Improved testing of an Arbitrary-Precision Arithmetic Library using Artificially Small Words

Per Olofsson

Bachelor's Thesis in Computer Science
at Stockholm University, Sweden 2012

Stockholms universitet

NADA

Improved testing of an Arbitrary-Precision Arithmetic Library using Artificially Small Words

Per Olofsson

Bachelor´s Thesis in Computer Science (15 ECTS credits)
Single Subject Courses
Stockholm University year 2012
Supervisor at Nada was Torbjörn Granlund
Examiner was Stefan Arnborg

TRITA-CSC-E 2012:003
ISRN-KTH/CSC/E--12/003--SE
ISSN-1653-5715

Department of Numerical Analysis and Computer Science
Royal Institute of Technology, School of Computer Science and Communication
SE-100 44 Stockholm, Sweden

Abstract

The GNU Multiple Precision Arithmetic Library (GMP) is a software library for making arithmetic computations with numbers of arbitrary size (so-called bignums). GMP is free software and is used by many popular software packages such as Sage, Mathematica, Wolfram Alpha, Maple, and GCC. GMP also includes a comprehensive test suite.

GMP works by splitting large numbers into small pieces (called limbs), where each piece is stored as a full native computer word. We have modified GMP to be able to split numbers into much smaller pieces as a compilation-time option, by using the operator overloading feature of the C++ programming language. The purpose is to make the test suite more effective, increasing the probability of finding potential errors.

The result has been improved source code quality in GMP, as well as the elimination of some errors. The source code modifications done during this project will be included in future releases of GMP, and will be used to test new code.

Referat

Förbättrad testning av ett bibliotek för aritmetik med godtycklig precision genom att använda artificiellt små ord

GNU Multiple Precision Arithmetic Library (GMP) är ett programbibliotek för att göra aritmetiska beräkningar med tal av godtycklig storlek (så kallade bignums). GMP är fri programvara och används av många populära programpaket som exempelvis Sage, Mathematica, Wolfram Alpha, Maple och GCC. GMP innehåller även en omfattande testsvit.

GMP arbetar genom att dela upp stora tal i små delar (som kallas limbar) där varje del lagras som ett helt datorord. Vi har modifierat GMP så att det, med en flagga vid kompilering, kan dela upp tal i mycket mindre delar genom att använda operatoröverlagring i programspråket C++. Syftet är att göra testsviten mer effektiv, vilket ökar sannolikheten för att hitta potentiella fel.

Resultatet har blivit förbättrad kvalitet i GMPs källkod, samt att vissa fel har rättats. Ändringarna som gjordes i källkoden inom ramen för det här projektet kommer att ingå i framtida utgåvor av GMP och användas för att testa ny kod.

Preface

This is my bachelor's thesis in computer science, done at the Department of Numerical Analysis and Computer Science (NADA) at Stockholm University, also part of the School of Computer Science and Communication (CSC) at the Royal Institute of Technology (KTH) in Stockholm.

The degree project was supervised by Torbjörn Granlund at CSC, who also suggested the idea to me. I thank him for his invaluable help during the process.

The degree project was to a large extent done in cooperation with my supervisor; therefore, I generally use the personal pronoun "we" in this thesis. To be clear, however, the artificial limb implementation (reproduced in the appendix) is my own work.

The reader is assumed to be familiar with the C and C++ programming languages. Excellent descriptions of these languages can be found in [6] and [7].

Contents

Chapter 1

Background

1.1 GMP

The GNU Multiple Precision Arithmetic Library (GMP) is a software library for making numerical calculations with arbitrary-size operands, also known as *bignums*. GMP can be used to make calculations involving numbers consisting of hundreds, thousands, and even billions of digits.

GMP is free software, licenced under the GNU Lesser General Public License (LGPL) version 3 [1], and used by many popular software packages such as Sage, Mathematica, Wolfram Alpha, Maple, and GCC. GMP is designed to be as fast as possible, and has been ported to many platforms.

GMP's principal author has been the supervisor of this degree project.

1.2 Limbs

GMP supports arbitrary-size integers and rational numbers, as well as arbitrary-precision floating point numbers. All of these types, however, are based on arbitrary-size natural numbers (non-negative integers), which are the basic building blocks of GMP. Integers are implemented as natural numbers with sign, while rational numbers consist of two arbitrary-size integers (numerator and denominator). Floating point numbers are implemented with the mantissa as an arbitrary-size integer.

