1992

D.T. Winter

A package for long integer arithmetic on the Cray Y-MP

Department of Numerical Mathematics Report NM-R9227 December

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

A Package for Long Integer Arithmetic on the Cray Y-MP

Dik T. Winter
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

This report describes a package for arbitrary long integer arithmetic as it has been implemented on the Cray Y-MP. A concise user manual is added in an appendix.

1991 Mathematics Subject Classification: Primary: 11-04, Secondary: 11Y99
Keywords & Phrases: Arithmetic, Long integers, Vector processors.

1. Introduction

We will describe a package for arbitrary long integers as it has been implemented on the Cray Y-MP/4. The actual contents are described in the appendix. The main paper details the reasons and history for some design decisions. The package is written mostly in C, but with quite a lot of support from Cray Assembler. As an added benefit, in the source of the code there is a lot of conditional compilation that allows:

a Easy porting to other systems (e.g. it is possible to just use C code only).
b Easy assembly support on other systems.

This is of course not the first package for such computations; and also it is not the first such package for the Cray. An earlier example can be found described in the paper by Buell and Ward (A Multiprecise Integer Arithmetic Package, The Journal of Supercomputing 3(1989), 89-107), which was specifically targeted at the Cray 2. The main reason for 'yet another package' is that most current packages are not specifically targeted at vector processors (and will indeed perform not so very well on such machines). The package described in this paper is designed to have good performance on such targets and, moreover, on non-vector processors the performance will be not much worse than that of other packages.

In the sequel we will describe the main reason for the packaging as was done; also some timing results are given, and compared with those of Buell and Ward.

2. Historical overview

The basic modules for this package were written in 1980-1983. The main objective was a program to do primality testing on long numbers as described in the article by Lenstra and Cohen (Mathematics of Computation, 48(1987), 103-121). For this purpose a number of base routines were written, first targeted at the CDC Cyber 175, and next at the Cray 1 (the last effort was supported by the 'Werkgroep Supercomputers'). Doing this the major effort was the design of low-level routines to do long integer arithmetic on these machines. Early on a decision was already made that for best performance on the Cray, the base representation of a *long integer* should be a series of machine words, holding the number with a number base of 2^23. Also in that early design there was no place for negative numbers (when doing primality testing all calculations are performed modulo the number to be tested), and that overflow would result in a fatal error (for the

Report NM-R9227
ISSN 0169-0388
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

same reason). A next decision was that *long integers* would be represented in a fixed number of words; two sizes were chosen that could coexist (because multiplication creates a double size number). For the Cray a single size *long integer* was chosen to be 32 words, while a double size *long integer* was 64 words (each 23 bits at most). Contrary to the decision of Buell and Ward to represent the base *digits* as floating-point numbers, this package stores them as integers.

The main feature of the package was a multiplication routine that would read the operands in registers, operate on those registers, and would write the total result back to memory again. I.e. there was no traffic to memory at all during the computations.

Also (similar to Buell and Ward) the carry propagation problem was solved as follows. Let us assume we want to add a to b. Set vector register c to the sum of a and b. Now a loop is performed over a sequence of instructions:

check whether c contains carries
if not leave loop
d = c shifted right the proper number of bits
c = c anded with the proper mask
shift up array d one position
add d into c
jump back

The number of times the loop is executed is amazingly small (on average the loop is performed only once), so this looks like a good way to vectorize long integer addition. (More details can be found in D.E. Knuth, The Art of Computer Programming, Volume 2 (1981), where the amount of carry propagation is discussed.)

From this we can cocnlude that the remark in Buell and Ward about the flatness of their 'normalization' is justified.

So far the history; the program was useful, and a lot of numbers have been proven prime using it.

3. What are the additions

One of the major defects in the package was that it did not cater for negative numbers; also the size of numbers was limited (it shares this feature with the Buell and Ward package).

