

Multiprecision integers; a Fortran 90 module

Per Håkan Lundow

September 4, 2000

Abstract

The author has written a Fortran 90 based module for multiprecision integers. This paper gives an introduction to its usage and contains a reference of its procedures.

1 Introduction

There are of course many modules for multiprecision integers (henceforth written as mp-integers) to be found on the internet, one of the better is written by David Bailey. If you need mp-integers in the range of thousands of binary digits then read no further, use his package instead ¹. If you however, like the author of this paper, need mp-integers in a smaller range, say less than 1000 binary digits, then you are reading the right paper since this module is likely to be faster. Even better, should you discover that you, after all, need much larger mp-integers, then the source code you have written is, to some extent, compatible with Bailey's package. This module should work on computers where `BIT_SIZE (0)` is a power of two and `DIGITS (0) = BIT_SIZE (0) - 1`. That should cover just about every computer on the market since typically `BIT_SIZE (0) = 32` (or 64) and `DIGITS (0) = 31` (or 63).

2 An example

Actually the package consists of three modules in three files depending upon what you need, unsigned (`mpiunsgn.f90`), signed (`mpisgn.f90`) and compactly stored unsigned mp-integers (`mpicmp.f90`). To use the package in the simplest case, unsigned mp-integers, you need only the file `mpiunsgn.f90` (you may need to change the suffix on the file to suit your compiler).

Open the file with your editor (emacs, vi, alpha...) and locate the "user defined" section at the beginning of the module containing the line

```
INTEGER, PRIVATE, PARAMETER :: initial_digits_mpi = 200
```

This parameter allows you to enter the number of binary digits in your mp-integers and it is the only parameter you need to set. Choose it with care since it will affect the running time of your programs.

Now we compile the file. On an IBM computer you would compile it with something like the command

```
> xlf90 -c -O3 -qstrict mpiunsgn.f
```

¹See <http://www.nersc.gov/~dnhb/mpdist/mpdist.html>

On a SUN we might use the command

```
> f90 -c -03 mpiunsgn.f90
```

Your program must contain a line

```
USE UNSIGNED_MP_INTEGERS
```

at the top, right after the PROGRAM statement but before the IMPLICIT NONE statement. This introduces the new type MP_INTEGER to your program. We start with a very short, very simple, not very useful, but complete program:

```
PROGRAM Main
USE UNSIGNED_MP_INTEGERS
IMPLICIT NONE

TYPE(MP_INTEGER) :: x, y

WRITE(*,*) "Enter x < " // TRIM (To_String (SQRT (huge_mpi)))

CALL MPREAD (5, x) ! Read from standard input

y = x**2 - 2*x + 1

WRITE(*,*) "x**2 - 2*x + 1: "
CALL MPWRITE (6, y) ! Write to standard output

STOP
END PROGRAM Main
```

If we run this program it will ask for a non-negative integer less than the square root of the largest integer it can store. Go ahead and type something in, let's say 1000000000000, and then hit return. The output will then be on the form you expect, as it would look if it had been an ordinary integer printed with WRITE(*,*). In this case it will be 99999999998000000000001.

That was a quick demonstration of the unsigned mp-integers but what if you need signed mp-integers? Well, that's where the file `mpisgn.f90` comes in. After having compiled `mpiunsgn.f90` and `mpisgn.f90`, in that order, the type MP_INTEGER now can store signed mp-integers. The only thing you need to change in your program is the USE-statement, replace it with

```
USE SIGNED_MP_INTEGERS
```

No changes are required beyond this. Note that the behavior of the signed mp-integers is the same as you might expect from ordinary integers, e.g. $-8/5 = -1$ and $\text{MOD}(-8, 5) = -3$.

So what about the third file then? If you have lots of unsigned mp-integers and need to save memory space, then you want to use compactly stored mp-integers. First compile the file `mpiunsgn.f90` and then the file `mpicomp.f90`. Also, change the USE-statement to

```
USE COMPACT_MP_INTEGERS
```

and you're off. Multiplications will run somewhat slower with this module but additions will run slightly faster. There is no module for compactly stored signed mp-integers though (yet).

3 Reference

Throughout this section we assume that I is a default integer, R is a real, A is a string and M is an mp-integer. The value of the kind type parameter for the real type is KIND (0.0) and is stored in the constant `rk_mpi` found at the beginning of the files. Feel free to change its value, preferably with the `SELECTED_REAL_KIND` function.

