AR2: A system for doing arithmetic on the PC

Tony Forbes, October 1999, [2.2.2E]

Introduction

AR2 is a collection of routines for performing exact calculations with integers. The largest number we can handle is somewhere in the region of ${}_2 2^{36}$ (about $10^{20686623780}$), but in real life you will probably run out of memory long before that limit is reached. The system consists of C functions, each of which is described in detail below. The package is fast, mainly because we have used assembler language for the low level routines, which provide the fundamental operations on large integers, such as move, add, subtract, compare, shift, multiply and divide.

There is some logic in the naming of our main routines. The first letter is either i, u, or x. Then there is a keyword, such as 'mov', 'add', 'sub', and so on, to indicate what sort of operation it is performing. Finally, sometimes there are extra 'modifiers'; for example, 's' indicates signed operands, 'k' means that one of the operands is an ordinary 32-bit integer, '2k' means that one of the operands is 2^k for $k < 2^{32}$.

Comments and bug reports welcome. E-mail: tonyforbes@ltkz.demon.co.uk

History

2.1.0 Initial version, loosely modelled on a QBASIC program.

2.1.1 New routine: xnxtprmk(k) returns next prime after k if $k < 2^{32}$, 1 otherwise.

2.1.2 New routine: igcdext(u, d, a, b) returns $d = \gcd(a, b)$ and $u = d/a \pmod b$.
Five temporary registers added to AR2I.C and AR2I.H.
Bug that gives incorrect remainder in idiv when $2^{31} \le$ divisor $< 2^{32}$ has been corrected.

2.1.3 New routine, idispf(f, a), writes decimal digits of a to file f.

2.1.4 Routines can now be used under DOS as well as Windows.
AR2XDOS.C provides DOS routines to replace Windows _dw routines.
Definitions for 'stop' and 'GiveUp' replaced by routines stop() and give_up().

2.1.5 New functions:
xsqumodF12(a, b) computes $a = b^2 \pmod{F_{12}}$.
isqufftC0(a) computes $a = a^2 \pmod{F_{12}}$ using FFT mod F_{12}.

2.2.0 Support for Windows 3.1 withdrawn. All _dw routines removed.
New functions: ieval(e) evaluates the expression e;
icmpk(a, k) compares a and k , $-2^{31} \le k < 2^{31}$;

iwd0(a) = $|a|$ mod 2^{32}, the low-order word of a;
ieven(a), iodd(a), ilen(a), ibit(a), ilog(a), isqrt(a, b).

2.2.1 New function; inxtprm(a): a => next prime after a if a is not too large. Bug in imul corrected; bug in isubk corrected; variables in ar2.c and ar2p.c declared as 'static'.

2.2.2 Bug! Routine isqrt crashes with 'invalid parameter in idiv' for numbers around 2^{500}. Corrected. We were not ensuring that the first guess is non-negative.

There are still problems with imul and isqu. The 'traditional' FFT routines do not work well with integers greater than about $2^{1000000}$. This is presumably due to lack of precision. The dimension of the FFT is at least 2^{17} and we are using 53-bit precision; $17 + 2\times16 = 49$, $2(17 + 16) = 66$.

Integer registers

The i- and u-routines operate on special objects which we call *integer registers*. These are structures, defined in C as follows.

```
struct IntegerRegister
    {
    long  *value;
    long  capacity;
    long  digits;
    long  flags;
    };
typedef struct IntegerRegister ireg;
```

***value**: Pointer to the units digit of the array containing the integer.

value: Array of 32-bit integers, a_0, a_1, ..., a_{d-1}, representing the non-negative number a, where

$$a = a_0 + 2^{32}a_1 + ... + 2^{32(d-1)}a_{d-1}, \qquad 0 \le a_0, a_1, ..., a_{d-1} < 2^{32}$$

a–>value[0] = a_0
a–>value[1] = a_1
...
a–>value[d–1] = a_{d-1}, $d = d(a) = a$–>digits.

There are two alternative ways of dealing with negative numbers.

(i) Negative a is stored as $2^{32d(a)} + a$. The range is $-2^{32*d(a)-1}$ to $2^{32*d(a)-1} - 1$. Not all the routines recognise these negative numbers. Those that don't just treat them as large

positive numbers. This method is possibly useful in applications involving numbers of some fixed size, and where performance is particularly important for addition and subtraction.