GMP stores an arbitrary-size natural number as an array of full computer words, called *limbs*.[1] The size of a limb depends on the processor architecture, but it is typically 32 or 64 bits long, matching the host system's natural word size. If limbs are b bits each, then a number is stored in base 2^b, thus using the entire words. The array is stored with the least significant limb first, as it makes it easier to grow the array to make room for larger numbers (for example, as the result of a computation).

[1] Why are they called limbs? From the GMP manual: "We chose this word because a limb of the human body is analogous to a digit, only larger, and containing several digits". [3]

More precisely, an arbitrary-size natural number k is stored as an array of n limbs $k_0, \ldots, k_{n-1}$ where $n = \lceil \log_{2^b}(k+1) \rceil$ and

$$k = \sum_{i=0}^{n-1} 2^{bi} k_i. \tag{1.1}$$

Example 1.1. Suppose one wants to store the base-16 number $k = \texttt{0x123456789ABC}$ in GMP, on a 32-bit architecture. Since k is 45 bits long, it needs two 32-bit limbs for storage. We put

$$\begin{aligned} k_0 &= \texttt{0x56789ABC}, \\ k_1 &= \texttt{0x00001234}. \end{aligned}$$

Plugging these values into (1.1) yields

$$\begin{aligned} k &= 2^{32\cdot 0} k_0 + 2^{32\cdot 1} k_1 = \\ &= 1 \cdot \texttt{0x56789ABC} + 2^{32} \cdot \texttt{0x00001234} = \\ &= \texttt{0x56789ABC} + \texttt{0x123400000000} = \texttt{0x123456789ABC} \end{aligned}$$

as expected.

1.3 Artificially small limbs

The goal of this degree project was to modify GMP so that it could be run with “artificially” small limbs; artificially in the sense that they are smaller than the machine’s natural word size. Specifically, the goal of this project was to make a large part of GMP run with limbs of b bits for any even number $b \leq 32$, down to at least 8-bit limbs, but if possible also 6 or even 4 bits.

The natural question then, of course, is: why would one want artificially small limbs? The answer is: for the purpose of testing.

GMP comes with an extensive test suite, consisting of 148 test cases. The idea is that, by modifying GMP to use smaller limb sizes, the test suite can become more effective, by increasing the probability of triggering bugs, should they exist.

Why would smaller limb sizes make the test suite more likely to find bugs? While we have no rigorous proof of this, we do have some heuristic arguments:

1. Consider the other extreme: very large limbs. If, for example, one wants to add two numbers which both fit inside one limb, there is not much for GMP to do. It can simply invoke the addition instruction for two words. Conversely, with very small limbs, there is much more to do: GMP will need to perform many limb additions, including the handling of carry propagation. Small limbs increase the “inter-limbness” of the calculations, as one could put it.

2. With 32-bit limbs, there are 2^{32} possible values for each limb. With 8-bit limbs, there are only 256 possible values. If there is a bug in an algorithm

which triggers only on a few specific values, one is much more likely to find them if one generates random 8-bit limbs than if one generates random 32-bit limbs. With 4-bit limbs, one could even perform an exhaustive search on, say, all numbers consisting of five 4-bit limbs. (Of course, this argument assumes that the bug in question can be triggered regardless of limb size.)

In addition, small limbs can be used to test GMP with values consisting of a large number of limbs. For example, say one wants to check if GMP handles numbers consisting of 2^{36} limbs. With 64-bit limbs, 512 gigabytes of memory would be needed, since one limb is 8 bytes. With 8-bit limbs, only 64 gigabytes of memory would be needed (which is much cheaper).

Chapter 2

Analysis

2.1 Considerations

When implementing the artificial limbs, we strived for the following aims:

1. Modify the GMP code itself as little as possible. Ideally, no changes would be needed.

2. If GMP needs to be modified, minimize any special code for artificial limbs. Ideally, exactly the same source code should be compiled whether compiling with or without artificial limbs.

3. It should be easy and feel natural to develop new code in GMP which is compatible with artificial limbs. Ideally, the developer should not need to worry about them (except that the code needs to handle small limb sizes).

To summarise, we wanted the artificial limb implementation to be as unobtrusive as possible. In addition, the second aim is important because we want to test the code that is used in GMP during normal operation, not code that is only used when running with artificial limbs.

