gmp

Ruby bindings to the GMP library
Edition 0.5.3
20 September 2010

written by Sam Rawlins
with extensive quoting from the GMP Manual

This manual describes how to use the gmp Ruby gem, which provides bindings to the GNU multiple precision arithmetic library, version 4.3.x or 5.0.x.

Contents

1 Introduction to GNU MP

This entire page is copied verbatim from the GMP Manual.

GNU MP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types.

Many applications use just a few hundred bits of precision; but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum.

The speed of GMP is achieved by using fullwords as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs, and by a general emphasis on speed (as opposed to simplicity or elegance).

There is assembly code for these CPUs: ARM, DEC Alpha 21064, 21164, and 21264, AMD 29000, AMD K6, K6-2, Athlon, and Athlon64, Hitachi SuperH and SH-2, HPPA 1.0, 1.1, and 2.0, Intel Pentium, Pentium Pro/II/III, Pentium 4, generic x86, Intel IA-64, i960, Motorola MC68000, MC68020, MC88100, and MC88110, Motorola/IBM PowerPC 32 and 64, National NS32000, IBM POWER, MIPS R3000, R4000, SPARCv7, SuperSPARC, generic SPARCv8, UltraSPARC, DEC VAX, and Zilog Z8000. Some optimizations also for Cray vector systems, Clipper, IBM ROMP (RT), and Pyramid AP/XP.

For up-to-date information on GMP, please see the GMP web pages at `http://gmplib.org/`

The latest version of the library is available at `ftp://ftp.gnu.org/gnu/gmp/`

Many sites around the world mirror '`ftp.gnu.org`', please use a mirror near you, see `http://www.gnu.org/order/ftp.html` for a full list.

There are three public mailing lists of interest. One for release announcements, one for general questions and discussions about usage of the GMP library, and one for bug reports. For more information, see `http://gmplib.org/mailman/listinfo/`.

The proper place for bug reports is gmp-bugs@gmplib.org. See Chapter 4 [Reporting Bugs], page 28 for information about reporting bugs.

2 Introduction to MPFR

The gmp gem optionally interacts with the MPFR library as well. This entire page is copied verbatim from the MPFR manual.

The MPFR library is a C library for multiple-precision floating-point computations with correct rounding. MPFR has continuously been supported by the INRIA and the current main authors come from the Caramel and Arnaire project-teams at Loria (Nancy, France) and LIP (Lyon, France) respectively; see more on the credit page. MPFR is based on the GMP multiple-precision library.

The main goal of MPFR is to provide a library for multiple-precision floating-point computation which is both efficient and has a well-defined semantics. It copies the good ideas from the ANSI/IEEE-754 standard for double-precision floating-point arithmetic (53-bit mantissa).

MPFR is free. It is distributed under the GNU Lesser General Public License (GNU Lesser GPL), version 3 or later (2.1 or later for MPFR versions until 2.4.x). The library has been registered in France by the Agence de Protection des Programmes under the number IDDN FR 001 120020 00 R P 2000 000 10800, on 15 March 2000. This license guarantees your freedom to share and change MPFR, to make sure MPFR is free for all its users. Unlike the ordinary General Public License, the Lesser GPL enables developers of non-free programs to use MPFR in their programs. If you have written a new function for MPFR or improved an existing one, please share your work!

3 Introduction to the gmp gem

The gmp Ruby gem is a Ruby library that provides bindings to GMP. The gem is incomplete, and will likely only include a subset of the GMP functions. It is built as a C extension for Ruby, interacting with gmp.h. The gmp gem is not endorsed or supported by GNU or the GMP team (or MPFR team). The gmp gem also does not ship with GMP (or MPFR), so GMP (and MPFR) must be compiled separately.

4 Installing the gmp gem

4.1 Prerequisites

OK. First, we've got a few requirements. To install the gmp gem, you need one of the following versions of Ruby:

- (MRI) Ruby 1.8.6 - tested lightly.
- (MRI) Ruby 1.8.7 - tested seriously.
- (MRI) Ruby 1.9.1 - tested seriously.
- (MRI) Ruby 1.9.2 - tested seriously.

As you can see only Matz's Ruby Interpreter (MRI) is supported. I haven't even put a thought into trying other interpreters/VMs. I intend to look into FFI, which supposedly will allow me to load this extension into JRuby and Rubinius, not sure about others...

Next is the platform, the combination of the architecture (processor) and OS. As far as I can tell, if you can compile GMP and Ruby (and optionally MPFR) on a given platform, you can use the gmp gem there too. Please report problems with that hypothesis.