So the current package grew out from the previous package to allow much more general use of large integers, that should be allowed to be much larger in size, and in fact arbitrarily large. This did however preclude the use of Fortran as the base language (as done in the previous package), as standard Fortran before Fortran 90 does not allow memory management. For that reason the new package was written such that it would primarily serve C users, although Fortran to C interfaces are available. When instructed to do so the routines will increase the size of the variables involved if the result would not fit in the previously allocated amount of storage. And in fact, the only error that will result from the calculation of successive powers of 2 comes when memory is exhausted.

A basic design decision has been that the original low-level Cray Assembler routines would be used also in this case. The result of this is that the base unit for the integers in the current package is not 23 bits, but 32 * 23 bits. So the package really works with numbers base 2 ^ (23 * 32) or:

361473786714651839609485931802192366508973300717001923159475447150424 8
102862334079879518618873894396122749267837803515619997819988324340412 9
619879532632910162314189970978766343329690527906605154864094201329081 9
886814068736

This appears to be very large, but it seems to work fairly well.

So the main result is that the original set of routines was expanded to support the specific needs for the new package (both Cray assembler and C code were expanded) and on top of that a complete interface was written to support arbitrary long integers. Also a number of operations were added to more easily support number theoretical computations.

Finally routines were added that deal with modular arithmetic both in normal representation and in the representation due to Peter Montgomery which is faster in many situations (Mathematics of Computation 44(1985), 519-521).

4. Timings for addition

Below we give the times for additions for a number of different cases. The examples used are the same as the examples of Buell and Ward; in addition, longer examples are timed. Column 1 displays the number of digits in the example numbers; column 2 gives the approximate time in microseconds. Column 3 gives the times found by Buell and Ward for their package on the Cray 2 (scaled to microseconds for a single operation). Buell and Ward did limit themselves to numbers of 100 decimal digits in the timings (although their package allows numbers of up to about 440 digits). It should be kept in mind that the timings by Buell and Ward are for a Cray 2 (with a 4.1 ns clock cycle) while our timings are for a Cray Y-MP (with a 6 ns clock cycle).

digits	time	Buell and Ward
20	11.8	13.0
30	11.7	13.1
40	11.6	13.2
50	11.4	13.1
60	11.3	13.3
70	12.7	13.5
80	11.0	14.6
90	10.9	13.8
100	10.6	13.6
200	9.3	
300	14.0	
400	12.7	
500	17.4	
600	16.0	
700	22.4	
800	19.5	
900	24.1	
1000	22.8	

It can be seen that for number sizes up to about 200 digits the timing is remarkably flat. The limit for a single block of 32 words on the Cray is 220 digits, so that is not very surprising. The reason the time decreases when the length of the number increases is because there is some additional overhead to deal with non-full numbers.

5. Timings for multiplication

Here we give a table of timings for the multiplication. Also here the examples are similar to those of Buell and Ward.

digits	time	Buell and Ward
20	14.5	21.2
30	15.1	27.3
40	15.1	30.9
50	16.0	34.3
60	16.2	41.7
70	16.7	45.6
80	17.0	50.5
90	18.2	57.5
100	21.1	61.0
200	29.9	
300	94.7	
400	104.2	
500	231.1	
600	229.8	
700	409.8	
800	410.5	
900	654.5	
1000	643.4	

Also in this table we see the effect of splitting the integers in blocks of 32 words. But there is more to it. Because the multiplication has to be general, it is important that intermediate data should not overwrite the result array. In a previous version of the routine two data array's were allocated at the start of the routine and deallocated at the end. But it appears that about 20 microseconds are needed for this allocation and deallocation, which seems a bit large. So the current scheme maintains global work arrays that are allocated at the first entry of the routine, probably reallocated if the size is too small for a specific call; but these two arrays are never deallocated. A problem might be that those two arrays remain being allocated and so use memory space.

6. Timings for division

Finally in this section we give timings for the division. The table comes in two parts, the first part gives times for dividing a 100 digit number by another number with a size ranging from 20 to 100 digits, the second part gives times for dividing a 1000 digit number by another number with a size ranging from 200 to 1000 digits. The second part does not have accompanying figures from Buell and Ward.