(ii) Negative *a* is stored as |*a*| and the NegativeReg bit is set in *a*–>flags. Only the i-routines recognise these negative numbers. The x- and u-routines do not.

capacity: Maximum number of 32-bit digits that integer array *a*–>value can hold. This is the number of words allocated to the array when it was created.

digits: Number of digits, $d = d(a)$ in base 2^{32}. The high order digit of the integer is *a*–>value[*a*–>digits – 1], the low order digit is *a*–>value[0].

flags: Status flags. (Used only by the i-routines.) NegativeReg bit indicates that the value of the integer register is $-a$ rather than a. Thus a is negative iff $a \neq 0$ and *a*–>flags has the NegativeReg bit is set.

i-routines

The i-routines tend to be more robust and there are not so many restrictions. They operate on integer registers, they take signs into account and have mechanisms for dealing with overflow resulting from addition and subtraction. The i-routines also make sure that the integer arrays have sufficient space for the extra digits required for certain operations, such as multiplication.

Integer registers are usually set up by routines 'imov', 'imovk' and 'imovd'. In general, they are non-decreasing in size. More space is allocated whenever needed. Some operations remove leading zeroes from their operands, but the result may acquire leading zero digits, for example, by subtracting large, nearly equal numbers. We usually do not remove non-significant zeroes from the result.

Negative numbers are represented by sign-and-magnitude format. Zero can be positive or negative. We usually do not remove the minus sign from a minus-zero result. The rules for division depend on whether it is a division with quotient, or a modulo operation. The divisor must be positive. Routines 'idiv', 'idivk' and 'idiv2k' return the remainder and the quotient with the same sign as that of the dividend. However, 'imod', 'imodk' and 'imod2k' always return a non-negative result; thus, 'mod(a, b)' returns a congruent to a (mod b), $0 \leq a < b$.

At low level many of the computations are performed in-place. Hence, in general, the integer array where the result is created must not overlap with any other operand. If there is overlap, then the results may be incorrect, as, for example, with 'imul', which expects the multiplier operand to remain intact throughout the computation of the product. The only possible exception is where the same integer register is specified in more than one operand. We will say so very clearly when (and only when) this is permissible.

There are certain global objects used by the i-routines. Integer register *res* is used primarily to return the remainder after a division operation. Similarly, *kres* is a 32-bit integer that contains the remainder after division by a 32-bit integer. Some of the higher mathematical routines use *res* as a work-area. Global variables *FFTsqu* and *FFTmul* determine the cut-over point from the 'school' method to the fast Fourier transform method for 'isqu' and 'imul', respectively.

void imov (ireg *_a_, ireg *_b_)

Set $a = b$. Note that the target, *a*, does not have to be initialised. It simply becomes a copy of *b*, together with any leading zeroes *b* might have.

void imovk (ireg *_a_, long _k_)

Set $a = k$, where *k* is an ordinary, signed, 32-bit integer, $-2^{31} < k < 2^{31}$. The target, *a*, does not have to be initialised. The number of digits is set to 1.

void imovd (ireg *_a_, char *_c_)

Set integer register *a* equal to the value of the string of decimal digits *c*. The string *c* can begin with a minus sign (-) to denote a negative number. If *c* begins with an underscore (_), then it is assumed to be a continuation, and *a* is given the value it would have by appending the decimal digits of *c* to it. Thus imovd(*a*, "–12345") followed by imovd(*a*, "_6789") sets *a* equal to –123456789.

The target, *a*, does not have to be initialised, unless *c* begins with an underscore. The first non-decimal character (after the leading underscore or minus sign, if any) terminates the operation.

void iadd (ireg *_a_, ireg *_b_)

Add *b* to *a*. Integer registers *a* and *b* can be the same, in which case the result is $2a$.

void iaddk (ireg *_a_, long _k_)

Add *k* to *b*, where *k* is an ordinary, signed, 32-bit integer, $-2^{31} < k < 2^{31}$.

void isub (ireg *_a_, ireg *_b_)

Subtract *b* from *a*. Integer registers *a* and *b* can be the same, in which case the result is zero.

void isubk (ireg *_a_, long _k_)

Subtract *k* from *b*, where *k* is an ordinary, signed, 32-bit integer, $-2^{31} < k < 2^{31}$.

ineg(ireg * *a*)

Set $a = -a$.

iabs(ireg **a*)

Set $a = |a|$.

long isgn (ireg **a*)

Return –1 if $a < 0$, 0 if $a = 0$, 1 if $a > 0$.

long icmp (ireg **a*, ireg **b*)

Return –1 if $a < b$, 0 if $a = b$, 1 if $a > b$.

long icmpk (ireg **a*, long *k*)

Return –1 if $a < k$, 0 if $a = k$, 1 if $a > k$, where k is an ordinary, signed, 32-bit integer, $-2^{31} < k < 2^{31}$.

long ieven (ireg **a*)
long iodd (ireg **a*)

Return 1 if a is even, 0 if a is odd, or the other way round.

long ilen (ireg **a*)

Return $\max\{[\log(|a|)/\log(2)] + 1, 0\}$, the length of a in bits.

long ibit (ireg **a*, long *k*)

Return $[|a|/2^k] \bmod 2$, the value of bit k of a. Bit 0 is the units bit.