Lastly, GMP (and MPFR). GMP (and MPFR) must be compiled and working. "And working" means you ran "make check" after compiling GMP (and MPFR), and it 'check's out. The following versions of GMP (and MPFR) have been tested:

- GMP 4.3.1 (with MPFR 2.4.2)
- GMP 4.3.2 (with MPFR 2.4.2 and 3.0.0)
- GMP 5.0.0 (with MPFR 3.0.0)
- GMP 5.0.1 (with MPFR 3.0.0)

That's all. I don't intend to test any older versions.

Here is a table of the exact environments on which I have tested the gmp gem. The (MPFR) version denotes that the gmp gem was tested both with and without the given version of MPFR:

Platform	Ruby	GMP	(MPFR)
Linux (Ubuntu NR 10.04) on x86 (32-bit)	(MRI) Ruby 1.8.7	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.8.7	GMP 5.0.1	(3.0.0)
	(MRI) Ruby 1.9.1	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.9.1	GMP 5.0.1	(3.0.0)
	(MRI) Ruby 1.9.2	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.9.2	GMP 5.0.1	(3.0.0)
Linux (Ubuntu 10.04) on x86_64 (64-bit)	(MRI) Ruby 1.8.7	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.8.7	GMP 5.0.1	(3.0.0)
	(MRI) Ruby 1.9.1	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.9.1	GMP 5.0.1	(3.0.0)
	(MRI) Ruby 1.9.2	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.9.2	GMP 5.0.1	(3.0.0)
Mac OS X 10.6.4 on x86_64 (64-bit)	(MRI) Ruby 1.8.7	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.8.7	GMP 5.0.1	(3.0.0)
	(MRI) Ruby 1.9.1	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.9.1	GMP 5.0.1	(3.0.0)
	(MRI) Ruby 1.9.2	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.9.2	GMP 5.0.1	(3.0.0)
Windows XP on x86 (32-bit)	(MRI) Ruby 1.9.1	GMP 4.3.2	(2.4.2)
	(MRI) Ruby 1.9.1	GMP 5.0.1	(3.0.0)

In addition, I *used to test* on the following environments, in versions 0.4.7 and earlier of the gmp gem:

Platform	Ruby	GMP
Cygwin on x86	(MRI) Ruby 1.8.7	GMP 4.3.1
Linux (LinuxMint 7) on x86	(MRI) Ruby 1.8.7	GMP 4.3.1
Mac OS X 10.5.7 on x86 (32-bit)	(MRI) Ruby 1.8.6	GMP 4.3.1
Mac OS X 10.5.7 on x86 (32-bit)	(MRI) Ruby 1.9.1	GMP 4.3.1

4.2 Installing

You may clone the gmp gem's git repository with:

```
git clone git://github.com/srawlins/gmp.git
```

Or you may install the gem from gemcutter (rubygems.org):

```
gem install gmp
```

At this time, the gem self-compiles. If required libraries cannot be found, you may compile the C extensions manually with:

```
cd <gmp gem directory>/ext
ruby extconf.rb
make
```

There shouldn't be any errors, or warnings.

5 Testing the gmp gem

Testing the gmp gem is quite simple. The test/unit_tests.rb suite uses Unit::Test. You can run this test suite with:

```
cd <gmp gem directory>/test
ruby unit_tests.rb
```

All tests should pass. If you don't have the test-unit gem installed, then you may run into one error. It would look like:

```
1) Error:
test_z_div(TC_division):
TypeError: GMP::Q can't be coerced into Float
C:/Ruby191/devkit/msys/1.0.11/projects/gmp_gem/test/tc_division.rb:18:in `test_z_div'
```

6 GMP and gmp gem basics

6.1 Classes

The gmp gem includes the namespace `GMP` and four classes within `GMP`:

- `GMP::Z` - Methods for signed integer arithmetic. There are about 64 methods here.
- `GMP::Q` - Methods for rational number arithmetic. There are at least 11 methods here (still accounting).
- `GMP::F` - Methods for floating-point arithmetic. There are at least 6 methods here (still accounting).
- `GMP::RandState` - Methods for random number generation. There are 3 methods here.