2.2 Arithmetic in C

GMP is mostly implemented in the C programming language, with certain performance-critical parts also implemented in assembly language for many processor architectures. We decided to implement artificial limbs only for the C code and disable the assembly parts when building with artificial limbs. The assembly code is highly dependent on the limb size, and probably cannot be generalized without rewriting it.

The C code in GMP uses a type called `mp_limb_t` for limbs. This type is only an alias for a built-in C type, defined with a typedef declaration. `mp_limb_t` is usually defined as an `unsigned long int` or `unsigned long long int`, depending on the compiler and target platform.

In C, expressions involving unsigned integer type operands follow the laws of arithmetic modulo 2^b where b is the size of the operand type in bits [6, section 2.2] [4, §6.2.5]. The result of a signed type expression overflowing is not specified, as the representation of signed types is not defined by the language. In practice, however, signed integers are almost always stored in two's complement representation;[1] this usually means that signed integers wrap-over in such a way that the largest positive value is followed by the largest negative value.

There are several integer types in C, including `char`, which is usually 8 bits; `short int`, which is usually 16 bits; and `int`, which is usually 32 bits. Assuming that `char` is 8 bits long, why not simply redefine `mp_limb_t` as an `unsigned char`?

It is not quite that simple. Even if we would store values as `unsigned char`'s, they are still converted to `int`'s when used in arithmetic expressions, because of a feature of the C language called *integer promotion* [4, §6.3.1.1]. In addition, we would like to allow limbs of sizes ranging from 4 to 30 bits, not just 8 and 16 bits.

Example 2.1. Why is integer promotion a problem? Consider the following C code:

```
unsigned char x = 128, y;
y = (2 * x) / 2;
```

What is the value of `y` after it has run? Had the code truly used arithmetic modulo 256 (2^8), then the result would have been zero, as $2 \cdot 128 \bmod 256 = 0$; however, because of integer promotion, `y` will actually be $256/2 = 128$.

2.3 Implementation strategies

In this section, we will describe three strategies for implementing artificial limbs in GMP. We chose one of them, and here we will explain why.

2.3.1 Use an abstraction layer

The most straightforward approach, in a sense, would be to rewrite GMP to use an abstraction layer to perform arithmetic. More specifically, it would have to call functions or macros to perform arithmetic operations, instead of the usual operators. Instead of writing `x + y`, one would write something like `add(x, y)`. Other operators would need to be replaced in the same way. It would then be possible to redefine the arithmetic functions to use fewer bits of precision and avoid the undesired effects of integer promotion.

For example, consider the following line of source code from GMP:

```
u01 += q * u00;
```

[1]In two's complement representation, the negation of a positive integer n is stored as $2^b - n$, where b is the size of the variable in bits.

With an abstraction layer for arithmetic, it would have to be rewritten to something like:

```
add_to (u01, mul (q, u00));
```

Since GMP consists of approximately 100 000 lines of C source code,[2] rewriting it in this way would be a huge effort. Additionally, the resulting code would be much more difficult to understand, maintain, and extend. We consider this solution to be unacceptable.

2.3.2 Modify the compiler

Another potential solution would be to modify a C compiler, adding a custom C type for limbs. In that case, the GMP source code could continue to use the ordinary arithmetic operators. While changes would still need to be made (as we shall see), the changes needed would be much smaller.

While this would probably be a viable solution, we did not choose to implement it for two reasons. First, modern compilers are complex pieces of software. GCC, for example, consists of 4 million lines of source code.[2] Since I am not familiar with the internals of any compiler, I assumed that it would be too difficult for me to add a custom type this way. Second, even if I did manage to modify a compiler, it would be difficult to maintain this modification since new releases of compilers are regularly made.

2.3.3 Overload the operators using C++

The C++ programming language is almost a superset of C, except for some minor details [5, §C.1] [7, §1.6]. The interesting thing about C++ is that we can redefine the meaning of almost all operators by *overloading* them. My supervisor had already made GMP compilable by a C++ compiler (except for a few trivial cases which had crept up). He suggested that by defining `mp_limb_t` as a class in C++ and overload the operators, it would be possible to make it behave similarly to a built-in integer type.

This is the approach we chose, and which we will describe in the next chapter.

[2]Measured with David A. Wheeler's 'SLOCCount' program.

