Ruby trunk - Feature #15172

Performance: create method(s) to mimic __builtin_ctz compiler directive functionality

09/27/2018 10:08 PM - jzakiya (Jabari Zakiya)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Background

This proposal emanates from issues I raised with regard to speeding up Ruby's implementation of it's gcd.

https://bugs.ruby-lang.org/issues/15166

The use case for these proposed methods exists for many mathematical|numerical
problems, but other too. Using these methods within the Ruby codebase alone
will also help facilitate Ruby's 3x3 goals, in addition to other improvements.

The compiler directive __builtin_ctz (count [of] trailing zeroes)
speeds up all the algorithms because it translates into 1 (a minimal) number
of machine instructions to perform this function.

https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

https://stackoverflow.com/questions/13517232/gcc-builtin-functions

It's possible to mimic it to achieve better comparable performance. The above referenced algorithms use __builtin_ctz for two distinct purposes.

1) To find the specific count of trailing bits of a value.

vz = __builtin_ctz(v);

2) To reduce a value by its count of trailing bits.

v >>= __builtin_ctz(v);

The equivalent Ruby code used is: while ((v & 1) == 0) v >>= 1;

This is inefficient as it processes only 1 bit per iteration, but can be improved by doing 2 bits per iteration.

```
while ((v & 0b11) == 0) v >>= 2;
if ((v & 1) == 0) v >>= 1;
```

Using more bits improves performance more. For this I created an array ctz_bits[] which provides the count of trailing zeroes for a given value.

First I tried 3 bits per iteration.

```
int ctz_bits[8] = {0, 0, 1, 0, 2, 0, 1, 0};

   while ((u & 0b111) == 0) u >>= 3;
   u >>= ctz_bits[u & 0b111];
```

Then 4 bits, with better performance,

```
int ctz_bits[16] = {0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0};

   while ((u & 0xf) == 0) u >>= 4;
   u >>= ctz_bits[u & 0xf];
```

Then 5 bits, with even better performance (notice the pattern here).

```
int ctz_bits[32] = {0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0};

   while ((u & 0x1f) == 0) u >>= 5;
   u >>= ctz_bits[u & 0x1f];
```

I settled on using 8 bits which is a 256 element array.

```
int ctz_bits[256] = {0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0};

   while ((u & 0xff) == 0) u >>= 8;
   u >>= ctz_bits[u & 0xff];
```

This can be standardized to accommodate any bit size as follows.

```
int ctz_shift_bits = 8;
int ctz_mask       = 255;  // 2**ctz_shift_bits - 1
int ctz_bits[256] = {0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0};

   while ((u & ctz_mask) == 0) u >>= ctz_shift_bits;
   u >>= ctz_bits[u & ctz_mask];
```

By sacrificing a little bit of speed (maybe?) we can compress the ctz_bits[] array. First we can eliminate odd index values, which all return '0' shift bits (lsb is '1'), into a single 8 element array. (This is conceptual code, that could accommodate any bit size).

```
int ctz_shift_bits = x;
int ctz_mask       = 2**ctz_shift_bits - 1
int ctz_bits[8] = {0, 1, 2, 1, 3, 1, 2, 1};

   while ((u & ctz_mask) == 0) u >>= ctz_shift_bits;
   if ((u & 1) == 0) {
     u >>= (u % 16) == 0 ? correct_value : ctz_bits[correct_mask];
   }
```

Proposal

Create a method that mimic the function of __builtin_ctz.
These names are just descriptive to show functionality.

```
1)

   x.ctz_val   -- 12.ctz_val => 3;   13.ctz_val => 13``

2)
   x.ctz_bits -- x = 12, x.ctz_bits => 2, x = 12
                 x = 13, x.ctz_bits => 0, x = 13
```

The ideal case is to use -__builtin_ctz, but I understand the concern about it's availability on all compilers. Creating these methods can provide the best of both worlds, by aliasing ctz_bits to __builtin_ctz and choosing which to use at compile time based on availability. Providing either (or both) will definitely increase Ruby's internal performance, and help it reach its 3x3 goal, while providing users with fast and standard methods for these functions.

You can see implementation comparisons with gcd from below.

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

The xx_a versions mimic those using __builtin_ctz.

Below shows the differences in ruby gcd implementations.
The standard lib implementation rubygcd is slowest, ruby_a is 1/3 faster, and ruby_b (using, builtin_ctz) is almost fully 2x faster. They clearly display specific, and possible, performance benefits of this proposals.

```
[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
```

History

#1 - 12/28/2018 05:11 PM - ahorek (Pavel Rosický)

hi, such fallback already exists
https://github.com/ruby/ruby/commit/4cf460a7bb258d3d61414d2f74df4c0f83c6a3af

I don't know if your implementation is faster, but it's certanly not worth to optimize for platforms without __builtin_ctz support.