In addition to the above four classes, there are also four constants within `GMP`:

- `GMP::GMP_VERSION` - The version of GMP linked into the gmp gem
- `GMP::GMP_CC` - The compiler that compiled GMP linked into the gmp gem
- `GMP::GMP_CFLAGS` - The compiler flags used to compile GMP linked into the gmp gem
- `GMP::GMP_BITS_PER_LIMB` - The number of bits per limb

7 MPFR basics

The gmp gem can optionally link to MPFR, the Multiple Precision Floating-Point Reliable Library. The x86-mswin32 version of the gmp gem comes with MPFR. This library uses the floating-point type from GMP, and thus the MPFR functions mapped in the gmp gem become methods in `GMP::F`.
There are additional constants within `GMP` when MPFR is linked:

- `GMP::MPFR_VERSION` - The version of MPFR linked into the gmp gem.
- `GMP::GMP_RNDN` - Rounding mode representing "round to nearest."
- `GMP::GMP_RNDZ` - Rounding mode representing "round toward zero."
- `GMP::GMP_RNDU` - Rounding mode representing "round toward positive infinity."
- `GMP::GMP_RNDD` - Rounding mode representing "round toward negative infinity."
- `GMP::MPFR_RNDN` - Rounding mode representing "round to nearest."
  (MPFR version 3.0.0 or higher only)
- `GMP::MPFR_RNDZ` - Rounding mode representing "round toward zero."
  (MPFR version 3.0.0 or higher only)
- `GMP::MPFR_RNDU` - Rounding mode representing "round toward positive infinity."
  (MPFR version 3.0.0 or higher only)
- `GMP::MPFR_RNDD` - Rounding mode representing "round toward negative infinity."
  (MPFR version 3.0.0 or higher only)
- `GMP::MPFR_RNDZ` - Rounding mode representing "round away from zero."
  (MPFR version 3.0.0 or higher only)

8 Integer Functions

8.1 Initializing, Assigning Integers

new

GMP::Z.new → *integer*
GMP::Z.new(*numeric = 0*) → *integer*
GMP::Z.new(*str*) → *integer*

This method creates a new *GMP::Z* integer. It takes one optional argument for the value of the integer. This argument can be one of several classes. Here are some examples:

```
GMP::Z.new                 #=> 0 (default)
GMP::Z.new(1)              #=> 1 (Ruby Fixnum)
GMP::Z.new("127")          #=> 127 (Ruby String)
GMP::Z.new(4294967296)     #=> 4294967296 (Ruby Bignum)
GMP::Z.new(GMP::Z.new(31)) #=> 31 (GMP Integer)
```

There is also a convenience method available, `GMP::Z()`.

8.2 Converting Integers

to_d

integer.to_d → *float*

Returns *integer* as an Float if *integer* fits in a Float.
Otherwise returns the least significant part of `integer`, with the same sign as *integer*.
If *integer* is too big to fit in a Float, the returned result is probably not very useful. To find out if the value will fit, use the function *mpz_fits_slong_p* (**Unimplemented**).

to_i

integer.to_i → *fixnum*

Returns *integer* as a Fixnum if *integer* fits in a Fixnum.

Otherwise returns the least significant part of *integer*, with the same sign as *integer*.

If *integer* is too big to fit in a *Fixnum*, the returned result is probably not very useful. To find out if the value will fit, use the function *mpz_fits_slong_p* (**Unimplemented**).

to_s *integer*.to_s(*base = 10*) → *str*

Converts *integer* to a string of digits in base *base*. The *base* argument may vary from 2 to 62 or from -2 to -36, or be a symbol, one of *:bin*, *:oct*, *:dec*, or *:hex*.

For *base* in the range 2..36, digits and lower-case letters are used; for -2..-36 (and *:bin*, *:oct*, *:dec*, and *:hex*), digits and upper-case letters are used; for 37..62, digits, upper-case letters, and lower-case letters (in that significance order) are used. Here are some examples:

```
GMP::Z(1).to_s          #=> "1"
GMP::Z(32).to_s(2)      #=> "100000"
GMP::Z(32).to_s(4)      #=> "200"
GMP::Z(10).to_s(16)     #=> "a"
GMP::Z(10).to_s(-16)    #=> "A"
GMP::Z(255).to_s(:bin) #=> "11111111"
GMP::Z(255).to_s(:oct) #=> "377"
GMP::Z(255).to_s(:dec) #=> "255"
GMP::Z(255).to_s(:hex) #=> "ff"
```

8.3 Integer Arithmetic

+ *integer* + *numeric* → *numeric*

Returns the sum of *integer* and *numeric*. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *GMP::Q*, *GMP::F*, or *Bignum*.

add! *integer*.add!(*numeric*) → *numeric*

Sums *integer* and *numeric*, in place. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *GMP::Q*, *GMP::F*, or *Bignum*.