Dividend of 100 digits:

digits	time	Buell and Ward
20	118.4	201
30	102.1	222
40	99.7	226
50	100.5	231
60	82.7	218
70	80.8	203
80	69.8	183
90	69.3	144
100	57.9	116

Dividend of 1000 digits:

digits	time
200	923.3
300	1155.3
400	1029.4
500	1272.9
600	889.6
700	1181.2
800	653.2
900	489.9
1000	174.1

It can be seen that the code works especially fast if the quotient is small. It should be noted that dividend and divisor are chosen such that the quotient is never zero, as that would trigger a short cut in the code.

7. Conclusion

This package appears to perform its operations faster than Buell and Ward's. Also its field of applicability is larger as there is no limit on the numbers manipulated. The performance is fairly flat for number ranges that lie within a specific range of *long integers*, as can be expected. One problem found is that memory allocation and deallocation takes very much time, and a workaround had to be found.

Acknowledgements

Financial support for this work was provided by Cray Research, Inc. under a 1991 University Research & Development Grant for the project 'A Long Integer Package for the Cray Y-MP4'. This work was sponsored by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organization for Scientific Research, NWO).

APPENDIX

1. Introduction

In this appendix we describe the current contents of the long integer package. There are three levels of routines. The first level deals with long integers in a standard way. They allow integers of arbitrary size. The second level is intended for use when modular arithmetic is performed using a fixed modulus. The third level is for the same use as the second level, but uses the transformation of Peter Montgomery on the numbers. This will speed up many operations, but is a bit more tricky to use. The current implementation is in C with assistance of code written in Assembler (CAL); interface routines to call these routines from Fortran are available.

2. Level 1

We describe the routines and type used in a number of subsections.

2.1. Type

There is one type defined:

```
typedef struct {
        int sign;
        int length;
        int allocated_length;
        int *contents;
} mp_int;
```

This type is the standard type used by the routines in the package. The fields *sign*, *length* and *contents* describe the value of the number, the field *allocated_length* indicates the amount of memory allocated for the number.
It is **not** sufficient to declare long integers with this type. One should initialize the variable with the routine from the next section.
Also, the intention of this type is to be opaque, i.e. one should not manipulate the fields directly, but one should use the routines provided.

2.2. Initialization

The routines below initialize a long integer:

```
void mp_allocate_space(mp_int *var, int nr_bits);
```

This routine will initialize the long integer *var* such that space is allocated to hold at least *nr_bits* binary digits.

```
void mp_reallocate_space(mp_int *var, int nr_bits);
```

This routine is similar to *mp_allocate_space*, except that the long integer *var* should already have space allocated. This routine may be used to increase the size of the long integer or to decrease its size. In both cases the old contents are copied to the newly allocated space. In case the size is decreased the high order part of the value will silently be truncated if it does not fit in the newly allocated size. The primary intention of this routine is for use in conjunction with the error control routines described below.

void mp_free(mp_int *var);
This routine frees the space previously allocated for *var*. This routine should be used when leaving a procedure that declares mp variables and allocates space for them. Otherwise that space is never reused.

2.3. Error Control

The routines below defines what should be used if the result of an operation does not fit in the long integer variable provided:

void (*mp_overflow)(mp_int *var, int needed_size);
This pointer to a routine is used by the package when overflow is detected (i.e. the result of an operation can not be represented in the space allocated). The actual routine is provided with two parameters. The first parameter *var* is the long integer that is about to overflow, the second parameter *needed_size* is the space needed for the result in bits. When the actual routine wishes to increase the space used by the long integer, it should ensure that the old contents of *var* are carried over to the newly created space. For this purpose it is best to use the routine *mp_reallocate_space* from the previous section. The default value for this pointer is *mp_overflow_trap*. One can assign *mp_reallocate_space* directly to this pointer, in which case no overflow is fatal.

void mp_overflow_trap(mp_int *var, int needed_size);
The standard function assigned to *mp_overflow*. It will trap and abort the program through *mp_terminate*.