long iwd0 (ireg **a*)

Return $|a| \bmod 2^{32}$, the low-order word of a.

void isqu (ireg **a*)

Multiply *a* by itself. Integer register *a* is trimmed of leading zeroes first, then extended to twice as many digits. If *a* has *FFTsqu* digits or more, we take the Fourier transform of *a*, square term-by-term, then invert. If *a* has less then *FFTsqu* digits, the usual 'school' method is used.

void imul (ireg **a*, ireg **b*)

Multiply *a* by *b*. Integer register *a* and *b* are trimmed of leading zeroes first, then *a* is extended by the number of digits in *b*. If both *a* and *b* have *FFTmul* digits or more, we take the Fourier transforms of *a* and *b*, multiply term-by-term, then invert. Otherwise, the usual 'school' method is used. Operands *a* and *b* must not overlap; use isqu(*a*) instead of imul(*a*, *a*).

void imulk (ireg *a, long k)

Multiply *a* by the 32-bit integer *k*, $-2^{31} < k < 2^{31}$. Integer register *a* is trimmed of leading zeroes first, then extended by one digit. We recommend iadd(*a*, *a*), rather than imulk(*a*, 2); it's faster.

void imul2k (ireg *a, long k)

Multiply *a* by 2^k, where *k* is a 32-bit non-negative integer. Integer register *a* is trimmed of leading zeroes first, then extended by [(*k* + 31)/32] digits.

void idiv (ireg *a, ireg *b)

Divide *a* by *b*, putting the quotient in *a* and the remainder in the global integer register *res*. The divisor *b* must be positive. If *a* is negative, then the remainder will be non-positive. Hence the operation gives

$a = [a / b]$, $res = (a \bmod b)$, when $a \geq 0$,
$a = -[-a / b]$, $res = -(-a \bmod b)$, when $a < 0$.

Both integer registers *a* and *b* are trimmed of leading zeroes first.

void idivk (ireg *a, long k)

Divide *a* by *k*, where *k* is a positive, 32-bit integer, putting the quotient in *a* and the remainder in the global integer *kres*. The divisor *k* must be positive. If *a* is negative, then the remainder will be non-positive. Hence the operation gives

$a = [a / k]$, $res = (a \bmod k)$, when $a \geq 0$,
$a = -[-a / k]$, $res = -(-a \bmod k)$, when $a < 0$.

Integer register *a* is trimmed of leading zeroes first.

void idiv2k (ireg *a, long k)

Divide *a* by 2^k, where *k* is a non-negative 32-bit integer. The remainder is lost. As with the other divides, the operation gives

$a = [a / 2^k]$, when $a \geq 0$,

$a = -[-a / 2^k]$, when $a < 0$.

Integer register *a* is trimmed first.

void imod (ireg *a, ireg *b)

Reduce *a* modulo *b*, where $b > 0$. The result is always in the range $0 \le a < b$, whatever the sign of *a*. Both integer registers *a* and *b* are trimmed of leading zeroes first.

void imodk (ireg *a, long k)

Reduce *a* modulo *k*, where *k* is a 32-bit positive integer. The result is always in the range $0 \le a < k$, whatever the sign of *a*. Integer register *a* is trimmed of leading zeroes first.

void imod2k (ireg *a, long k)

Reduce a modulo 2^k, where *k* is a 32-bit non-negative integer. The result is always in the range $0 \le a < 2^k$, whatever the sign of *a*.

void iexp(ireg *a, ireg *b, ireg *c)

Set $a = b^c$ if $c \ge 0$, otherwise set $a = 1$. Integer arrays *b* and *c* may coincide or overlap, but not with *a*.

void iexpmod(ireg *a, ireg *b, ireg *c, ireg *q)

Set $a = b^c \pmod q$ if $c \ge 0$, otherwise set $a = 1$. The result is in the range $0 \le a < q$. Integer arrays *b*, *c* and *q* may coincide or overlap, but not with *a*.

void iexpmodm2ke(ireg *a, ireg *b, ireg *c, ireg *q, long m, long k, ireg *e)

On the assumption that $q = m2^k + e > 0$, set $a = b^c \pmod q$ if $c \ge 0$, otherwise set $a = 1$. Here, *m* and *k* are 32-bit integers, and *e* is an integer register. The result is in the range $0 \le a < q$. This routine should be significantly faster than 'iexpmod' if $|e| < 2^{k/2}$.