3.1 Assignments

The following types of assignments are allowed: `M = M`, `M = I`, `M = R`, `M = A`, `I = M`, `R = M` and `A = M` as in the following examples:

```
TYPE(MP_INTEGER) :: x, y( 10 )
CHARACTER(60) :: a
INTEGER :: i
REAL :: r      ! or REAL(rk_mpi) :: r
y = ten_mpi    ! all elements of y have value 10
x = y( 1 )     ! x has value 10
x = 1.97e4     ! x has value 19700
x = 123456     ! x has value 123456
r = x         ! r has value 123456.0
i = x         ! i has value 12345
x = "123456"   ! x has value 123456
x = "123 456"  ! x has value 123456
x = "1.97E+4"  ! x has value 19700
a = x         ! a has value "12345" (left-adjusted)
```

3.2 Operators

You can use the standard binary operators on your mp-integers and in the case of arithmetic operators you can also combine them with the intrinsic integers. The arithmetic operators are `+`, `-`, `*`, `/` and `**` and the relational operators are `==`, `/=`, `<`, `<=`, `>` and `>=`. Both operands of the relational operators must be mp-integer. For result types of arithmetic operations, see Table 3.2. Note that if you are using unsigned mp-integers then $x - y$ is only defined when $x \geq y$. Also, $-x$ is only defined when $x = 0$. Here is an example of how to use the operators:

```
TYPE(MP_INTEGER) :: x, y, z
x = ten_mpi ** range_mpi ! compute square root of x
y = x
DO
  z = y
  y = (y + x / y) / 2
  IF ( y >= z ) EXIT
END DO
IF ( y > z ) y = z
! y has value SQRT (x)
```

Operation	First operand	Second operand	Result
+	M	M	M
+	M	I	M
+	I	M	M
-	M	M	M
-	M	I	M
-	I	M	M
-	M		M
*	M	M	M
*	M	I	M
*	I	M	M
/	M	M	M
/	M	I	M
/	I	M	M
**	M	I	M

Table 1: Arithmetic operations

3.3 Constants

`zero_mpi` is of type mp-integer and has the value zero.

`one_mpi` is of type mp-integer and has the value one.

`two_mpi` is of type mp-integer and has the value two.

`ten_mpi` is of type mp-integer and has the value ten.

`huge_mpi` is of type mp-integer and its value is that of the largest mp-integer.
See the intrinsic function `HUGE`.

`digits_mpi` is of type integer and its value is the number of significant binary digits for mp-integers. See the intrinsic function `DIGITS`.

`range_mpi` is of type integer and its value is the decimal exponent range for mp-integers. See the intrinsic function `RANGE`.

`rk_mpi` is of type integer and its value is the kind type parameter for the real type used by the module.

3.4 Procedures

ABS (x)

Extended intrinsic function. Returns the absolute value $|x|$.

Argument is of type mp-integer.

Result is of type integer.

The value of the result is the absolute value $|x|$.

This function is only available when using signed mp-integers.

```
TYPE(MP_INTEGER) :: x, y
x = -123
y = ABS (x) ! y has value 123.
```

DIFF (x, y)

Function. Returns the difference $|x - y|$.

Two arguments, both of type mp-integer.

Result is of type mp-integer.

The value of the result is $|x - y|$.

```
TYPE(MP_INTEGER) :: x, y, z
x = 123
y = 321
z = DIFF (x, y) ! z has value 198
z = DIFF (y, x) ! z has value 198
```

DIM (x, y)

Extended intrinsic function. Returns the difference $x - y$ if it is positive, otherwise zero.

Two arguments, both of type mp-integer.

Result is of type mp-integer.

If $x > y$ then the value of the result is $x - y$. If $x \leq y$ then the value of the result is zero.

```
TYPE(MP_INTEGER) :: x, y, z
x = 123
y = 321
z = DIM (x, y) ! z has value 0
z = DIM (y, x) ! z has value 198
```

DIV_MOD (x, y, q, r)

Subroutine. Returns both quotient and remainder.

All arguments are of type mp-integer. Third and fourth argument are optional and INTENT(OUT).

On return we have $x = qy + r$ where $0 \leq r < y$. The result value of q is $\lfloor x/y \rfloor$ and the result value of r is $x - qy$. If $y = 0$ an error occurs. For signed mp-integers the same rules holds as for division and remainders of INTEGERS, i.e. r has the same sign as x and the result value of q is the same as x/y would return.

```
TYPE(MP_INTEGER) :: x, y, q, r
x = 37
y = 10
CALL DIV_MOD (x, y, q, r)
! q has value 3 and r has value 7
```

DOT_PRODUCT (x, y)

Extended intrinsic function. Returns the dot product of two vectors x and y .