-

integer - *numeric* → *numeric*
integer.sub!(*numeric*) → *numeric*

Returns the difference of *integer* and *numeric*. The destructive method calculates the difference in place. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *GMP::Q*, *GMP::F*, or *Bignum*. Here are some examples:

```
seven = GMP::Z(7)
nine  = GMP::Z(9)
half  = GMP::Q(1,2)
pi    = GMP::F("3.14")
nine - 5       #=> 4 (GMP Integer)
nine - seven   #=> 2 (GMP Integer)
nine - (2**32) #=> -4294967287 (GMP Integer)
nine - nine    #=> 0 (GMP Integer)
nine - half    #=> 8.5 (GMP Rational)
nine - pi      #=> 5.86 (GMP Float)
```

*****

integer * *numeric* → *numeric*
integer.mul(*numeric*) → *numeric*
integer.mul!(*numeric*) → *numeric*

Returns the product of *integer* and *numeric*. The destructive method calculates the product in place. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *GMP::Q*, *GMP::F*, or *Bignum*.

addmul!

integer.addmul!(*a*, *b*) → *numeric*

Sets *integer* to the sum of *integer* and the product of *a* and *b*. This destructive method calculates the result in place. Both *a* and *b* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

<<

integer <<*numeric* → *integer*

Returns *integer* times 2 to the *numeric* power. This can also be defined as a left shift by *numeric* bits.

-@

-*integer*
integer.neg
integer.neg!

Returns the negation, the additive inverse, of *integer*. The destructive method negates in place.

abs *integer*.abs
integer.abs!

Returns the absolute value of *integer*. The destructive method calculates the absolute value in place.

8.4 Integer Division

tdiv *integer*.tdiv(*numeric*) → *integer*

Returns the division of *integer* by *numeric*, truncated. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *Bignum*. The return object's class is always *GMP::Z*.

fdiv *integer*.fdiv(*numeric*) → *integer*

Returns the division of *integer* by *numeric*, floored. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *Bignum*. The return object's class is always *GMP::Z*.

cdiv *integer*.cdiv(*numeric*) → *integer*

Returns the ceiling division of *integer* by *numeric*. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *Bignum*. The return object's class is always *GMP::Z*.

tmod *integer*.tmod(*numeric*) → *integer*

Returns the remainder after truncated division of *integer* by *numeric*. *numeric* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*. The return object's class is always *GMP::Z*.

fmod *integer*.fmod(*numeric*) → *integer*

Returns the remainder after floored division of *integer* by *numeric*. *numeric* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*. The return object's class is always *GMP::Z*.

cmod *integer*.cmod(*numeric*) → *integer*

Returns the remainder after ceilinged division of *integer* by *numeric*. *numeric* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*. The return object's class is always *GMP::Z*.

% *integer* % *numeric* → *integer*

Returns *integer* modulo *numeric*. *numeric* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*. The return object's class is always *GMP::Z*.

8.5 Integer Exponentiation

******

integer ** *numeric* → *numeric*
integer.pow(*numeric*) → *numeric*
GMP::Z.pow(*integer*, *numeric*) → *numeric*

Returns *integer* raised to the *numeric* power. In the singleton method (*GMP::Z.pow()*), *integer* can be either a *GMP::Z*, *Fixnum*, *Bignum*, or *String*.

powmod

integer.powmod(*exp*, *mod*) → *integer*

Returns *integer* raised to the *exp* power, modulo *mod*. Negative *exp* is supported if an inverse, $integer^{-1}$ modulo *mod*, exists. If an inverse doesn't exist then a divide by zero exception is raised.

8.6 Integer Roots

root

integer.root(*numeric*) → *numeric*

Returns the integer part of the *numeric*'th root of *integer*.

sqrt

integer.sqrt → *numeric*
integer.sqrt! → *numeric*

Returns the truncated integer part of the square root of *integer*.

sqrtrem

integer.sqrtrem → *sqrt*, *rem*

Returns the truncated integer part of the square root of *integer* as *sqrt* and the remainder, $integer - sqrt * sqrt$, as *rem*, which will be zero if *integer* is a perfect square.

power?

integer.power? → *true* | *false*

Returns true if *integer* is a perfect power, i.e., if there exist integers a and b, with $b > 1$, such that *integer* equals a raised to the power b.