If *q* is not equal to $m2^k + e$, the result will be meaningless. Integer arrays *b*, *c*, *e* and *q* may coincide or overlap, but not with *a*. The value of the global integer register *res* and the global integer *kres* are destroyed in the process.

void igcd(ireg *a, ireg *b)

Set $a = \gcd(a, b)$. The result is non-negative. The value of *b* is destroyed in the process.

void igcdext(ireg *u, ireg *d, ireg *a, ireg *b)

Extended gcd. Set $d = \gcd(a, b)$, where $b > 0$. If $a = 0$, then set $u = 0$, otherwise set $u =$

$d/a \pmod b$, that is, u is one of the d integers satisfying $0 \le u < b$ and $ua \equiv d \pmod b$. Operands u and a may coincide, in which case a is replaced by the result; otherwise the values of a and b are preserved.

double ilog(ireg *a)

Return $\max\{\log(|a|), 0\}$.

void isqrt(ireg *a, ireg *b)

Set $a = [b^{1/2}]$ and $res = b - [b^{1/2}]^2$, where $b \ge 0$.

ireg * ieval(char *e)

Return a pointer to an integer register containing the value of the expression e. Thus, for example, 'imov(a, ieval("25*2^1024+1"));' sets a to $25.2^{1024} + 1$. If there were errors, the character pointer eval_err contains the address of an error-message; otherwise eval_err points to a NULL value. If e is string of decimal digits, eval_exp is set to 0; otherwise eval_exp is set to 1;

Valid symbols for e are decimal digits, '+', '–', '*', '/', '@' (modulo), '^' (exponentiation), '!', '#' (prime product), 'q' (integer square root) and brackets to a reasonable level. A semicolon or a NULL value terminates the expression. Blanks and controls (other than NULL) are ignored.

'+' and '–' can also be used as prefixes; '–' negates what follows, '+' does nothing. '\' can be used instead of '/' but in C it must be coded as '\\'. '/' is integer division with truncation towards zero; for example, 10/3 = 3, –10/3 = –3, $x/–y = –x/y$. '@' always gives a non-negative result; for example 10@3 = 1, –10@3 = 2, $x@–y = –x@y$. $x\#$ is the product of the primes $\le x$. q$x = [x^{1/2}]$. $x! = 1$ if $x < 1$, $x\# = 1$ if $x < 2$. Exponentiation is left to right (for example, $2\text{^}3\text{^}3 = 8^3 = 512$, not 2^{27}).

The order of precedence is '(' and ')'; '!' and '#'; prefix '+', prefix '–' and 'q'; '^'; '*', '/' and '@'; '+' and '–'.

void inxtprm (ireg *a)

If $a < 1041230445623$, replace a with the next prime after a. Otherwise replace a with the next positive integer which is coprime to 1020401!. This routine uses the Sieve of Eratosthenes and on average it is much faster than xnxtprmk(k), provided that it is invoked repeatedly and often to obtain primes consecutively. Every so often there is a slowing-down while 'inxtprm' sieves a batch of 2560000 numbers, but for most calls it just has to find the next unsieved number. The largest prime used for sieving is 1020401.

void idisp (ireg *a)

Display the decimal value of an integer register. There is no carriage return. The values of the global integer register *res* and the global integer *kres* are destroyed in the process.

void idispf (FILE *f, ireg *a)

Write the decimal value of an integer register to file *f*. There is no carriage return. The values of the global integer register *res* and the global integer *kres* are destroyed in the process.

void isqufft77 (ireg *a)

Compute $a = a^2$ using FFT. If *a* has 120 digits or less, *a* is extended to 242 digits, the global integer register *res* (which is used as a work-area) is extended to 645 digits and a^2 is computed with the 128-dimensional fast Fourier transform modulo F_7. If *a* has more than 120 digits, 'isqu' is used.

void isqufft88 (ireg *a)

Compute $a = a^2$ using FFT. If *a* has 480 digits or less, *a* is extended to 962 digits, the global integer register *res* (which is used as a work-area) is extended to 2313 digits and a^2 is computed with the 256-dimensional fast Fourier transform modulo F_8. If *a* has more than 480 digits, 'isqu' is used.

void isqufft89 (ireg *a)

Compute $a = a^2$ using FFT. If *a* has 960 digits or less, *a* is extended to 1922 digits, the global integer register *res* (which is used as a work-area) is extended to 4617 digits and a^2 is computed with the 512-dimensional fast Fourier transform modulo F_8. If *a* has more than 960 digits, 'isqu' is used.

void isqufft8A (ireg *a)

Compute $a = a^2$ using FFT. If *a* has 1920 digits or less, *a* is extended to 3842 digits, the global integer register *res* (which is used as a work-area) is extended to 9225 digits and a^2 is computed with the1024-dimensional fast Fourier transform modulo F_8. If *a* has more than 1920 digits, 'isqu' is used.

void isqufft80 (ireg *a)

Compute $a = a^2$ using FFT. If *a* has 1920 digits or less, *a* is extended to four times its size, or 3842 digits, whichever is less, the global integer register *res* (which is used as a work-area) is extended to 9225 digits, and a^2 is computed with the fast Fourier transform modulo F_8, choosing the best dimension from the set {32, 64, 128, 256, 512, 1024}. If *a* has more than 1920 digits, 'isqu' is used.