The next routines define what should be done when an error is detected:

void (*mp_terminate)();
This pointer to a routine is used by the package when it discovers an unrecoverable error. Normally the routine *mp_exit* is assigned to this pointer, but other (user defined) routines may be assigned as well.

void mp_exit();
Terminates the program with exit status of 1. This is the standard function assigned to *mp_terminate*.

2.4. Comparisons

The routines below compare two long integers or compare a long integer with a normal integer:

int mp_eq(mp_int val1, val2);
int mp_ne(mp_int val1, val2);
int mp_gt(mp_int val1, val2);
int mp_ge(mp_int val1, val2);
int mp_le(mp_int val1, val2);
int mp_lt(mp_int val1, val2);
These routines compare the long integers *val1* and *val2*. The comparison involved is part of the name. A non-zero value is returned for a *true* result, a zero value is returned for a *false* result.

int mp_eq_int(mp_int val1, int val2);
int mp_ne_int(mp_int val1, int val2);
int mp_gt_int(mp_int val1, int val2);
int mp_ge_int(mp_int val1, int val2);
int mp_le_int(mp_int val1, int val2);
int mp_lt_int(mp_int val1, int val2);

These routines compare the long integer *val1* with the normal integer *val2*; in all other aspects they are similar to the routines above.

2.5. Conversion

The routines below do some standard conversions:

void mp_store_int(int val, mp_int *res);

This routine stores the value of the normal integer *val* into the long integer *res*.

int mp_value(mp_int val);

This routine retrieves the value of the long integer *val* and returns that value as a normal integer. If the value does not fit in a normal integer the routine aborts the program.

float mp_float_value(mp_int val, int *scale);
double mp_double_value(mp_int val, int *scale);

These routines convert the value of the long integer *val* to a floating point number. The result is returned partly as the function result and partly in the second parameter *scale*. The function result is either a single precision floating point number or a double precision floating point number. The result in *scale* indicates a scale factor that should be applied to the function result to get the actual value. The scale factor is 10 to the power *scale*. The scale factor is present because otherwise the result might overflow; its base is 10 for ease of manual dissemination.

2.6. Addition and subtraction

The routines below add or subtract two numbers:

void mp_add(mp_int val1, val2, *res);

This routine adds the long integer values stored in *val1* and *val2* and return the sum in *res*. The actual parameters can all be the same.

void mp_sub(mp_int val1, val2, *res);

This routine subtracts the long integer value stored in *val2* from the long integer value stored in *val1* and returns the difference in *res*. The actual parameters can all be the same.

void mp_neg(mp_int val, *res);

This routine negates the value stored in *val* and stores the result in *res*. The two parameters can be the same.

void mp_abs(mp_int val, *res);

This routine takes the absolute value of *val* and stores the result in *res*. The two parameters can be the same.

void mp_add_int(mp_int val1, int val2, mp_int *res);

This routine adds the integer *val2* to the long integer stored in *val1* and stores the sum in *res*.

val1 and *res* can be the same actual parameters.

void mp_sub_int(mp_int val1, int val2, mp_int *res);
This routine subtracts the integer *val2* from the long integer stored in *val1* and stores the difference in *res*. *val1* and *res* can be the same actual parameters.

2.7. Multiplication

void mp_mul(mp_int val1, val2, *res);
This routine multiplies the long integers provided in *val1* and *val2* and stores the result in *res*. The actual parameters for *val1* and *val2* can be the same, but the actual parameter for *res* must be different from the first two.

void mp_mul_int(mp_int val1, int val2, mp_int *res);
This routine multiplies the long integer in *val1* and the integer *val2* and stores the result in *res*. The actual parameters *val1* and *res* can be the same.