Under this definition both 0 and 1 are considered to be perfect powers. Negative values of integers are accepted, but of course can only be odd perfect powers.

square?

integer.square? → *true* | *false*

Returns true if *integer* is a perfect square, i.e., if the square root of *integer* is an integer. Under this definition both 0 and 1 are considered to be perfect squares.

8.7 Number Theoretic Functions

is_probab_prime? — *integer*.is_probab_prime?(*reps* = 5) → 0, 1, or 2

Determine whether *integer* is prime. Returns 2 if *integer* is definitely prime, returns 1 if *integer* is probably prime (without being certain), or returns 0 if *integer* is definitely composite.

This function does some trial divisions, then some Miller-Rabin probabilistic primality tests. *reps* controls how many such tests are done, 5 to 10 is a reasonable number, more will reduce the chances of a composite being returned as probably prime.

Miller-Rabin and similar tests can be more properly called compositeness tests. Numbers which fail are known to be composite but those which pass might be prime or might be composite. Only a few composites pass, hence those which pass are considered probably prime.

next_prime

integer.next_prime → *prime*
integer.nextprime → *prime*
integer.next_prime! → *prime*
integer.nextprime! → *prime*

Returns the next prime greater than *integer*. The destructive method sets *integer* to the next prime greater than *integer*.

This function uses a probabilistic algorithm to identify primes. For practical purposes it's adequate, the chance of a composite passing will be extremely small.

gcd — *a*.gcd(*b*) → *g*

Computes the greatest common divisor of *a* and *b*. *g* will always be positive, even if *a* or *b* is negative. *b* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

```
GMP::Z(24).gcd(GMP::Z(8))   #=> GMP::Z(8)
GMP::Z(24).gcd(8)   #=> GMP::Z(8)
GMP::Z(24).gcd(2**32)   #=> GMP::Z(8)
```

invert — *a*.invert(*m*) → *integer*

Computes the inverse of *a* mod *m*. *m* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

```
GMP::Z(2).invert(GMP::Z(11))   #=> GMP::Z(6)
GMP::Z(3).invert(11)   #=> GMP::Z(4)
GMP::Z(5).invert(11)   #=> GMP::Z(9)
```

jacobi — a.jacobi(b) $\rightarrow$ *integer*
GMP::Z.jacobi(a, b) $\rightarrow$ *integer*

Returns the Jacobi symbol (a/b). This is defined only for b odd. If b is even, a range exception will be raised.

GMP::Z.jacobi (the instance method) requires b to be an instance of *GMP::Z*. *GMP::Z#jacobi* (the class method) requires a and b each to be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

legendre — a.legendre(b) $\rightarrow$ *integer*

Returns the Legendre symbol (a/b). This is defined only for p an odd positive prime. If p is even, negative, or composite, a range exception will be raised.

remove — n.remove($factor$) $\rightarrow$ (*integer*, *times*)

Remove all occurrences of the factor $factor$ from n. $factor$ can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*. *integer* is the resulting integer, an instance of *GMP::Z*. *times* is how many times $factor$ was removed, a $Fixnum$.

fac — GMP::Z.fac(n) $\rightarrow$ *integer*

Returns $n!$, or, n factorial.

fib — GMP::Z.fib(n) $\rightarrow$ *integer*

Returns $F[n]$, or, the nth Fibonacci number.

8.8 Integer Comparisons

<=> — a <=> b $\rightarrow$ *fixnum*

Returns a negative Fixnum if a is less than b.
Returns 0 if a is equal to b.
Returns a positive Fixnum if a is greater than b.

< — $a < b \rightarrow$ *boolean*

Returns true if a is less than b.

<= — a <= b $\rightarrow$ *boolean*

Returns true if a is less than or equal to b.

== — a == b $\rightarrow$ *boolean*

Returns true if a is equal to b.

>=	$a >= b \rightarrow boolean$
	Returns true if a is greater than or equal to b.

>	$a > b \rightarrow boolean$
	Returns true if a is greater than b.

cmpabs	a.cmpabs(b) $\rightarrow$ *fixnum*
	Returns a negative Fixnum if abs(a) is less than abs(b). Returns 0 if abs(a) is equal to abs(b). Returns a positive Fixnum if abs(a) is greater than abs(b).

sgn	a.sgn $\rightarrow -1$, 0, or 1
	Returns -1 if a is less than b. Returns 0 if a is equal to b. Returns 1 if a is greater than b.

eql?	a.eql?(b) $\rightarrow$ *boolean*
	Used when comparing objects as Hash keys.

hash	a.hash $\rightarrow$ *string*
	Used when comparing objects as *Hash* keys.

