Ruby - Feature #15166

2.5 times faster implementation than current gcd implmentation

09/26/2018 06:30 PM - jzakiya (Jabari Zakiya)

Status:	Assigned	
Priority:	Normal	
Assignee:	watson1978 (Shizuo Fujita)	
Target version:		

Description

This is to be more explicit (and accurate) than https://bugs.ruby-lang.org/issues/15161

This is my modified gcd benchmarks code, originally presented by Daniel Lemire (see 15161).

https://gist.github.com/jzakiya/44eae4feeda8f6b048e19ff41a0c6566

Ruby's current implementation of Stein's gcd algorithm is only slightly faster than the code posted on the wikepedia page, and over 2.5 times slower than the fastest implementation in the benchmarks.

```
[jzakiya@localhost ~]$ ./gcdbenchmarks
gcd between numbers in [1 and 2000]

gcdwikipedia7fast32   :  time = 99
gcdwikipedia4fast     :  time = 121
gcdFranke             :  time = 126
gcdwikipedia3fast     :  time = 134
gcdwikipedia2fastswap :  time = 136
gcdwikipedia5fast     :  time = 139
gcdwikipedia7fast     :  time = 138
gcdwikipedia2fast     :  time = 136
gcdwikipedia6fastxchg :  time = 144
gcdwikipedia2fastxchg :  time = 156
gcd_iterative_mod     :  time = 210
gcd_recursive         :  time = 215
basicgcd              :  time = 211
rubygcd               :  time = 267
gcdwikipedia2         :  time = 321
gcd between numbers in [1000000001 and 1000002000]

gcdwikipedia7fast32   :  time = 100
gcdwikipedia4fast     :  time = 121
gcdFranke             :  time = 126
gcdwikipedia3fast     :  time = 134
gcdwikipedia2fastswap :  time = 136
gcdwikipedia5fast     :  time = 138
gcdwikipedia7fast     :  time = 138
gcdwikipedia2fast     :  time = 136
gcdwikipedia6fastxchg :  time = 144
gcdwikipedia2fastxchg :  time = 156
gcd_iterative_mod     :  time = 210
gcd_recursive         :  time = 215
basicgcd              :  time = 211
rubygcd               :  time = 269
gcdwikipedia2         :  time = 323
```

This is Ruby's code per: https://github.com/ruby/ruby/blob/3abbaab1a7a97d18f481164c7dc48749b86d7f39/rational.c#L285-L307 which is basically the wikepedia implementation.

```
inline static long
i_gcd(long x, long y)
{
    unsigned long u, v, t;
    int shift;
```

```
    if (x < 0)
 x = -x;
    if (y < 0)
 y = -y;

    if (x == 0)
 return y;
    if (y == 0)
 return x;

    u = (unsigned long)x;
    v = (unsigned long)y;
    for (shift = 0; ((u | v) & 1) == 0; ++shift) {
 u >>= 1;
 v >>= 1;
    }

    while ((u & 1) == 0)
 u >>= 1;

    do {
 while ((v & 1) == 0)
     v >>= 1;

 if (u > v) {
     t = v;
     v = u;
     u = t;
 }
 v = v - u;
    } while (v != 0);

    return (long)(u << shift);
}
```

This is the fastest implementation from the benchmarks. (I originally, wrongly, cited the implementation in the article, which is 4|5th fastest in benchmarks, but still almost 2x faster than the Ruby implementation.)

```
// based on wikipedia's article,
// fixed by D. Lemire,  K. Willets
unsigned int gcdwikipedia7fast32(unsigned int u, unsigned int v)
{
     int shift, uz, vz;
     if ( u == 0) return v;
     if ( v == 0) return u;
     uz = __builtin_ctz(u);
     vz = __builtin_ctz(v);
     shift = uz > vz ? vz : uz;
     u >>= uz;
     do {
       v >>= vz;
       int diff = v;
       diff -= u;
       if ( diff == 0 ) break;
       vz = __builtin_ctz(diff);
       if ( v <  u ) u = v;
       v = abs(diff);

     } while( 1 );
     return u << shift;
}
```

The key to speeding up all the algorithms is using the __builtin_ctz(x) directive to determine the number of trailing binary '0's.

History

#1 - 09/26/2018 06:34 PM - jzakiya (Jabari Zakiya)

- Tracker changed from Bug to Feature

- Backport deleted (2.3: UNKNOWN, 2.4: UNKNOWN, 2.5: UNKNOWN)

#2 - 09/26/2018 11:33 PM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to watson1978 (Shizuo Fujita)

Thanks. Assigned to @watson1978 (Shizuo Fujita).

It would be very helpful if you could provide us a patch and perform the benchmark with Ruby implementation, not a toy benchmark program. Note that __builtin_ctzl is not available on some compilers. You need to check if it is available or not.