Two arguments, both are mp-integer arrays of rank 1 having the same size.

Result is of type mp-integer

The value of the result is $x(1)y(1) + x(2)y(2) + \dots + x(n)y(n)$ where n is the size of the vectors x and y . If the vectors are zero-sized then the value of the result is zero. If the vectors have different sizes an error occurs.

```
TYPE(MP_INTEGER) :: x( 10 ), y( 10 ), z
INTEGER :: i
DO i = 1, 10
  x( i ) = i
  y( i ) = i
END DO
y = DOT_PRODUCT (x, y) ! z has value 385
```

INT (x)

Extended intrinsic function. Convert mp-integer to integer.

Argument is of type mp-integer.

Result is of type integer.

Result has the same value as the argument. The result value is undefined if it cannot be represented in the integer type.

```
TYPE(MP_INTEGER) :: x
INTEGER :: n
x = MPINT (123)
n = INT (y) ! n has value 123
```

LOG (x, n)

Extended intrinsic function. Returns the natural logarithm.

First argument is of type mp-integer. The second argument is optional and of type integer.

Result is of type real.

If the second argument is present then the result value is $\log_n x$, otherwise it is $\log_e x$.

```
TYPE(MP_INTEGER) :: x
REAL :: r
x = 12345
r = LOG (x)      ! r has value 9.421006402 (approx)
r = LOG (x, 2) ! r has value 13.59163922 (approx)
```

MAX (x, y)

Extended intrinsic function. Returns maximum value.

Two arguments, both of type mp-integer.

Result is of type mp-integer.

Result value is that of the largest argument.

```
TYPE(MP_INTEGER) :: x, y, z
x = 123
y = 321
z = MAX (x, y) ! z has value 321
```

MIN (x, y)

Extended intrinsic function. Returns minimum value.

Two arguments, both of type mp-integer.

Result is of type mp-integer.

Result value is that of the smallest argument.

```
TYPE(MP_INTEGER) :: x, y, z
x = 123
y = 321
z = MIN (x, y) ! z has value 123
```

MOD (x, y)

Extended intrinsic function. Returns the remainder.

Two arguments, both of type mp-integer.

Result is of type mp-integer.

If $y \neq 0$ then the value of the result is $x - y[x/y]$. If $y = 0$ then an error occurs. For signed mp-integers r has the same sign as x .

```

TYPE(MP_INTEGER) :: x, y, z
x = 123
y = 321
z = MOD (x, y) ! z has value 123
z = MOD (y, x) ! z has value 75

```

MPINT (x)

Function. Convert to mp-integer.

Argument is of type integer, real or string.

Result is of type mp-integer.

Result has the same value as the argument if the argument is of type integer or the string contains an exact integer. If the argument is of type real then the result value is the largest integer not exceeding the value of the argument, i.e. the floor of the argument. An error occurs if the result value cannot be represented in the mp-integer type.

```

TYPE(MP_INTEGER) :: x
x = MPINT (1.9)           ! x has value 1
x = MPINT (23)           ! x has value 23
x = MPINT ("12 34 56") ! x has value 123456

```

MPREAD (u, x, iostat)

Subroutine. Reads x from unit u .

Argument x is of type mp-integer. Argument u and $iostat$ is of type integer. Argument $iostat$ is optional and INTENT(OUT).

Reads x from unit u . Argument $iostat$ work like for READ. Internally, a string is read as list-directed input and then copied to x . See Section 3.1 for more on readable formats of mp-integers.

```

TYPE(MP_INTEGER) :: x
CALL MPREAD (5, x)
! Type 12345 and return.
! x has value 12345

```

MPWRITE (u, x, fmt, iostat)

Subroutine. Writes x to unit u .

Argument x is of type mp-integer. Argument u and $iostat$ is of type integer. Argument fmt is of type string. Both fmt and $iostat$ are optional, $iostat$ is INTENT(OUT).

Writes x to unit u . Argument $iostat$ work like for WRITE. Internally, x is first copied to a string so that one can use A-editing for formatting. If fmt is present this string is right-adjusted.

```

TYPE(MP_INTEGER) :: x
x = 123
CALL MPWRITE (6, x, fmt='(a5)') ! Prints ' 123'
CALL MPWRITE (6, x)             ! Prints '123'

```

ORDER (x, y)

Function. Returns 1, 0 or -1 depending on whether $x < y$, $x = y$ or $x > y$.

Two arguments, both of type mp-integer.

Result is of type integer.