void isqufftA0 (ireg *a)

Compute $a = a^2$ using FFT. If a has 32256 digits or less, a is extended to four times its size, or 64514 digits, whichever is less, the global integer register *res* (which is used as a work-area) is extended to 135201 digits, and a^2 is computed with the fast Fourier transform modulo F_{10}, choosing the best dimension from the set {64, 128, 256, 512, 1024, 2048, 4096}. If a has more than 32256 digits, 'isqu' is used.

void isqufftC0 (ireg *a)

Compute $a = a^2$ using FFT. If a has 129024 digits or less, a is extended to four times its size, or 258048 digits, whichever is less, the global integer register *res* (which is used as a work-area) is extended to 528513 digits, and a^2 is computed with the fast Fourier transform modulo F_{12}, choosing the best dimension from the set {32, 64, 128, 256, 512, 1024, 2048, 4096}. If a has more than 129024 digits, 'isqu' is used.

u-routines

The u-routines also operate on integer registers. However, they do not concern themselves with tiresome details like signs, overflow and invalid parameters. Hence they are not as idiot-proof as the i-routines. They are provided as a convenient way of calling the x-routines using the integer register structure. You should be capable of dealing with signs and overflow. You must ensure that the operands are valid and that there is sufficient room for the result. In particular, you must avoid dividing by zero.

There are special u-routines for allocating, freeing, extending and trimming integer registers.

For an integer register a, we write $c(a)$ for the maximum number of digits a can hold, and $d(a)$ for the number of active digits.

void umovf (ireg *a, ireg *b)

$a = b$, where $d(a) = d(b) > 0$.

void umov (ireg *a, ireg *b)

$a = b$, where $d(a) \geq d(b) > 0$.

void umovs (ireg *a, ireg *b)

$a = b$, where $d(a) \geq d(b) > 0$; b is sign-extended if $d(a) > d(b)$.

void umovk (ireg *a, long k)

$a = k$, where $d(a) > 0$ and k is a non-negative, 32-bit integer; k is zero-extended if $d(a) > 1$.

void umovks (ireg *a, long k)

$a = k$, where $d(a) > 0$ and k is a 32-bit integer; k is sign-extended if $d(a) > 1$.

void umovz (ireg *a)

$a = 0$, where $d(a) > 0$.

void uaddf (ireg *a, ireg *b)

$a = a + b \pmod{2^{32d(a)}}$, where $d(a) = d(b) > 0$. Operands a and b can be the same.

long uadd (ireg *a, ireg *b)

$a = a + b \pmod{2^{32d(a)}}$, where $d(a) \geq d(b) > 0$; b is zero-extended if $d(a) > d(b)$. Returns the carry flag in bit 8 and the sign of the result in bit 31. Operands a and b can be the same.

long uadds (ireg *a, ireg *b)

$a = a + b \pmod{2^{32d(a)}}$, where $d(a) \geq d(b) > 0$; b is sign-extended if $d(a) > d(b)$. Returns the carry flag in bit 8 and the sign of the result in bit 31. Operands a and b can be the same.

long uaddk (ireg *a, long k)

$a = a + k \pmod{2^{32d(a)}}$, where $d(a) > 0$ and k is a non-negative, 32-bit integer; k is zero-extended if $d(a) > 1$. Returns the carry flag in bit 8 and the sign of the result in bit 31.

void usubf (ireg *a, ireg *b)

$a = a - b \pmod{2^{32d(a)}}$, where $d(a) = d(b) > 0$. Operands a and b can be the same.

long usub (ireg *a, ireg *b)

$a = a - b \pmod{2^{32d(a)}}$, where $d(a) \geq d(b) > 0$; b is zero-extended if $d(a) > d(b)$. Returns the carry flag in bit 8 and the sign of the result in bit 31. Operands a and b can be the same.

long usubs (ireg *a, ireg *b)

$a = a - b \pmod{2^{32d(a)}}$, where $d(a) \geq d(b) > 0$; b is sign-extended if $d(a) > d(b)$. Returns the carry flag in bit 8 and the sign of the result in bit 31. Operands a and b can be the same.

long usubk (ireg *a, long k)

$a = a - k \pmod{2^{32d(a)}}$, where $d(a) > 0$ and k is a non-negative, 32-bit integer; k is zero-extended if $d(a) > 1$. Returns the carry flag in bit 8 and the sign of the result in bit 31.

void uneg (ireg *a)

$a = 2^{32d(a)} - a$, the twos complement of a. We assume $d(a) > 0$.