Chapter 3

Implementation

3.1 Basics

We have implemented artificial limbs in GMP as two classes in C++: `mp_limb_t` and `mp_limb_signed_t`. They are very similar, except for signedness, and replace the typedefs in `gmp.h` with the same names. They are not used by default, but can be activated by passing the flag `--enable-alimbs=`*`B`* to the `configure` script when building GMP, where *`B`* is the desired limb size in bits. The flag tells the build process to omit the regular typedefs, and instead include the file `alimb.h` where the classes are defined. The flag also causes GMP to be built with a C++ compiler (`g++` by default), and with all assembly code disabled.

The source file `alimb.h` is included in the appendix, for reference.

3.2 Value storage

The artificial limb classes each has exactly one member variable, called `value`, which is used for storing the numeric value of a limb instance. The variable is declared to have a built-in integer type just large enough to hold the desired number of limb bits. Since the C++ language standard does not define the exact size of built-in integer types, the `(u)int`*`N`*`_t` type aliases from the standard library are used, where *`N`* is the size of the type in bits [4, §7.18.1.1]. These are usually available in 8-, 16-, 32-, and 64-bit sizes.[1]

Often, the desired limb size does not match any of the integer types exactly, in which case the next larger integer type is used. For example, if GMP is built with 10-bit limbs, a 16-bit integer type is used. In that case, the unused bits are masked out by the constructor of the limb class. In the case of `mp_limb_signed_t`, the

[1]Note that even if the sole data member of a class instance only occupies one byte (for example), nothing in the C++ standard guarantees that the object as a whole occupies one byte, as the compiler is free to add padding at the end of an object [5, §5.3.3]. When we tested with g++ 4.6.2 on GNU/Linux and x86, the artificial limbs did not get any extra padding at the end, but this need to be the case on other platforms.

Table 3.1. Operators overloaded by the artificial limb classes.

Unary	Arithmetic	Bitwise	Relational
`++`	`+`	`&`	`==`
`--`	`-`	`|`	`!=`
`-`	`*`	`^`	`>`
`~`	`/`	`<<`	`>=`
	`%`	`>>`	`<`
			`<=`

unused bits are set to ones if the number is negative, so that comparisons work as expected.[2]

3.3 Overloaded operators

Most of the C operators are overloaded for the artificial limb classes. The overloaded operators are listed in table 3.1. In addition, the corresponding assignment operators for the arithmetic and bitwise operations are also overloaded (i.e., `+=`, `-=`, `&=`, and so on). The assignment operator (`=`) does not need to be overloaded, since the default implementation supplied by C++ suffices. The operator overloads are straightforward, although they contain some subtleties. As an example, here is an overload for the addition operator:

```
template<class T> inline mp_limb_t
operator+ (mp_limb_t a, T b)
{
  return a.value + b;
}
```

Because a template is used, this overload applies to any addition where the left operand is an `mp_limb_t`; the right operand can be of any type. When the right operand is a numeric type, it is added to the value of the limb and returned. Since the return type of the overload is `mp_limb_t`, the result is implicitly converted to an `mp_limb_t` by being passed to the constructor of the class. The constructor truncates the result to the limb size, thus implementing arithmetic modulo 2^b.

There are many different combinations which need to be overloaded:

- 14 binary operators plus four unary operators;
- two distinct limb classes;

[2]This feature assumes that two's complement representation of signed integers is used, and thus implements sign extension in these cases.

- for each binary operator, either the left or the right operand can be a limb, or both;
- the 10 bitwise and arithmetic operators also have an assignment version.

Because of the large number of overloads needed, we have written them using preprocessor macros to make the code shorter and less repetitive. This means that the example of the addition operator above is not actually written as such in the source file `alimb.h`, but is generated by a macro invocation.

3.4 Implicit type conversion

One of the issues during the implementation was which implicit type conversions to allow. By default, one cannot convert a built-in numeric type to a class in C++, or vice versa. By defining a constructor for the class which takes a numeric type argument, it becomes possible to convert that numeric type to an instance of the class. Similarly, it is possible to define conversions from a class type to a numeric type, by defining a conversion function.