#3 - 09/27/2018 10:37 PM - jzakiya (Jabari Zakiya)

Hi

I just submitted this issue feature request:

https://bugs.ruby-lang.org/issues/15172

to deal with the issue of using (or not) the __builtin_ctz compiler directive.

I implemented code that mimicked it that also greatly increases the ruby gcd performance.
I included the new code and benchmarks to the gist I previously linked.

https://gist.github.com/jzakiya/44eae4feeda8f6b048e19ff41a0c6566

```
[jzakiya@jabari-pc ~]$ ./gcd2
gcd between numbers in [1 and 2000]

gcdwikipedia7fast32   :  time = 73
gcdwikipedia4fast     :  time = 113
gcdFranke             :  time = 133
gcdwikipedia3fast     :  time = 139
gcdwikipedia2fastswap :  time = 162
gcdwikipedia5fast     :  time = 140
gcdwikipedia7fast     :  time = 129
gcdwikipedia2fast     :  time = 161
gcdwikipedia6fastxchg :  time = 145
gcdwikipedia2fastxchg :  time = 168
gcd_iterative_mod     :  time = 230
gcd_recursive         :  time = 232
basicgcd              :  time = 234
rubygcd               :  time = 305
gcdwikipedia2         :  time = 312
gcdwikipedia7fast32_a :  time = 129
gcdwikipedia4fast_a   :  time = 149
rubygcd_a             :  time = 193
rubygcd_b             :  time = 169
gcd between numbers in [1000000001 and 1000002000]

gcdwikipedia7fast32   :  time = 76
gcdwikipedia4fast     :  time = 106
gcdFranke             :  time = 121
gcdwikipedia3fast     :  time = 127
gcdwikipedia2fastswap :  time = 153
gcdwikipedia5fast     :  time = 126
gcdwikipedia7fast     :  time = 118
gcdwikipedia2fast     :  time = 148
gcdwikipedia6fastxchg :  time = 134
gcdwikipedia2fastxchg :  time = 154
gcd_iterative_mod     :  time = 215
gcd_recursive         :  time = 214
basicgcd              :  time = 220
rubygcd               :  time = 287
gcdwikipedia2         :  time = 289
gcdwikipedia7fast32_a :  time = 116
gcdwikipedia4fast_a   :  time = 142
```

```
rubygcd_a             :  time = 180
rubygcd_b             :  time = 155
```

Note using the __builtin_ctz mimicking code, instead of the directive itself,
still makes the gcdwikipedia7fast32_a the third fastest version, and obviously
the preferred implementation if not using __builtin_ctz.

I present this in asking you how you want me to proceed, because I don't really know
C code and how to do PRs to Ruby. If you can lay out a detailed process for me to do
that maybe I can assess what is in my capacity to do.

At minimum, the code for rubygcd_a could|can be incorporated into the codebase
without dealing right now with the __builtin_ctz directive issue.

#4 - 09/28/2018 12:48 AM - mame (Yusuke Endoh)

No, no. You can just use __builtin_ctzl when it is available. All you need is check if it is available or not, and keep the original code for the case where __builtin_ctzl is unavailable. Gcc and clang provide it, so it is actually available in almost all cases. Even if __builtin_ctzl is unavailable, it should still build and work, but the performance does not matter, I think.

Ruby is already using __builtin_ctz and __builtin_ctzll. See configure.ac and internal.h.

#5 - 12/28/2018 06:23 PM - ahorek (Pavel Rosický)

your micro-benchmarks aren't always fair, because some algorithms don't handle all edge cases, different data types etc.

for jruby I choose a different algorithm that is slightly slower than the fastest gcdwikipedia7fast32 (~15%) but in my opinion more readable.
https://github.com/jruby/jruby/blob/1d0c3d643a6841f388e646678ee243bff571450c/core/src/main/java/org/jruby/util/Numeric.java#L512

here's the PR (gcdwikipedia7fast32 + minor changes) https://github.com/ruby/ruby/pull/2060

and some ruby numbers (benchmark https://github.com/ruby/ruby/pull/1596)

all variants tested on AMD FX 8300 8C and gcc version 8.1.0 (Ubuntu 8.1.0-5ubuntu1~14.04)
ruby 2.7.0dev (2018-12-28 trunk 66617) [x86_64-linux]

```
          Time#subsec      2.969M (± 9.6%) i/s -     14.733M in   5.010950s
               Time#-      5.716M (±11.4%) i/s -     28.103M in   5.000934s
           Time#round    400.712k (±11.9%) i/s -      1.992M in   5.046665s
            Time#to_f      6.422M (±10.5%) i/s -     31.613M in   4.999488s
            Time#to_r      2.251M (±10.4%) i/s -     11.124M in   5.002516s
          Rational#+       5.377M (±10.1%) i/s -     26.577M in   5.001636s
          Rational#-       5.542M (± 9.5%) i/s -     27.419M in   5.001546s
          Rational#*       6.341M (± 9.5%) i/s -     31.390M in   5.002212s
                 gcd       6.922M (± 9.0%) i/s -     34.285M in   5.001389s
```