long ucmp (ireg *a, ireg *b)

a and b are compared, the shorter being zero-extended, and the flag register is returned in bits 8 to 15. We assume $d(a)$, $d(b) > 0$.

long ucmps (ireg *a, ireg *b)

a and b are compared, the shorter being sign-extended, and the flag register is returned in bits 8 to 15. We assume $d(a)$, $d(b) > 0$.

long usig (ireg *a)

Returns the high-order digit of a. We assume $d(a) > 0$.

void usqu (ireg *a)

$a = a^2$ (using the 'school' method). We assume $c(a) \geq 2d(a) > 0$.

void umul (ireg *a, ireg *b)

$a = a * b$ (using the 'school' method). We assume $c(a) \geq d(a) + d(b)$, $d(a) > 0$, $d(b) > 0$.

void umulk (ireg *a, long k)

$a = a * k$, where k is a 32-bit integer. We assume $c(a) \geq d(a) + 1$, $d(a) > 0$.

void umul2k (ireg *a, long k)

$a = a * 2^k$, where k is a non-negative, 32-bit integer. We assume $c(a) \geq d(a) + [(k + 31)/32]$, $d(a) > 0$.

void umul2d (ireg *a, long d)

$a = a * 2^{32d}$, where d is a non-negative, 32-bit integer. We assume $c(a) \geq d(a) + d$, $d(a) > 0$.

void udiv (ireg *a, ireg *b, ireg *q)

$a = (a \bmod b)$, $q = [a / b]$. We assume $c(a) > d(a) \geq d(b) > 0$, $c(q) \geq d(a) - d(b) + 1$, $b > 0$.

long udivk (ireg *a, long k)

$a = [a / k]$, where k is a positive, 32-bit integer. Returns $(a \bmod k)$. We assume $d(a) > 0$.

void udiv2k (ireg *a, long k)

$a = [a / 2^k]$, where k is a non-negative, 32-bit integer. We assume $0 \leq k < 32d(a)$, $d(a) > 0$.

void udiv2d (ireg *a, long d)

$a = [a / 2^{32d}]$, where k is a non-negative, 32-bit integer. We assume $0 \leq d < d(a)$, $d(a) > 0$.

void umod (ireg *a, ireg *b)

$a = (a \bmod b)$. We assume $c(a) > d(a) \geq d(b) > 0$, $b > 0$.

void umodk (ireg *a, long k)

$a = (a \bmod k)$, where k is a positive, 32-bit integer. We assume $d(a) > 0$.

void usep2k (ireg *a, ireg *b, long k)

Separate a into two parts at bit k. Thus, $a = (a \bmod 2k)$ and $b = [a/2^k]$. We assume $0 \leq k < 32d(a)$, $d(a) > 0$, $c(q) \geq d(a) - [k/32]$.

void ualloc (ireg *a, long c, long d)

Allocate array memory for an integer register. Allocates c digits to integer register a and sets the first d digits to zero.

void ufree (ireg *a)

Free the array memory occupied by integer register a.

void uextend (ireg *a, long d)

Extend an integer register to d digits.

void utrim (ireg *a)

Remove non-significant digits from a.

void udump (ireg *a)

Unformatted print of an integer register.

x-routines

The x-routines are intended for internal use and most are written in assembler. They operate on ordinary integer arrays and must, where necessary, specify the size of each operand (as a number of digits in base 2^{32}) explicitly as extra parameters. There is no array bound checking of any kind. We always assume that the integer arrays are set up properly and that the digit parameters are sensible. For instance, zero number of digits is sometimes treated as 2^{32}, which is fine, if you don't mind a vast chunk of your memory getting clobbered.

We always assume that the result integer array has sufficient space to allow the operation to complete successfully. If not, things can go wrong. For example, adding 2^{254} to 2^{254} when digits = 8 results in a number that some routines treat as -2^{255}.

Multiplication requires high order space in the result array that is equal in size to the multiplier. If you have not left enough room, then the program will use the space anyway. It is not necessary for you to zeroise the space above the high-order digit of the multiplicand. The FFT square routines expect their own, particular amounts of space. For instance, xsqufft77 requires 968 bytes, regardless of the operand value.

The C compiler passes words to the assembler code in the order EAX, EDX, EBX, ECX; then any further parameters are placed on the stack.

The compiler insists that we do not alter any register that is not used for passing parameters. Conversely, we think we can do what we like with the registers that are used for passing parameters.