Allowing implicit conversion to a limb type is useful, as it enables initialization and assignment from built-in numeric types such as integers:

```
mp_limb_t x = 1;
x = 2;
```

It is also useful to be able to convert *from* limb types to other numeric types, and this is done in several places in GMP (for example, when calculating how many limbs are needed to represent the result of a computation). Here we have a problem. Consider this slightly modified version of example 2.1 on page 6:

```
mp_limb_t x = 128, y;
y = (2 * x) / 2;
```

Assuming that `mp_limb_t` is 8 bits, what happens here? If we allow implicit conversion to integer types, the compiler might decide to convert `x` to an `int` before evaluating the expression. This is exactly the problem we wanted to avoid when deciding to implement the limb types as C++ classes.

Fortunately, we can avoid this problem by making sure to overload the operators for all possible operand types (with the help of templates). Thanks to the rather complex C++ overload resolution rules [5, §13.3.3], these overloaded operators are guaranteed to be used instead of the built-in operators, and will thus not cause any implicit conversions.

3.5 Result type of expressions

Consider the following code:

```
int x, y = 255;
mp_limb_t z = 1;
x = y + z;
```

With 8-bit limbs, what is the value of `z` after this code has run? One would expect it to be 256, since `x` and `y` have type `int`; but with the `+` operator overloaded, the result of `y + z` will have type `mp_limb_t`, which truncates the result to 8 bits, yielding zero.

It is not clear what the best way to solve this is. One idea we had was to change the result types of additions, such that they would always return the type of the first operand. This would, in effect, make addition non-commutative. In this case, `y + z` would result in 256, while `z + y` would be equal to zero. While this might seem strange, it could make sense in the context of GMP, because the GMP code often uses idioms where this works. We did not explore this possibility, however.

The solution we chose was to modify the problematic code using explicit type-casts. The example above would then be rewritten

```
int x, y = 255;
mp_limb_t z = 1;
x = y + (int) z;
```

which forces the addition to be done using the `int` type. It also does the same thing when built without artificial limbs, so there is no need to have any separate code.[3]

The problem now is actually to *find* the problematic places in the code. We found them by running the test cases and investigating the ones that failed. There is an alternative to this, however, and that is to forbid implicit type conversion from limbs. In that case, the code above would have provoked an error from the compiler, thus making it easy to find. The downside is that there are many "false alarms": places in the code where limbs are assigned to other types, but which do not have this problem. There is a genuine trade-off to be made here: whether to simplify the problem-finding process, or limit the amount of code that needs to be changed.

[3]This is not completely true. If `mp_limb_t` is larger than `int` (which is true on some platforms), then `y` would be promoted to an `mp_limb_t` in the first example but not the second. While it does not matter here, it can make a difference in certain cases.

Chapter 4

Results

4.1 Building GMP with artificial limbs

Building GMP with artificial limbs went surprisingly well. Some minor changes were needed; in particular, about 20 places invoking the conditional operator (`?:`) needed to be modified because the operands were of different type. Typically, it looked like

```
return a == b ? a : 0;
```

where `a` has type `mp_limb_t`. Since `0` is an `int`, the compiler does not know what type the entire expression should return. Normally, when `mp_limb_t` is a numeric type, the compiler can simply perform the usual arithmetic promotion to figure out a common type, but it is not possible here since `mp_limb_t` is not a numeric type. We changed these lines to the equivalent of

```
return a == b ? a : (mp_limb_t) 0;
```

which solves the problem, and works like before when building without artificial limbs.

4.2 Issues and bugs found

Being able to compile GMP with artificial limbs was only the first step, however. It also needed to be able to *run* with artificial limbs. Running the test suite with artificial limbs generated different types of issues. Ordered from the least to the most severe, these were:

a) Errors due to the implementation of artificial limbs. These errors included mainly incorrect value truncations, see section 3.5. Some errors were also bugs in `alimb.h` which we then fixed. These were not “errors” in GMP by any measure, but happened because of peculiarities of the artificial limb implementation.

Table 4.1. Number of issues of each category fixed in GMP.

Category	Issues
a)	32
b)	31
c)	11
d)	3
e)	2

b) Errors due to intentional lack of generality. While GMP in general tries to handle limbs of all sizes, certain parts of the code handled only limbs of specific sizes (usually 32 or 64 bits). In some cases, we were able to generalise the code; in other cases, we disabled the problematic code because we felt it was not worth the trouble to generalise it.

c) Errors due to accidental lack of generality. These were cases where the code was probably supposed to handle limbs of “any” size but, in fact, did not.

d) Errors which would only show up with artificial limb sizes. These were real bugs, in a sense, because the code made some kind of logical error. However, for various reasons, they would never occur when using normal limb sizes, such as 32 and 64 bits.

e) Errors which could be triggered when running with real limb sizes.