2.8. Division and remaindering

The following routines calculate quotients and remainders after a division. Division truncates in the direction of 0. Remainder is defined such that

quotient * *divisor* + *remainder* = *dividend.*

Hence the remainder has the same sign as the dividend. There are also routines that calculate the modulus; the modulus has always the same sign as the divisor. When remainder and modulus are not equal their difference is the value of the divisor.

void mp_divrem(mp_int val1, val2, *qot, *rem);
This routine divides the long integer *val1* by the long integer *val2* and returns the quotient in *qot* and the remainder in *rem*. The actual parameters can be the same, except that *qot* and *rem* can not be the same.

void mp_div(mp_int val1, val2, *qot);
void mp_rem(mp_int val1, val2, *rem);
These routines act like *mp_divrem* except that only one result is returned.

void mp_mod(mp_int val1, val2, *mod);
The mod function. Similar to *mp_rem* except when *mp_rem* delivers a result that does not have the same sign as the divisor; in that case the divisor is added or subtracted, but see the introduction to this subsection.

int mp_divrem_int(mp_int val1, int val2, mp_int *qot);
This routine divides the long integer *val1* by the integer *val2* and returns the quotient in *qot* and the remainder as the function result. The actual parameters can be the same.

void mp_div_int(mp_int val1, int val2, mp_int *qot);
int mp_rem_int(mp_int val1, int val2);
int mp_mod_int(mp_int val1, int val2);
These routines are similar to *mp_div*, *mp_rem* and *mp_mod* except that the dividend is a normal integer, and the remainder or modulus result is returned as the function value.

2.9. Miscellaneous

The routines below perform some miscellaneous arithmetic operations:

int mp_log2(mp_int val);
: This routine calculates the floor of the base two logarithm of *val*. That is the number of significant bits in *val* minus one. If *val* is zero, –1 is returned.

int mp_pow2(mp_int val);
: This routine calculates the largest power of two that evenly divides *val*. That is the number of trailing zero bits in *val*. If *val* is zero, –1 is returned.

int mp_sign(mp_int val);
: This routine returns the signum of *val*, –1 for a negative value, 0 for a zero value and +1 for a positive value.

int mp_odd(mp_int val);
: This routine returns 1 (*true*) if the value represented by *val* is odd, 0 (*false*) otherwise.

void mp_gcd(mp_int val1, val2, *res);
: This routine calculates the greatest common divisor of long integers *val1* and *val2* and returns the result in *res*. The actual parameters can be the same.

void mp_lcm(mp_int val1, val2, *res);
: This routine calculates the least common multiple of long integers *val1* and *val2* and returns the result in *res*. The actual parameters can be the same.

void mp_inv(mp_int val1, val2, *res);
: This routine calculates the inverse of *val1* mod *val2* and stores the result in *res*. This means that
val1 * *res* = 1 mod *val2*.

void mp_sqrt(mp_int val1, *res);
: This routine calculates the integer square root of *val1* and stores the result in *res*. The integer square root is defined as the largest integer such that the square of it is not larger than *val1*.

2.10. Input and output

void mp_fprint(FILE *fp, mp_int val);
: This routine prints the value in long integer *val* on file *fp*. During printing numbers may be split along multiple lines. Intermediate lines are always terminated by a backslash.

void mp_fread(FILE *fp, mp_int *val);
: This routine reads a long integer from file *fp* and stores the value in *val*. First leading white space on the file is skipped until the start of a long integer is found (the start is a plus or a minus sign or a digit). From that point symbols are read that either are the (optional) starting sign or digits. A backslash followed by a newline read on input is ignored and reading continues on the following line. Likewise, whitespace (except newlines not preceded by a backslash) is ignored. Also intermediate commas are ignored (they can be used as thousands separators). Reading digits terminates when a non-escaped new-line is found or when a non-digit is found. The number so read is converted and the value is stored in *val*.

void mp_sprint(char *string, mp_int val);
This routine will convert the number represented by *val* to an ASCII string in the parameter *string*. It is important that the character array is long enough; no checking is performed.

void mp_sread(char *string, mp_int *val);
This routine will convert the number represented by the ASCII string stored in the parameter *string* and will store the result in *val*. The character array must start with either a sign or a digits; reading continues until a non-digit is found.