8.9 Integer Logic and Bit Fiddling

and	a & $b \rightarrow$ *integer*
	Returns *integer*, the bitwise and of a and b.

ior	$a \mid b \rightarrow$ *integer*
	Returns *integer*, the bitwise inclusive or of a and b.

xor	a ^ $b \rightarrow$ *integer*
	Returns *integer*, the bitwise exclusive or of a and b.

com	*integer*.com $\rightarrow$ *complement* *integer*.com! $\rightarrow$ *complement*
	Returns the one's complement of *integer*. The destructive method sets *integer* to the one's complement of *integer*.

popcount n.popcount $\rightarrow$ *fixnum*

If $n >= 0$, return the population count of n, which is the number of 1 bits in the binary representation. If $n < 0$, the number of 1s is infinite, and the return value is the largest possible *mp_bitcnt_t*.

scan0 n.scan0(i) $\rightarrow$ *integer*

Scans n, starting from bit i, towards more significant bits, until the first 0 bit is found. Return the index of the found bit.

If the bit at i is already what's sought, then i is returned.

If there's no bit found, then $INT2FIX(ULONG\_MAX)$ is returned. This will happen in scan0 past the end of a negative number.

scan1 n.scan1(i) $\rightarrow$ *integer*

Scans n, starting from bit i, towards more significant bits, until the first 1 bit is found. Return the index of the found bit.

If the bit at i is already what's sought, then i is returned.

If there's no bit found, then $INT2FIX(ULONG\_MAX)$ is returned. This will happen in scan1 past the end of a negative number.

[] $n[bit\_index] \rightarrow 0$ or 1

Tests bit *bit_index* in n and return 0 or 1 accordingly.

[]= $n[bit\_index]=i \rightarrow nil$

Sets bit *bit_index* in n to i.

8.10 Miscellaneous Integer Functions

odd? n.odd? $\rightarrow$ *boolean*

Returns whether n is odd.

even? n.even? $\rightarrow$ *boolean*

Returns whether n is even.

sizeinbase n.sizeinbase(b) $\rightarrow$ *digits*

Returns the number of digits in base b. b can vary between 2 and 62.

size_in_bin n.size_in_bin $\rightarrow$ *digits*

Returns the number of digits in n's binary representation.

8.11 Integer Special Functions

size *integer*.size $\rightarrow$ *fixnum*

Returns the size of *integer* measured in number of limbs. If *integer* is zero, then the returned value will be zero.

9 Rational Functions

9.1 Initializing, Assigning Rationals

new

GMP::Q.new → *rational*
GMP::Q.new(*numerator = 0*, *denominator = 1*) → *rational*
GMP::Q.new(*str*) → *rational*

This method creates a new *GMP::Q*rational number. It takes two optional arguments for the value of the numerator and denominator. These arguments can each be an instance of several classes. Here are some examples:

```
GMP::Q.new                  #=> 0 (default)
GMP::Q.new(1)               #=> 1 (Ruby Fixnum)
GMP::Q.new(1,3)             #=> 1/3 (Ruby Fixnums)
GMP::Q.new("127")           #=> 127 (Ruby String)
GMP::Q.new(4294967296)      #=> 4294967296 (Ruby Bignum)
GMP::Q.new(GMP::Z.new(31)) #=> 31 (GMP Integer)
```

There is also a convenience method available, `GMP::Q()`.

9.2 Converting Rationals

to_d

rational.to_d → *float*

Returns *rational* as an Float if *rational* fits in a Float.
Otherwise returns the least significant part of `rational`, with the same sign as *rational*.
If *rational* is too big to fit in a Float, the returned result is probably not very useful.

to_s

rational.to_s → *str*

Converts *rational* to a string.

10 Floating-point Functions

10.1 Initializing, Assigning Floats

10.2 Floating-point Special Functions (MPFR Only)

Every method below accepts two additional parameters in addition to any required parameters. These are rnd_mode, the rounding mode to use in calculation, which defaults to *GMP::GMP_RNDN*, and res_prec, the precision of the result, which defaults to the *f.prec*, the precision of f.

log **log2** **log10**	f.log(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.log2(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.log10(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
	Returns the natural log, $\log_2$, and $\log_1 0$ of f, respectively. Returns $-Inf$ if f is -0.

exp **exp2** **exp10**	f.exp(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.exp2(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.exp10(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
	Returns the exponential of f, 2 to the power of f, and 10 to the power of f, respectively.

cos **sin** **tan**	f.cos(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.sin(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.tan(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
	Returns the cosine, sine, and tangent of f, respectively.

sec **csc** **cot**	f.sec(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.csc(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.cot(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
	Returns the secant, cosecant, and cotangent of f, respectively.

acos **asin** **atan**	f.acos(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.asin(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.atan(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
	Returns the arc-cosine, arc-sine, and arc-tangent of f, respectively.

cosh **sinh** **tanh**	f.cosh(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.sinh(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$ f.tanh(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
	Returns the hyperbolic cosine, sine, and tangent of f, respectively.

sech
csch
coth

f.sech(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
f.csch(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
f.coth(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the hyperbolic secant, cosecant, and cotangent of f, respectively.