The number of issues we fixed in each category above is listed in table 4.1. Changes in the test suite are not included. The numbers should be taken with a grain of salt since what constitutes “one change” is not exactly defined. In particular, the changes in category a) were generally one-line changes, while changes in categories b) and c) were more invasive.[1] In addition, these changes have not made all test cases run successfully with all limb sizes; more changes will be needed to be able to run the complete test suite successfully with all limb sizes. By investigating the remaining failing test cases, more issues will be found, and some of them could belong to category e).

The two bugs from category e) we found were two cases of buffer overflow; i.e., a buffer which was not large enough to hold all possible values which could be stored there. The bugs were not present in a stable release of GMP though, only in development code which will be a part of the next major GMP release.

[1]Most of the changes, particularly the larger ones, were done by the project's supervisor, Torbjörn Granlund.

4.3 Conclusions

In total, we modified approximately 800 lines of code. Considering that GMP consists of about 100 000 lines of C code, this is only 0.8% of the code base. While we did not get all test cases to run successfully with all limb sizes, it still shows that only minor changes were needed to make GMP run with small limb sizes. Although most changes so far have not been done to fix actual bugs in a strict sense, we do believe that they have improved the quality of the code in GMP. For example, the type `mp_limb_t` was occasionally used as a generic “fast integer type” previously, but is now more stringently used as a part of arbitrary-size numbers.

Another important result of the project is that future versions of GMP will include the artificial limb code. Thus, any newly developed code in GMP will be able to take advantage of the improved testing offered by the artificial limbs.

Operator overloading is sometimes derided as being merely an unnecessary, “cosmetic” feature. For example, the web site of the Go programming language states that operator overloading “seems more a convenience than an absolute requirement” [2]. We believe that this project has provided a counter-example.

Bibliography

[1] Free Software Foundation. GNU Lesser General Public License, version 3, June 2007. Available from: `http://www.gnu.org/licenses/lgpl.html` [cited February 18, 2012].

[2] The Go Authors. FAQ — The Go Programming Language, January 2012. Available from: `http://golang.org/doc/go_faq.html#overloading` [cited February 18, 2012].

[3] Torbjörn Granlund and the GMP development team. *The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, 5.0.2 edition, 2011. Available from: `http://gmplib.org/gmp-man-5.0.2.pdf` [cited February 18, 2012].

[4] ISO. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, December 1999.

[5] ISO. *ISO/IEC 14882:2011: Programming Languages — C++*. International Organization for Standardization, Geneva, September 2011.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, 2nd edition, 1988.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, 3rd special edition, 2000.

