holds the dividend. As the dividend bits are transferred to the remainder register by left shift, the unused least significant bits of the lower half are used to accumulate the quotient bits.

Below is x86 assembly language and C implementations of this algorithm. This particular variant of a shift & add division is sometimes referred to as the "no-performing" variant, as the subtraction of the divisor from the current remainder is not performed unless the remainder is greater than or equal to the divisor. In C, there is no notion of the carry flag used by the assembly version in the register pair left shift. Instead, it is emulated, based on the observation that the result of an addition modulo 2^n can be smaller that either addend only if there was a carry out.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define USE_ASM 0

#if USE_ASM
uint32_t bitwise_division (uint32_t dividend, uint32_t divisor)
{
    uint32_t quot;
```

```
    __asm {
        mov  eax, [dividend];// quot = dividend
        mov  ecx, [divisor]; // divisor
        mov  edx, 32;        // bits_left
        mov  ebx, 0;         // rem
    $div_loop:
        add  eax, eax;       // (rem:quot) << 1
        adc  ebx, ebx;       //  ...
        cmp  ebx, ecx;       // rem >= divisor ?
        jb  $quot_bit_is_0;  // if (rem < divisor)
    $quot_bit_is_1:          //
        sub  ebx, ecx;       // rem = rem - divisor
        add  eax, 1;         // quot++
    $quot_bit_is_0:
        dec  edx;            // bits_left--
        jnz  $div_loop;      // while (bits_left)
        mov  [quot], eax;    // quot
    }
    return quot;
}
#else
uint32_t bitwise_division (uint32_t dividend, uint32_t divisor)
{
    uint32_t quot, rem, t;
    int bits_left = CHAR_BIT * sizeof (uint32_t);

    quot = dividend;
    rem = 0;
    do {
            // (rem:quot) << 1
            t = quot;
            quot = quot + quot;
            rem = rem + rem + (quot < t);

            if (rem >= divisor) {
                rem = rem - divisor;
                quot = quot + 1;
            }
            bits_left--;
    } while (bits_left);
    return quot;
}
#endif
```

edited Sep 17 '15 at 9:56

answered Sep 7 '15 at 17:05

njuffa
10.6k 3 33 65

@greybeard Thanks for the pointer, you are correct, I mixed up the dividend with the quotient. I'll fix it. – njuffa Sep 17 '15 at 9:55

For anyone interested in a 16-bit x86 solution, there is a piece of code by **JasonKnight** here1 (he also includes a signed multiply piece, which I haven't tested). However, that code has issues with large inputs, where the "add bx,bx" part would overflow.

The fixed version:

```
softwareMultiply:
;    INPUT  CX,BX
;   OUTPUT  DX:AX - 32 bits
; CLOBBERS  BX,CX,DI
    xor   ax,ax     ; cheap way to zero a reg
    mov   dx,ax     ; 1 clock faster than xor
    mov   di,cx
    or    di,bx     ; cheap way to test for zero on both regs
    jz    @done
    mov   di,ax     ; DI used for reg,reg adc
@loop:
    shr   cx,1      ; divide by two, bottom bit moved to carry flag
    jnc   @skipAddToResult
    add   ax,bx
    adc   dx,di     ; reg,reg is faster than reg,imm16
@skipAddToResult:
    add   bx,bx     ; faster than shift or mul
    adc   di,di
    or    cx,cx     ; fast zero check
    jnz   @loop
@done:
    ret
```

Or the same in GCC inline assembly:

```
asm("mov $0,%%ax\n\t"
    "mov $0,%%dx\n\t"
    "mov %%cx,%%di\n\t"
    "or %%bx,%%di\n\t"
    "jz done\n\t"
    "mov %%ax,%%di\n\t"
```

```
    "loop:\n\t"
    "shr $1,%%cx\n\t"
    "jnc skipAddToResult\n\t"
    "add %%bx,%%ax\n\t"
    "adc %%di,%%dx\n\t"
    "skipAddToResult:\n\t"
    "add %%bx,%%bx\n\t"
    "adc %%di,%%di\n\t"
    "or %%cx,%%cx\n\t"
    "jnz loop\n\t"
    "done:\n\t"
    : "=d" (dx), "=a" (ax)
    : "b" (bx), "c" (cx)
    : "ecx", "edi"
);
```

answered Jul 14 '15 at 1:38

axic
161 5

Try this. https://gist.github.com/swguru/5219592

```
import sys
# implement divide operation without using built-in divide operator
def divAndMod_slow(y,x, debug=0):
    r = 0
    while y >= x:
            r += 1
            y -= x
    return r,y


# implement divide operation without using built-in divide operator
def divAndMod(y,x, debug=0):

    ## find the highest position of positive bit of the ratio
    pos = -1
    while y >= x:
            pos += 1
            x <<= 1
    x >>= 1
    if debug: print "y=%d, x=%d, pos=%d" % (y,x,pos)

    if pos == -1:
            return 0, y

    r = 0
    while pos >= 0:
            if y >= x:
                    r += (1 << pos)
                    y -= x
            if debug: print "y=%d, x=%d, r=%d, pos=%d" % (y,x,r,pos)

            x >>= 1
            pos -= 1

    return r, y


if __name__ =="__main__":
    if len(sys.argv) == 3:
        y = int(sys.argv[1])
        x = int(sys.argv[2])
     else:
            y = 313271356
            x = 7

print "=== Slow Version ...."
res = divAndMod_slow( y, x)
print "%d = %d * %d + %d" % (y, x, res[0], res[1])

print "=== Fast Version ...."
res = divAndMod( y, x, debug=1)
print "%d = %d * %d + %d" % (y, x, res[0], res[1])
```

answered Apr 25 '13 at 12:04

swguru.net
7 2

5 This looks like python. The question was asked for assembly and/or C. – void Apr 25 '13 at 12:24