MIRACL User's Manual

Shamus Software Ltd.

4 Foster Place North
Ballybough
Dublin 3
Ireland

http://www.shamus.ie

February 2007

Contents

Abstract

The MIRACL library consists of well over 100 routines that cover all aspects of multi-precision arithmetic. Two new data-types are defined — `big` for large integers and `flash` (short for floating-slash) for large rational numbers. The large integer routines are based on Knuth's algorithms, described in Chapter 4 of his classic work 'The Art of Computer Programming'. Floating-slash arithmetic, which works with rounded fractions, was originally proposed by D. Matula and P. Kornerup. All routines have been thoroughfully optimised for speed and efficiency, while at the same time remaining standard, portable C. However optional fast assembly language alternatives for certain time-critical routines are also included, particularly for the popular Intel 80x86 range of processors. A C++ interface is also provided. Full source code is included.

Chapter 1

Introduction

Remember when as a naive young computer user, you received delivery of your brand new state-of-the-art micro; remember your anticipation at the prospect of the computer power now available at your fingertips; remember recalling all those articles which promised that 'todays microcomputers are as powerful as yesterday's mainframes'. Remember then slowly and laboriously typing in your first program, to calculate, say, 1000! (i.e. $1000 \times 999 \times 998 \times \ldots \times 1$) — a calculation unimaginable by hand.

```
10 LET X=1
20 FOR I=1 TO 1000
30 X=X*I
40 NEXT I
50 PRINT X
60 END

RUN
```

After a few seconds the result appeared:

```
        Too big at line 30
```

Remember your disappointment.

Now try the MIRACL approach. MIRACL is a portable C library which implements multiprecision integer and rational data-types, and provides the routines to perform basic arithmetic on them.

Run the program fact from the distribution media, and type in 1000. There is your answer — a 2568 digit number.

Now compile and run the program roots, and ask it to calculate the square root of 2. Virtually instantly your computer comes back with the value correct to 100+ decimal places. Now thats what I call computing!

Next run the Public Key Cryptography program enciph. When it asks the name of a file to be enciphered press return. When it asks for an output filename, type

FRED followed by return. Now type in any message, finishing with CONTROL-Z. Your message has been thoroughly enciphered in the file FRED.BLG (type it out and see). Now run deciph, and type in FRED. Press return for the requested output filename. Your original message appears on the screen.

This type of encipherment, based as it is on the difficulty of factoring large numbers, offers much greater security and flexibility than more traditional methods.

A useful demonstration of the power of MIRACL is given by the program ratcalc, a powerful scientific calculator — accurate to 36 decimal places and with the unusual ability to handle fractions directly.

It is assumed in this manual that the reader is familiar with the C language, and with his/her own computer. On a first reading Chapters 4, 5 and 6 may be safely skipped. Examination of the example programs' source code will be very rewarding.

Chapter 2

Installation

The MIRACL library has been successfully installed on a VAX11/780, on a variety of UNIX workstations (Sun, SPARC, Next, IBM RS/6000), on an IBM PC using the Microsoft C and C++ compilers, Borlands Turbo C and Borland C++ compilers, the Watcom C compiler and the DJGPP GNU compiler; on ARM based computers, and on an Apple Macintosh. Recently it has been implemented on Itanium and AMD 64-bit processors.

The complete source code for each module in the MIRACL library, and for each of the example programs is provided on the distribution media. Most are written in Standard ANSI C, and should compile using any decent ANSI C compiler. Some modules contain extensive amounts of in-line assembly language, used to optimise performance for certain compiler/processor combinations. However these are invoked transparently by conditional compilation commands and will not interfere with other compilers. The batch files xxdoit.xxx contain the commands used for the creation of a library file and the example programs for several compilers. Print out and examine the appropriate file for your configuration.

Pre-compiled libraries for immediate use with certain popular compilers may be found on the distribution media: ready-to-run versions of only some of the example programs may be included, to conserve space.

To create a library you will need access to a compiler, a text editor, a linker, a librarian utility, and an assembler (optional). Read your compiler documentation for further details. The file mrmuldv.any, which contains special assembly language versions of the time-critical routines `muldiv`, `muldvd`, `muldvd2` and `muldvm` together with some portable C versions, which may need to be tailored for your configuration. These modules are particularly required if the compiler does not support a double length type which can hold the product of two word-length integers. Most modern compilers do provide this support (often the double length type is called `long long`), and in this case it is often adequate to use the standard C version of this module mrmuldv.ccc which can simply be copied to mrmuldv.c. Read this manual carefully, and the comments in mrmuldv.any for more details.

The hardware/compiler specific file mirdef.h needs to be specified. To assist with this, five example versions of the header are supplied: mirdef.h16 for use with a 16-bit processor, mirdef.h32 for 32-bit processors, mirdef.haf if using a 32-bit processor in a 16-bit mode, and mirdef.hpc for pseudo 32-bit working in a 16-bit environment. Note that the full 32-bit version is fastest, but only possible if using a true 32-bit compiler with a 32-bit processor. Try mirdef.gcc for use with gcc and g++ in a Unix environment (no assembler).

To assist with the configuration process, a file config.c is provided. When compiled and run *on the target processor* it automatically generates a mirdef.h file and gives general advice on configuration. It also generates a miracl.lst file with a list of MIRACL modules to be included in the associated library build. Experimentation with this program is strongly encouraged. When compiling this program DO NOT use any compiler optimization.

The mirdef.h file contains some optional definitions: Define `MR_NOFULLWIDTH` if you are unable to supply versions of `muldvd`, `muldvd2` and `muldvm` in mrmuldv.c. Define `MR_FLASH` if you wish to use flash variables in your programs. Either one of `MR_LITTLE_ENDIAN` or `MR_BIG_ENDIAN` must be defined. The config.c program automatically determines which is appropriate for your processor.