3. Level 2

This section describes the routines specific for modular arithmetic. In these routines the modulus used is implied and not passed as a parameter. These routines are useful if much arithmetic must be done modulo a single number. Parameters are not required to fall in a specific range, only the result is reduced to the base range. On the other hand, a number of routines will be faster if the operands are also in the base range.

3.1. Initialization etc.

The following routines are used to define c.q. retrieve the modulus used:

void mp_setmod(mp_int val);
Set the modulus to be used to the number represented by *val*.

void mp_getmod(mp_int *val);
Retrieve the modulus used and store the value in *val*.

void mp_reduce_modn(mp_int val1, mp_int *res);
Reduce *val1* modulo the preset modulus and store the result in *res*. This routine is provided so that some calculations can be speeded up a bit. It is not always necessary to reduce the result of every operation. For instance, the expression:
$e = (a * b + c * d) \bmod n$
can be expressed through the following sequence:

```
mp_mul_modn(a, b, &tmp1);
mp_mul_modn(c, d, &tmp2);
mp_add_modn(tmp1, tmp2, &e);
```

but is more efficient than:

```
mp_mul(a, b, &tmp1); /* Note: no reduction on the first statements */
mp_mul(c, d, &tmp2);
mp_add(tmp1, tmp2, &tmp1);
mp_reduce_modn(tmp1, &e);
```

3.2. Arithmetic

The following routines are similar to the routines described in section 2; the only difference is that the results are always modulo the modulus defined by *mp_setmod* (and hence always positive). There is no restriction on the input operands (they can be negative or larger than the modulus):

```
void mp_add_modn(mp_int val1, val2, *res);
void mp_sub_modn(mp_int val1, val2, *res);
void mp_neg_modn(mp_int val1, *res);
```

void mp_add_int_modn(mp_int val1, int val2, mp_int *res);
void mp_sub_int_modn(mp_int val1, int val2, mp_int *res);
void mp_mul_modn(mp_int val1, val2, *res);
void mp_mul_int_modn(mp_int val1, int val2, mp_int *res);
void mp_inv_modn(mp_int val1, *res);
void mp_div_modn(mp_int val1, val2, *qot);
For a description of these routines, see the descriptions in section 2 on non-modular arithmetic.

4. Level 3

This section describes the routines specific for modular arithmetic when the representation from Peter Montgomery is used. Also in these routines the modulus used is implied and not passed as a parameter. These routines are useful if much arithmetic must be done modulo a single number, where many operations are multiplications. The representation allows a fast multiplication modulo a fixed number. A disadvantage is that all numbers should be converted to and from that representation; also not all operations are readily available. Parameters must fall in a specific range, however the routines do no checking. And of course the parameters must be in the representation.

4.1. Initialization etc.

The following routines are used to define c.q. retrieve the modulus used:

void mp_setmod_mg(mp_int val);
Set the modulus to be used to the number represented by *val*. This routine implicitly sets the modulus for the Level 2 routines.

void mp_getmod_mg(mp_int *val);
Retrieve the modulus used and store the value in *val*.

void mp_to_mg(mp_int val1, mp_int *res);
Convert the value presented in *val1* to the correct representation. The result is returned in *res*.

void mp_from_mg(mp_int val1, mp_int *res);
Convert *val1* (which is a number in the Montogomery representation) back to a standard number in *res*.

4.2. Arithmetic

The following routines are similar to the routines described in section 2 and section 3; the difference is that the operands and the parameters must be in Montgomery representation.

void mp_add_mg(mp_int val1, val2, *res);
void mp_sub_mg(mp_int val1, val2, *res);
void mp_mul_mg(mp_int val1, val2, *res);
For a description of these routines, see the descriptions in section 2 on non-modular arithmetic.