Appendix

The contents of the file `alimb.h` is reproduced below for reference. It contains the artificial limb implementation which was developed during this project.

```
/* alimb.h -- Artificially small limbs using -*- C++ -*-.

   Contributed to the GNU project by Per Olofsson.

Copyright 2011, 2012 Free Software Foundation, Inc.

This file is part of the GNU MP Library.

The GNU MP Library is free software; you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 3 of the License, or (at your
option) any later version.

The GNU MP Library is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public
License for more details.

You should have received a copy of the GNU Lesser General Public License
along with the GNU MP Library.  If not, see http://www.gnu.org/licenses/.  */

/* This file is included by gmp.h if GMP is built with artificial
   limbs.  It defines the types mp_limb_t and mp_limb_signed_t to be
   of a certain (compile-time configured) size n == GMP_LIMB_BITS for
   any even n such that 4 <= n <= 32.  The purpose of this exercise is
   to improve the effectiveness of the test suite.

   These artificially small limbs are implemented as C++ classes
   with overloaded operators.  */

#include <stdint.h>

/* Here we assume that int is 32 bits.  We cannot use GMP_NUMB_MASK
   because it depends on mp_limb_t having been defined already.  */
#if GMP_LIMB_BITS == 32
```

```
#define __GMP_ALIMB_MASK (~0)
#else
#define __GMP_ALIMB_MASK (~(~0 << GMP_LIMB_BITS))
#endif

/* Make the mp_alimb_*_storage_t types large enough to hold an
   artificial limb of the requested size, but as small as possible to
   save memory.  */
#if GMP_LIMB_BITS <= 8
typedef uint8_t  mp_alimb_storage_t;
typedef int8_t   mp_alimb_signed_storage_t;
#elif GMP_LIMB_BITS <= 16
typedef uint16_t mp_alimb_storage_t;
typedef int16_t  mp_alimb_signed_storage_t;
#else
typedef uint32_t mp_alimb_storage_t;
typedef int32_t  mp_alimb_signed_storage_t;
#endif

struct mp_limb_signed_t;

struct mp_limb_t {
  mp_alimb_storage_t value;

  mp_limb_t () {}

  /* Allow implicit conversion from any numeric type.
     Always store the value truncated.  */
  template<class T>
  mp_limb_t (T v)
    : value(static_cast<mp_alimb_storage_t>(v) & __GMP_ALIMB_MASK) {}

  /* Allow explicit conversion from pointers, used by certain tests. */
  explicit mp_limb_t (void *v)
    : value(reinterpret_cast<uintptr_t>(v) & __GMP_ALIMB_MASK) {}

  /* Allow implicit promotion from signed limbs. */
  mp_limb_t (mp_limb_signed_t l);

  /* Allow implicit conversion to other numeric types (built-in
     conversions will be implicitly applied to
     mp_alimb_storage_t).  */
  operator mp_alimb_storage_t () const { return value; }
};

struct mp_limb_signed_t {
  mp_alimb_signed_storage_t value;

  mp_limb_signed_t () {}
  /* Sign-extend the stored value (for comparisons). */
  template<class T>
  mp_limb_signed_t (T v)
    : value((static_cast<mp_alimb_signed_storage_t>(v) & __GMP_ALIMB_MASK)
            | -(v & (1 << (GMP_LIMB_BITS - 1)))) {}

  /* Demand an explicit cast when converting from an unsigned limb. */
  explicit mp_limb_signed_t (mp_limb_t l)
  {
    *this = mp_limb_signed_t(l.value);
  }

  operator mp_alimb_signed_storage_t () const { return value; }
};

inline mp_limb_t::mp_limb_t(mp_limb_signed_t l)
{
  *this = mp_limb_t(l.value);
}


/* Because we need to overload so many different combinations of
   operators and argument types, we use macros.  This makes the code
   much shorter and easier to modify.

   Templates can be and (in some cases) are used, but unlike macros
   they cannot be used to parametrize the operator to overload.  It is
   also difficult to force templates to only apply to some types
   (mp_limb_t and mp_limb_signed_t).

   Macro arguments:
   op            operator
   aop           assignment version of operator
   ltype         limb type
   other_ltype   the other limb type (with opposite signedness)  */

/**************** Assignment and binary operators ****************/

/* Overload an assignment operator for a specific limb type. */
#define __GMP_ALIMB_ASSIGN_OP(op, aop, ltype, other_ltype)         \
  inline ltype&                                                    \
  operator aop (ltype& lhs, ltype rhs)                             \
  { return lhs = lhs.value op rhs.value; }                         \
                                                                   \
  inline ltype&                                                    \
  operator aop (ltype& lhs, other_ltype rhs)                       \
  { return lhs = lhs.value op rhs.value; }                         \
                                                                   \
  template<class T> inline ltype&                                   \
  operator aop (ltype& lhs, T rhs)                                  \
  { return lhs = lhs.value op rhs; }                                \
                                                                    \
  template<class T> inline T&                                       \
  operator aop (T& lhs, ltype rhs)                                  \
  { return lhs aop rhs.value; }

/* Overload a binary operator for a specific limb type. */
#define __GMP_ALIMB_BINARY_OP(op, ltype, other_ltype)               \
  inline ltype                                                      \