---

acosh
asinh
atanh

f.acosh(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
f.asinh(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$
f.atanh(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the hyperbolic arc-cosine, arc-sine, and arc-tangent of f, respectively.

---

log1p

f.log1p(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the logarithm of 1 plus f.

---

expm1

f.expm1(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the exponential of f minus 1.

---

eint

f.eint(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the exponential integral of f. For positive f, the exponential integral is the sum of Euler's constant, of the logarithm of f, and of the sum for k from 1 to infinity of f to the power k, divided by k and factorial(k). For negative f, this method returns NaN.

---

li2

f.li2(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the real part of the dilogarithm of f. MPFR defines the dilogarithm as the integral of $-\log(1-t)/t$ from 0 to f.

---

gamma

f.gamma(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the value of the Gamma function on f. When f is a negative integer, this method returns NaN.

---

lngamma

f.lngamma(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the value of the logarithm of the Gamma function on f. When $-2k-1 \leq f \leq -2k$, k being a non-negative integer, this method returns NaN.

---

digamma

f.digamma(rnd_mode = GMP_RNDN, res_prec=f.prec) $\rightarrow g$

Returns the value of the Digamma (sometimes called Psi) function on f. When f is negative, this method returns NaN.

Only available in MPFR version 3.0.0 or later.

zeta	f.zeta(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$
	Returns the value of the Riemann Zeta function on f.

erf **erfc**	f.erf(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$ f.erfc(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$
	Returns the value of the error function on f (respectively the complementary error function on f).

j0 **j1** **jn**	f.j0(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$ f.j1(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$ f.jn(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$
	Returns the value of the first kind Bessel function of order 0 (respectively 1 and n) on f. When f is NaN, this method returns NaN. When f is +Inf or -Inf, this method returns +0. When f is zero, this method returns +Inf or -Inf, depending on the parity and sign of n, and the sign of f.

y0 **y1** **yn**	f.y0(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$ f.y1(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$ f.yn(rnd_mode = GMP_RNDN, res_prec=f.prec) $\to g$
	Returns the value of the second kind Bessel function of order 0 (respectively 1 and n) on f. When f is NaN or negative, this method returns NaN. When f is +Inf, this method returns +0. When f is zero, this method returns +Inf or -Inf, depending on the parity and sign of n.

11 Random Number Functions

11.1 Random State Initialization

new

GMP::RandState.new → *mersenne twister state*
GMP::RandState.new(:default) → *mersenne twister state*
GMP::RandState(:mt) → *mersenne twister random state*
GMP::RandState.new(:lc_2exp, a, c, m2exp) → *linear congruential state*
GMP::RandState.new(:lc_2exp_size, size) → *linear congruential state*

This method creates a new *GMP::RandState*instance. The first argument defaults to *:default* (also *:mt*), which initializes the *GMP::RandState*for a Mersenne Twister algorithm. No other arguments should be given if *:default* or *:mt* is specified.

If the first argument given is *:lc_2exp*, then the *GMP::RandState*is initialized for a linear congruential algorithm. *:lc_2exp* must be followed with a, c, and $m2exp$. The algorithm can then proceed as ($X = (a * X + c) \mod 2^{m2exp}$).

*GMP::RandState*can also be initialized for a linear congruential algorithm with *:lc_2exp_size*. This initializer instead takes just one argument, *size*. a, c, and $m2exp$ are then chosen from a table, with $m2exp/2 > size$. The maximum size currently supported is 128.

```
GMP::RandState.new
GMP::RandState.new(:mt)
GMP::RandState.new(:lc_2exp, 1103515245, 12345, 15)   #=> Perl's
old rand()
GMP::RandState.new(:lc_2exp, 25_214_903_917, 11, 48)   #=> drand48
```

11.2 Random State Seeding

seed

state.seed(*integer*) → *integer*

Set an initial seed value into *state*. *integer* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

11.3 Integer Random Numbers

urandomb

state.urandomb(n) → *integer*

Generates a uniformly distributed random integer in the range 0 to $2^n - 1$, inclusive.

urandomm

state.urandomm(n) → *integer*

Generates a uniformly distributed random integer in the range 0 to $n-1$, inclusive. n can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

12 Benchmarking

Starting with, I believe, GMP 5.0.0, their benchmarking suite was updated to version 0.2, a significant improvement over version 0.1. The suite consists of 3 parts.