5. Fortran support

In this section we describe the parameters and routine names that should be used when calling the package from a Fortran program. The first subsection gives the difference between the parameter use in C and

Fortran, the second subsection gives the different routines for space allocation and miscellaneous use while the last subsection indicates for the arithmetic routines how they are to be called from Fortran.

5.1. Parameter use

Contrary to the C version, a long integer is not represented by a compound type but by a single integer for obvious reasons. There is one drawback: it is more difficult to safeguard against errors, i.e., the integers used to represent long integers should not be used in assignments of one or another form. Actually these integers are pointers pointing to a compound structure, but this is not expressible in Fortran.

In all places where the C routines specify a parameter of type 'mp_int' or of type '*mp_int', in Fortran the integer should be passed. In all other places where the C routines specify a parameter of pointer type, in Fortran a plain parameter must be passed.

Apart from this, and from the naming as indicated in the third subsection, the only differences are found in the next subsection.

5.2. Space allocation and miscellaneous

Below follow the routines that have different calling conventions between C and Fortran:

mp_allocate_space:

```
SUBROUTINE MPALSP(MPINT, NBYTES)
INTEGER MPINT, NBYTES
```

mp_reallocate_space:

```
SUBROUTINE MPRESP(MPINT, NBYTES)
INTEGER MPINT, NBYTES
```

mp_free:

```
SUBROUTINE MPFREE(MPINT)
INTEGER MPINT
```

mp_overflow:

```
SUBROUTINE MPSOVF(N)
INTEGER N
```

Contrary to C error checking there is only one routine to support overflow checking. The parameter N indicates whether overflow checking has to be performed (N .NE. 0) or not (N .EQ. 0). When no overflow checking is performed the system will silently increase the space allocated for the long integers.

mp_fprint:

```
SUBROUTINE MPPRNT(U, MPINT)
INTEGER U, MPINT
```

The parameter U indicates the unit where the number should be printed.

mp_fread:

```
SUBROUTINE MPREAD(U, MPINT)
INTEGER U, MPINT
```

The parameter U indicates the unit where the number should be printed.

5.3. Arithmetic routines

mp_eq, mp_ne, mp_gt, mp_ge, mp_le and mp_lt:

```
      INTEGER FUNCTION MPEQ(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPNE(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPGT(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPGE(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPLE(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPLT(A, B)
      INTEGER A, B
```

mp_eq_int, mp_ne_int, mp_gt_int, mp_ge_int, mp_le_int and mp_lt_int:

```
      INTEGER FUNCTION MPEQI(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPNEI(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPGTI(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPGEI(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPLEI(A, B)
      INTEGER A, B

      INTEGER FUNCTION MPLTI(A, B)
      INTEGER A, B
```

mp_store_int:

```
      SUBROUTINE MPSETI(A, B)
      INTEGER A, B
```

mp_value, mp_float_value and mp_double_value:

```
      INTEGER FUNCTION MPIVAL(A)
      INTEGER A

      FLOAT FUNCTION MPFVAL(A)
      INTEGER A
```

```
      DOUBLE PRECISION FUNCTION MPDVAL(A)
      INTEGER A
```

mp_copy:

```
      SUBROUTINE MPCOPY(A, B)
      INTEGER A, B
```

mp_add and mp_sub:

```
      SUBROUTINE MPADD(A, B)
      INTEGER A, B

      SUBROUTINE MPSUB(A, B)
      INTEGER A, B
```

mp_neg and mp_abs:

```
      SUBROUTINE MPNEG(A, B)
      INTEGER A, B

      SUBROUTINE MPABS(A, B)
      INTEGER A, B
```

mp_add_int and mp_sub_int:

```
      SUBROUTINE MPADDI(A, B, C)
      INTEGER A, B, C

      SUBROUTINE MPSUBI(A, B, C)
      INTEGER A, B, C
```