For an integer register *a*, we write *c*(*a*) for the maximum number of digits *a* can hold, and *d*(*a*) for the number of active digits.

void xmovf (long *a, long *b, long *da*)

a = *b*, where $d(a) = d(b) = da > 0$.

void xmov (long *a, long *b, long *da, long *db*)

a = *b*, where $d(a) = da \geq d(b) = db > 0$.

void xmovs (long *a, long *b, long *da, long *db*)

a = *b*, where $d(a) = da \geq d(b) = db > 0$; *b* is sign-extended if $d(a) > d(b)$.

void xmovk (long *a, long *k, long *da*)

$a = k$, where $d(a) = da > 0$ and k is a non-negative, 32-bit integer; k is zero-extended if $d(a) > 1$.

void xmovks (long *a, long k, long da)

$a = k$, where $d(a) = da > 0$ and k is a 32-bit integer; k is sign-extended if $d(a) > 1$.

void xmovz (long *a, long da)

$a = 0$, where $d(a) = da > 0$.

void xaddf (long *a, long *b, long da)

$a = a + b \pmod{2^{32d(a)}}$, where $d(a) = d(b) = da > 0$. Operands a and b can be the same.

long xadd (long *a, long *b, long da, long db)

$a = a + b \pmod{2^{32d(a)}}$, where $d(a) = da \geq d(b) = db > 0$; b is zero-extended if $d(a) > d(b)$. Returns the carry flag in bit 8 and the sign of the result in bit 31. Operands a and b can be the same.

long xadds (long *a, long *b, long da, long db)

$a = a + b \pmod{2^{32d(a)}}$, where $d(a) = da \geq d(b) = db > 0$; b is sign-extended if $d(a) > d(b)$. Returns the carry flag in bit 8 and the sign of the result in bit 31. Operands a and b can be the same.

long xaddk (long *a, long k, long da)

$a = a + k \pmod{2^{32d(a)}}$, where $d(a) = da > 0$ and k is a non-negative, 32-bit integer; k is zero-extended if $d(a) > 1$. Returns the carry flag in bit 8 and the sign of the result in bit 31.

void xsubf (long *a, long *b, long da)

$a = a - b \pmod{2^{32d(a)}}$, where $d(a) = d(b) = da > 0$. Operands a and b can be the same.

long xsub (long *a, long *b, long da, long db)

$a = a - b \pmod{2^{32d(a)}}$, where $d(a) = da \geq d(b) = db > 0$; b is zero-extended if $d(a) > d(b)$. Returns the carry flag in bit 8 and the sign of the result in bit 31. Operands a and b can be the same.

long xsubs (long *a, long *b, long da, long db)

$a = a - b \pmod{2^{32d(a)}}$, where $d(a) = da \geq d(b) = db > 0$; b is sign-extended if $d(a) > d(b)$.

Returns the carry flag in bit 8 and the sign of the result in bit 31. Operands *a* and *b* can be the same.

long xsubk (long *a, long k, long da)

$a = a - k \pmod{2^{32d(a)}}$, where $d(a) = da > 0$ and *k* is a non-negative, 32-bit integer; *k* is zero-extended if $d(a) > 1$. Returns the carry flag in bit 8 and the sign of the result in bit 31.

void xneg (long *a, long da)

$a = 2^{32d(a)} - a$, the twos complement of *a*. We assume $d(a) = da > 0$.

long xcmp (long *a, long *b, long da, long db)

a and *b* are compared, the shorter being zero-extended, and the flag register is returned in bits 8 to 15. We assume $d(a) = da > 0$, $d(b) = db > 0$.

long xcmps (long *a, long *b, long da, long db)

a and *b* are compared, the shorter being sign-extended, and the flag register is returned in bits 8 to 15. We assume $d(a) = da > 0$, $d(b) = db > 0$.

long xsig (long *a, long da)

Returns the high-order digit of *a*. We assume $d(a) = da > 0$.

long xdig (long *a, long da)

Returns the number of significant digits of *a*. We assume $d(a) = da > 0$.

void xcar (long *a)

Add 1 to *a* and carry indefinitely.

void xbor (long *a)

Subtract 1 from *a* and borrow indefinitely.

void xsqu (long *a, long da)

$a = a^2$ (using the 'school' method). We assume $c(a) \geq 2d(a) > 0$, $d(a) = da$.

void xmul (long *a, long da, long *b, long db)

$a = a * b$ (using the 'school' method). We assume $c(a) \geq d(a) + d(b)$, $d(a) > 0$, $d(b) > 0$,

$d(a) = da$, $d(b) = db$.

void xmulk (long *a, long k, long da)

$a = a * k$, where k is a 32-bit integer. We assume $c(a) \geq d(a) + 1$, $d(a) = da > 0$.

void xmul2k (long *a, long k, long da)

$a = a * 2^k$, where k is a non-negative, 32-bit integer. We assume $c(a) \geq d(a) + [(k + 31)/32]$, $d(a) = da > 0$.

void xmul2d (long *a, long d, long da)

$a = a * 2^{32d}$, where d is a non-negative, 32-bit integer. We assume $c(a) \geq d(a) + d$, $d(a) = da > 0$.

void xdiv (long *a, long *b, long da, long db, long *q)