First is six individual programs that will benchmark a single concept. These are `multiply`, `divide`, `gcd`, `gcdext`, `rsa`, and `pi`.
Second is gexpr.c, a nifty little program (compiled with `gcc -o gexpr gexpr.c`, or `gcc -o gexpr gexpr.c -lm` on some systems). They use this basically for weighing the results of individual tests and stringifying them to a certain precision.
Third is runbench, a shell script that runs each of the six benchmark programs, each several times with a different set of arguments. It compiles the results, using gexpr to carefully calculate a final "score" for each benchmark, each benchmark family (`rsa` and `pi` are separated as "app" benchmarks.), and for the suite as a whole.

12.1 gmp gem benchmarking

The structure described above lends itself perfectly to some simple modifications that allow me to benchmark the gmp gem. Instead of using the C programs that are the individual benchmarks, I have converted them into Ruby programs, and called them the same names. When runbench executes "multiply" for example, it is running a Ruby program that does the same general thing as "multiply" the C program. This allows me to benchmark the gmp gem, and I can even compare the results with those of benchmarking GMP itself. I have done so below.

12.2 Benchmark Results

I've benchmarked an array of Ruby-version/GMP-version combinations, which can be compared with results of the actual GMPbench benchmark, in order to understand the overhead that Ruby and the Ruby C Extension API impose on the GMP library. The benchmarks listed were all executed on an Apple MacBook "5.1." booting three operating systems with rEFIt 0.14. Because not all of the marks from GMPbench have been ported to the Ruby gmp gem, results for 'base' and 'app' are not listed; they cannot be compared with GMPbench results.

MacBook, Intel Core 2 Duo, 2 GHz, 2 GB DDR3

Mac OS X 10.6.4, x86 (32-bit)
Ruby and GMP compiled with gcc 4.2.1 (Apple Inc. build 5664)

GMP	bench	Ruby 1.8.7	Ruby 1.9.1	Ruby 1.9.2	C
4.3.2	multiply	7696.5	7482.8	7608.2	16789
	divide	7304.3	7148.0	7199.4	9872.5
	gcd	2388.4	2412.1	2470.9	2938.9
	rsa	2042.3	2058.0	2065.9	2136.7
5.0.1	multiply	8142.6	7886.0	7952.7	17925
	divide	10256	10088	10124	16069
	gcd	2498.1	2547.0	2574.2	3063.7
	rsa	2132.4	2157.4	2167.9	2229.2

Ubuntu 10.04, Kernel 2.6.32-24, x86_64 (64-bit)
Ruby and GMP compiled with gcc 4.4.3 (Ubuntu 4.4.3-4ubuntu5)

GMP	bench	Ruby 1.8.7	Ruby 1.9.1	Ruby 1.9.2	C
4.3.2	multiply	7793.7	6825.7	7460.6	14716
	divide	7687.6	7725.9	7473.3	8614.3
	gcd	2450.1	2507.6	2476.6	2779.8
	rsa	1979.1	2065.6	2059.2	1984.3
5.0.1	multiply	9727.1	7020.3	7902.1	17695
	divide	9306.4	10785	10799	15629
	gcd	2345.3	2525.5	2541.2	2925.1
	rsa	2116.5	2117.8	2143.9	2233.4

Windows 7 (64-bit)
Ruby and GMP compiled with gcc 4.5.0 (tdm-1)

GMP	bench	Ruby 1.8.7	Ruby 1.9.1	Ruby 1.9.2	C
4.3.2	multiply	3514.7	3649.4	3631.9	6448.7
	divide	2996.7	3082.0	3065.2	3717.5
	gcd	1142.8	1183.3	1203.6	1359.4
	rsa	719.41	730.05	736.03	757.97
5.0.1	multiply	3702.9	3824.1	3763.8	6835.2
	divide	4549.7	4600.9	4609.2	6497.4
	gcd	1204.9	1233.4	1235.0	1394.2
	rsa	729.68	735.13	728.56	754.72