  operator op (ltype a, ltype b)                                    \
  { return a.value op b.value; }                                    \
                                                                    \
  template<class T> inline ltype                                    \
  operator op (ltype a, T b)                                        \
  { return a.value op b; }                                          \
                                                                    \
  template<class T> inline ltype                                    \
  operator op (T a, ltype b)                                        \
  { return a op b.value; }                                          \
                                                                    \
  /* Pointer arithmetic should always return a pointer. */          \
  template<class T> inline T *                                      \
  operator op (T *a, ltype b)                                       \
  { return a op b.value; }                                          \
                                                                    \
  /* A binary operation with both signed and unsigned limb types   \
     should return an unsigned limb. */                             \
  inline mp_limb_t                                                  \
  operator op (ltype a, other_ltype b)                              \
  { return a.value op b.value; }

/* Overload both the binary and assignment version of an operator
   for both limb types. */
#define __GMP_ALIMB_AB_OP(op)                                       \
  __GMP_ALIMB_ASSIGN_OP(op, op ## =, mp_limb_t, mp_limb_signed_t)  \
  __GMP_ALIMB_ASSIGN_OP(op, op ## =, mp_limb_signed_t, mp_limb_t)  \
  __GMP_ALIMB_BINARY_OP(op, mp_limb_t, mp_limb_signed_t)            \
  __GMP_ALIMB_BINARY_OP(op, mp_limb_signed_t, mp_limb_t)

__GMP_ALIMB_AB_OP(+)
__GMP_ALIMB_AB_OP(-)
__GMP_ALIMB_AB_OP(*)
__GMP_ALIMB_AB_OP(/)
__GMP_ALIMB_AB_OP(%)
__GMP_ALIMB_AB_OP(<<)
__GMP_ALIMB_AB_OP(>>)
__GMP_ALIMB_AB_OP(|)
__GMP_ALIMB_AB_OP(&)
__GMP_ALIMB_AB_OP(^)

#undef __GMP_ALIMB_AB_OP
#undef __GMP_ALIMB_BINARY_OP
#undef __GMP_ALIMB_ASSIGN_OP

/**************** Unary operators ***************/

/* Overload increment or decrement operator */
#define __GMP_ALIMB_INCDEC(ltype, op)                            \
  /* Prefix version */                                           \
  inline ltype&                                                  \
  operator op ## op (ltype& l)                                   \
  { return l op ## = 1; }                                        \
                                                                 \
  /* Postfix version */                                          \
  inline ltype                                                   \
  operator op ## op (ltype& l, int)                              \
  {                                                              \
    ltype old(l);                                                \
    l op ## = 1;                                                 \
    return old;                                                  \
  }

/* Overload all unary operators for a certain limb type */
#define __GMP_ALIMB_UNARY_OPS(ltype)                     \
  inline ltype operator- (ltype l) { return -l.value; }  \
  inline ltype operator~ (ltype l) { return ~l.value; }  \
  __GMP_ALIMB_INCDEC(ltype, +)                           \
  __GMP_ALIMB_INCDEC(ltype, -)

__GMP_ALIMB_UNARY_OPS(mp_limb_t)
__GMP_ALIMB_UNARY_OPS(mp_limb_signed_t)

#undef __GMP_ALIMB_UNARY_OPS
#undef __GMP_ALIMB_INCDEC

/**************** Relational operators ***************/

/* Overload a relational operator for a specific limb type. */
#define __GMP_ALIMB_REL_OP_T(op, ltype, other_ltype)    \
  inline bool                                          \
  operator op (ltype a, ltype b)                       \
  { return a.value op b.value; }                       \
                                                       \
  /* Perform unsigned comparisons between mp_limb_t    \
     and mp_limb_signed_t.  */                         \
  inline bool                                          \
  operator op (ltype a, other_ltype b)                  \
  {                                                     \
    return ((a.value & __GMP_ALIMB_MASK) op             \
            (b.value & __GMP_ALIMB_MASK));              \
  }                                                     \
                                                        \
  template<class T> inline bool                         \
  operator op (ltype a, T b)                            \
  { return a.value op b; }                              \
                                                        \
  template<class T> inline bool                         \
  operator op (T a, ltype b)                            \
  { return a op b.value; }

/* Overload a relational operator for both limb types. */
#define __GMP_ALIMB_REL_OP(op)                          \
  __GMP_ALIMB_REL_OP_T(op, mp_limb_t, mp_limb_signed_t) \
  __GMP_ALIMB_REL_OP_T(op, mp_limb_signed_t, mp_limb_t)

__GMP_ALIMB_REL_OP(==)
__GMP_ALIMB_REL_OP(!=)
__GMP_ALIMB_REL_OP(<)
__GMP_ALIMB_REL_OP(>)
__GMP_ALIMB_REL_OP(<=)
__GMP_ALIMB_REL_OP(>=)

#undef __GMP_ALIMB_REL_OP
#undef __GMP_ALIMB_REL_OP_T
```

TRITA-CSC-E 2012:003
ISRN-KTH/CSC/E--12/003-SE
ISSN-1653-5715