$a = (a \bmod b)$, $q = [a / b]$. We assume $c(a) > d(a) \geq d(b) > 0$, $c(q) \geq d(a) – d(b) + 1$, $b > 0$, $d(a) = da$, $d(b) = db$.

long xdivk (long *a, long k, long da)

$a = [a / k]$, where k is a positive, 32-bit integer. Returns $(a \bmod k)$. We assume $d(a) = da > 0$.

void xdiv2k (long *a, long k, long da)

$a = [a / 2^k]$, where k is a non-negative, 32-bit integer. We assume $0 \leq k < 32d(a)$, $d(a) = da > 0$.

void xdiv2d (long *a, long d, long da)

$a = [a / 2^{32d}]$, where k is a non-negative, 32-bit integer. We assume $0 \leq d < d(a)$, $d(a) = da > 0$.

void xmod (long *a, long *b, long da, long db)

$a = (a \bmod b)$. We assume $c(a) > d(a) \geq d(b) > 0$, $b > 0$, $d(a) = da$, $d(b) = db$.

void xmodk (long *a, long k, long da)

$a = (a \bmod k)$, where k is a positive, 32-bit integer. We assume $d(a) = da > 0$.

void xsep2k (long *a, long *q, long k, long da)

Separate a into two parts at bit k. Thus, $a = (a \bmod 2k)$ and $b = [a/2^k]$. We assume $0 \le k < 32d(a)$, $d(a) = da > 0$, $c(q) \ge d(a) - [k/32]$.

long xnxtprmk (long *k*)

Return the next prime after k if $k < 2^{32}$. Return 1 if $k \ge 2^{32}$. For obtaining many (not too large) primes consecutively, 'inxtprm' is faster on average.

void xsqumodF12 (long *a, long *b)

$a = b^2 \pmod{2^{4096} + 1}$ using the Schönhage-Strassen method based on the 128-dimensional fast Fourier transform modulo F_7. We assume $c(a) \ge 129$, $c(b) \ge 129$, $d(a) = d(b) = 129$, $0 \le b < 2^{4096}+1$.

void xsqufft77 (long *a, long da, long *f)

$a = a^2$ with the 128-dimensional fast Fourier transform modulo F_7, using f as a work area. We assume $c(a) \ge 242$, $0 < d(a) \le 120$, $c(f) \ge 645$.

void xsqufft88 (long *a, long da, long *f)

$a = a^2$ with the 256-dimensional fast Fourier transform modulo F_8, using f as a work area. We assume $c(a) \ge 962$, $0 < d(a) \le 480$, $c(f) \ge 2313$.

void xsqufft89 (long *a, long da, long *f)

$a = a^2$ with the 512-dimensional fast Fourier transform modulo F_8, using f as a work area. We assume $c(a) \ge 1922$, $0 < d(a) \le 960$, $c(f) \ge 4617$.

void xsqufft8A (long *a, long da, long *f)

$a = a^2$ with the 1024-dimensional fast Fourier transform modulo F_8, using f as a work area. We assume $c(a) \ge 3842$, $0 < d(a) \le 1920$, $c(f) \ge 9225$.

void xsqufft80 (long *a, long da, long *f)

$a = a^2$. If a is small, or $d(a) > 1920$, this is equivalent to xsqu(a, da). Otherwise, we compute a^2 with the fast Fourier transform modulo F_8 (using f as a work area) and choosing the best dimension from the set {32, 64, 128, 256, 512, 1024}. We assume $c(f) \ge 9225$, $0 < d(a) \le 1920$ and $c(a)$ sufficiently large, for example, $c(a) = \min\{4d(a), 3842\}$.

void xsqufftA0 (long *a, long da, long *f)

$a = a^2$. If a is small, or $d(a) > 32256$, this is equivalent to xsqu(a, da). Otherwise, we compute a^2 with the fast Fourier transform modulo F_{10} (using f as a work area) and

choosing the best dimension from the set {64, 128, 256, 512, 1024, 2048, 4096}. We assume $c(f) \geq 135201$, $0 < d(a) \leq 32256$ and $c(a)$ sufficiently large, for example, $c(a) = \min\{4d(a), 64514\}$.

void xsqufftC0 (long *a, long da, long *f)

$a = a^2$. If a is small, or $d(a) > 129024$, this is equivalent to xsqu(a, da). Otherwise, we compute a^2 with the fast Fourier transform modulo F_{12} (using f as a work area) and choosing the best dimension from the set {32, 64, 128, 256, 512, 1024, 2048, 4096}. We assume $c(f) \geq 528513$, $0 < d(a) \leq 129024$ and $c(a)$ sufficiently large, for example, $c(a) = \min\{4d(a), 258048\}$.

void xdump (long *a, long da)

Unformatted print of an integer array.