If $x < y$ then the value of the result is 1. If $x = y$ then the value of the result is 0. If $x > y$ then the value of the result is -1.

TYPE(MP_INTEGER) :: x, y

INTEGER :: n

x = 123

y = 321

n = ORDER (x, y) ! n has value 1

n = ORDER (x, x) ! n has value 0

n = ORDER (y, x) ! n has value -1

PARITY (x)

Function. Returns 1 or 0 depending on whether x is odd or even.

Argument is of type mp-integer.

Result is of type integer.

If x is odd then the value of the result is 1. If x is even then the value of the result is 0.

TYPE(MP_INTEGER) :: x, y

INTEGER :: n

x = 0

y = 3

n = PARITY (x) ! n has value 0

n = PARITY (y) ! n has value 1

PRODUCT (x)

Extended intrinsic function. Returns the product of vector x .

Argument is an mp-integer array of rank 1.

Result is of type mp-integer

The value of the result is $x(1) \cdot x(2) \cdot \dots \cdot x(n)$ where n is the size of the array x . If the vector is zero-sized then the value of the result is one.

TYPE(MP_INTEGER) :: x(10), y

INTEGER :: i

DO i = 1, 10

 x(i) = i

END DO

y = PRODUCT (x) ! y has value 10! = 3628800

RANDOM_NUMBER (x, n)

Extended intrinsic subroutine. Returns a pseudorandom number from the uniform distribution over the range $0 \leq x < 2^n$.

First argument is of type mp-integer and is INTENT(OUT). The second argument is of type integer.

The result is in x and is a pseudorandom number uniformly chosen from the interval $[0, 2^n)$. Note that the internal pseudorandom number generator is used to build up the result so the randomness can (and should) be doubted.

```
TYPE(MP_INTEGER) :: x
CALL RANDOM_NUMBER (x, 10) ! x is in [0,1023].
```

REAL (x)

Extended intrinsic function. Convert mp-integer to real.

Argument is of type mp-integer.

Result is of type real.

Result has the same value as its argument. The result value is undefined if it cannot be represented in the real type.

```
TYPE(MP_INTEGER) :: x
REAL :: r
x = MPINT (123)
r = REAL (y) ! r has value 123.0
```

SCALE (x, i)

Extended intrinsic function. Returns the scaled value $x2^i$.

First argument is of type mp-integer. Second argument of type integer.

Result is of type mp-integer

The value of the result is $x2^i$. If $x < 2^n$ then the value of the result of **SCALE (x, -n)** is zero.

```
TYPE(MP_INTEGER) :: x, y
x = 30
y = SCALE (x, 5) ! y has value 960
y = SCALE (x, -5) ! y has value 0
```

SIGN (x, y)

Extended intrinsic function. Returns $|x| \cdot \text{sign}(y)$.

Two arguments, both of type mp-integer.

Result is of type mp-integer.

If $y \geq 0$ then the value of the result is $|x|$. If $y < 0$ then the value of the result is $-|x|$.

This function is only available when using signed mp-integers.

```
USE SIGNED_MP_INTEGERS ! OBS
TYPE(MP_INTEGER) :: x, y, z
x = -123
y = 321
z = SIGN (x, y) ! z has value 123
z = SIGN (y, x) ! z has value -321
```

SQRT (x)

Extended intrinsic function. Returns the square root.

Argument is of type mp-integer.

Result is of type mp-integer.

The result value is $\lfloor \sqrt{x} \rfloor$.

```
TYPE(MP_INTEGER) :: x, y
x = 12345
y = SQRT (x) ! y has value 111
```

SUM (x)

Extended intrinsic function. Returns the sum of vector x .

Argument is an mp-integer array of rank 1.

Result is of type mp-integer

The value of the result is $x(1) + x(2) + \dots + x(n)$ where n is the size of the array x . If the vector is zero-sized then the value of the result is zero.

```
TYPE(MP_INTEGER) :: x( 10 ), y
INTEGER :: i
DO i = 1, 10
    x( i ) = i
END DO
y = SUM (x) ! y has value 55
```

TO_STRING (x, n)

Function. Returns x as a string.

Argument x is of type mp-integer. Argument n is optional and of type integer.

Result is of type string.

If n is present then the value of x is returned as a right-adjusted string of length n . If the value of x can't be fitted into a string of length n an error occurs. If n is not present then the value of x is returned as a left-adjusted string of length `range_mpi+3`.

```
TYPE(MP_INTEGER) :: x
x = 123
WRITE(*,*) 'x = ' // TRIM (To_String (x))
! Prints 'x = 123'
```