trunk + new gcd

```
          Time#subsec      3.348M (± 8.9%) i/s -     16.592M in   4.999620s / 1.13
               Time#-      5.840M (±11.6%) i/s -     28.728M in   5.000946s / 1.02
           Time#round    468.770k (±12.5%) i/s -      2.319M in   5.028050s / 1.17
            Time#to_f      6.713M (± 9.8%) i/s -     33.214M in   4.999639s / 1.05
            Time#to_r      3.191M (± 7.9%) i/s -     15.884M in   5.010305s / 1.42
          Rational#+       5.893M (±10.6%) i/s -     29.082M in   4.999884s / 1.10
          Rational#-       6.183M (±11.2%) i/s -     30.443M in   4.999746s / 1.12
          Rational#*       7.069M (±10.5%) i/s -     34.922M in   5.001804s / 1.11
                 gcd       9.742M (±10.4%) i/s -     48.159M in   5.007085s / 1.40
```

trunk + new gcd without __builtin_ctz support

```
          Time#subsec      2.699M (± 8.9%) i/s -     13.385M in   5.002527s / 0.89
               Time#-      5.734M (±10.6%) i/s -     28.224M in   5.002541s / 1.00
           Time#round    392.314k (±13.8%) i/s -      1.926M in   5.012040s / 0.98
            Time#to_f      6.725M (±10.5%) i/s -     33.163M in   4.999346s / 1.04
            Time#to_r      2.366M (± 9.1%) i/s -     11.705M in   5.004491s / 1.05
          Rational#+       5.429M (±10.1%) i/s -     26.851M in   5.006358s / 1.01
          Rational#-       5.544M (± 9.8%) i/s -     27.430M in   5.002418s / 0.98
          Rational#*       6.225M (±10.7%) i/s -     30.833M in   5.018386s / 0.98
                 gcd       7.001M (± 7.1%) i/s -     34.855M in   5.006972s / 1.01
```

alternative implementations
jruby 9.2.6.0-SNAPSHOT (2.5.3) 2018-12-27 e51a3e4 Java HotSpot(TM) 64-Bit Server VM 11.0.1+13-LTS on 11.0.1+13-LTS +jit [linux-x86_64]

```
          Time#subsec      5.018M (± 6.3%) i/s -     24.866M in   4.979170s
               Time#-      7.868M (± 5.6%) i/s -     39.066M in   4.985576s
```

```
          Time#round      3.461M (± 8.1%) i/s -     17.138M in   4.998527s
           Time#to_f      8.198M (± 5.2%) i/s -     40.775M in   4.990224s
           Time#to_r      4.789M (± 6.9%) i/s -     23.777M in   4.992261s
         Rational#+       5.217M (± 6.3%) i/s -     25.944M in   4.995694s
         Rational#-       5.701M (± 7.4%) i/s -     28.329M in   4.998743s
         Rational#*       6.290M (± 6.7%) i/s -     31.283M in   4.997365s
                gcd       7.376M (± 7.2%) i/s -     36.625M in   4.995073s
```

truffleruby 1.0.0-rc10, like ruby 2.4.4, GraalVM CE Native [x86_64-linux]

```
        Time#subsec      3.541M (±67.8%) i/s -     13.706M in   4.986699s
             Time#-      8.279M (± 9.4%) i/s -     38.671M in   4.984896s
          Time#round    311.696k (±43.3%) i/s -    502.226k in   4.991276s
           Time#to_f     16.719M (± 9.2%) i/s -     75.067M in   4.981367s
           Time#to_r      1.386M (±21.2%) i/s -      5.045M in   4.993055s
         Rational#+       7.332M (±14.7%) i/s -     28.100M in   4.982371s
         Rational#-       7.354M (±24.3%) i/s -     22.682M in   4.992218s
         Rational#*       7.340M (±19.3%) i/s -     28.534M in   5.003816s
                gcd      68.576M (± 4.7%) i/s -    326.812M in   4.908116s
```

as you can see Time#to_r and Integer#gcd is about 40% faster which is the best case scenario even when in your micro-benchmark it was 300% faster.

using the new algorithm without __builtin_ctz introduced some perf regressions, but they're within margin of error

I don't think this change will have some impact on real application's performance, all of these cases are just micro-benchmarks...

#6 - 12/28/2018 06:33 PM - ahorek (Pavel Rosický)

- File rational.c.patch added

Files

rational.c.patch	1.22 KB	12/28/2018	ahorek (Pavel Rosický)