mp_mul and mp_mul_int:

```
      SUBROUTINE MPMUL(A, B, C)
      INTEGER A, B, C

      SUBROUTINE MPMULI(A, B, C)
      INTEGER A, B, C
```

mp_divrem, mp_div, mp_rem and mp_mod:

```
      SUBROUTINE MPDVR(A, B, C, D)
      INTEGER A, B, C, D

      SUBROUTINE MPDIV(A, B, C)
      INTEGER A, B, C

      SUBROUTINE MPREM(A, B, C)
      INTEGER A, B, C

      SUBROUTINE MPMOD(A, B, C)
      INTEGER A, B, C
```

mp_divrem_int, mp_div_int, mp_rem_int and mp_mod_int:

```
      SUBROUTINE MPDVRI(A, B, C, D)
      INTEGER A, B, C, D
```

```
      SUBROUTINE MPDIVI(A, B, C)
      INTEGER A, B, C

      SUBROUTINE MPREMI(A, B, C)
      INTEGER A, B, C

      SUBROUTINE MPMODI(A, B, C)
      INTEGER A, B, C
```

mp_log2 and mp_pow2:

```
      INTEGER FUNCTION MPLOG2(A)
      INTEGER A

      INTEGER FUNCTION MPPOW2(A)
      INTEGER A
```

mp_sign and mp_odd:

```
      INTEGER FUNCTION MPSIGN(A)
      INTEGER A

      INTEGER FUNCTION MPODD(A)
      INTEGER A
```

mp_gcd and mp_lcm:

```
      SUBROUTINE MPGCD(A, B, C)
      INTEGER A, B, C

      SUBROUTINE MPLCM(A, B, C)
      INTEGER A, B, C
```

mp_inv and mp_sqrt:

```
      SUBROUTINE MPINV(A, B, C)
      INTEGER A, B, C

      SUBROUTINE MPSQRT(A, B, C)
      INTEGER A, B, C
```

mp_setmod and mp_getmod:

```
      SUBROUTINE MNSMOD(A)
      INTEGER A

      SUBROUTINE MNGMOD(A)
      INTEGER A
```

mp_reduce_modn:

```
      SUBROUTINE MNRED(A, B)
      INTEGER A, B
```

mp_add_modn and mp_sub_modn:

```
      SUBROUTINE MNADD(A, B, C)
      INTEGER A, B, C
```

```
        SUBROUTINE MNSUB(A, B, C)
        INTEGER A, B, C

mp_neg_modn:
        SUBROUTINE MNNEG(A, B)
        INTEGER A, B

mp_add_int_modn and mp_sub_int_modn:
        SUBROUTINE MNADDI(A, B, C)
        INTEGER A, B, C

        SUBROUTINE MNSUBI(A, B, C)
        INTEGER A, B, C

mp_mul_modn and mp_mul_int_modn:
        SUBROUTINE MNMUL(A, B, C)
        INTEGER A, B, C

        SUBROUTINE MNMULI(A, B, C)
        INTEGER A, B, C

mp_inv_modn and mp_div_modn:
        SUBROUTINE MNINV(A, B)
        INTEGER A, B

        SUBROUTINE MNDIV(A, B)
        INTEGER A, B

mp_setmod_mg and mp_getmod_mg:
        SUBROUTINE MGSMOD(A)
        INTEGER A

        SUBROUTINE MGGMOD(A)
        INTEGER A

mp_to_mg and mp_from_mg:
        SUBROUTINE MGTO(A, B)
        INTEGER A, B

        SUBROUTINE MGFROM(A, B)
        INTEGER A, B

mp_add_mg, mp_sub_mg and mp_mul_mg:
        SUBROUTINE MGADD(A, B, C)
        INTEGER A, B, C

        SUBROUTINE MGSUB(A, B, C)
        INTEGER A, B, C

        SUBROUTINE MGMUL(A, B, C)
        INTEGER A, B, C
```