By omitting the `MR_FLASH` definition big variables can be made much larger, and the library produced will be much smaller, leading to more compact executables. Define `MR_STRIPPED_DOWN` to omit error messages, to save even more space in production code. Use with care!

If you dont want any assembler, define `MR_NOASM`. This generates standard C code for the four time-critical routines, and generates it in-line. This is faster — saves on function calling overhead — and also gives an optimising compiler something to chew on. Note that if `MR_NOASM` is defined, then the mrmuldv module is not required in the MIRACL library.

If using the Microsoft Visual C++ tool, some helpful advice can be found in the file msvisual.txt. If using the Linux operating system, check out linux.txt. Users of the Borland compiler should look at borland.txt.

In the majority of cases where pre-built libraries or specific advice in a .txt file is not available, the following procedure will result in a successful build of the MIRACL library:

1. Compile and run config.c on the target processor.
2. Rename the generated file mirdef.tst to mirdef.h
3. If so advised by the config program, extract a suitable mrmuldv.c file from mrmuldv.any (or copy the standard C version mrmuldv.ccc to mrmuldv.c and use this). If it is pure assembly language it may be appropriate to name it mrmuldv.s or mrmuldv.asm.
4. If the fast KCM or Comba methods for modular multiplication were selected (see below), compile and run the mex.c utility on any workstation. Use it to automatically generate either the module mrcomba.c or mrkcm.c. This will require a processor/compiler-specific xxx.mcs file. The compiler must support inline assembly.
5. Make sure that all the MIRACL header files are accessible to the compiler. Typically the flag `-I.` or `/I.` allows these headers to be accessed from the current directory.
6. Compile the MIRACL modules listed in the generated file miracl.lst and create a library file, typically miracl.a or miracl.lib. This might be achieved by editing miracl.lst into a suitable batch or make file. On UNIX it might be as simple as:

   ```
   gcc -I. -c -O2 mr*.c
   ar rc miracl.a mr*.o
   ```
7. If using the C++ MIRACL wrapper, compile the required modules, for example zzn.cpp and/or big.cpp etc.

8. Compile and link your application code to any C++ modules it requires and to the MIRACL library.

Remember that MIRACL is portable software. It may be ported to any computer which supports an ANSI C compiler.

Note that MIRACL is a C library, not C++. It should always be built as a C library otherwise you might get compiler errors. To include MIRACL routines in a C program, include the header miracl.h at the start of the program, after including the C standard header stdio.h. You may also call MIRACL routines directly from a C++ program by:

```
extern "C"
{
    #include "miracl.h"
}
```

although in most cases it will be preferable to use the C++ wrapper classes described in Chapter 7.

2.1 Optimising

In the context of MIRACL this means speeding things up. A critical decision to be made when configuring MIRACL is to determine the optimal underlying type to use. Usually this will be the `int` type. In general try to define the maximum possible underlying type, as requested by config. If you have a 64-bit processor, you should be able to specify a 64-bit underlying type. In some circumstances it may be faster to use a floating-point double underlying type.

Obviously an all-C build of MIRACL will be slowest (but still pretty fast!). It is also the easiest to start with. This requires an integer data type twice the width of the underlying type. In this context note that these days most compilers support a long long integer type which is twice the width of the `int`. Sometimes it is called `__int64` instead of `long long`.

If your processor is of the extreme RISC variety and supports no integer multiplication/division instruction, or if using a very large modulus, then the Karatsuba-Montgomery-Comba technique for fast modular multiplication may well be faster for exponentiation cryptosystems. Again the config program will guide you through this.

It is sometimes faster to implement the mrmuldv module in assembly language. This does not require the double-width data type. If you are lucky your compiler will also be supported by automatically invoked inline assembly, which will speed things up even further. See miracl.h to see which compilers are supported in this way.

For the ultimate speed, use the extreme techniques implemented in mrkcm.c, mrcomba.c. See kcmcomba.txt for instructions on how to automatically generate these files using the supplied mex utility. See also Chapter 5 for more details.

2.2 Upgrading from Version 3

Version 4.0 introduces the MIRACL Instance Pointer, or *mip*. Previous versions used a number of global and static variables to store internal status information. There are two problems with this. Firstly such globals have to be given obscure names to avoid clashes with other project globals. Secondly it makes multi-threaded applications much more difficult to develop. So from Version 4.0 all such variables, now referred to as instance variables, are members of a structure of type `miracl`, and must be accessed via a pointer to an instance of this structure. This global pointer is now the only static/global variable maintained by the MIRACL library. Its value is returned by the `mirsys` routine, which initialises the MIRACL library.

C++ programmers should note the change in the name of the instance class from `miracl` to `Miracl`. The *mip* can be found by taking the address of this instance.

```
Miracl precision = 50;
...
mip = &precision;
...
etc
```

2.3 Multi-Threaded Programming

From version 4.4 MIRACL offers full support for multi-threaded programming. This comes in various flavours.

The problem to be overcome is that MIRACL has to have access to a lot of instance specific status information via its *mip*. Ideally there should be no global variables, but MIRACL has this one pointer. Unfortunately every thread that uses MIRACL needs to have its own *mip*, pointing to its own independent status. This is a well-known issue that arises with threads.

The first solution is to modify MIRACL so that the *mip*, instead of being a global, is passed as a parameter to every MIRACL function. The MIRACL routines can be automatically modified to support this by defining `MR_GENERIC_MT` in mirdef.h. Now (almost all) MIRACL routines are changed such that the *mip* is the first parameter to each function. Some simple functions are exceptions and do not require the *mip* parameter — these are marked with an asterix in the reference manual[1]. For an example of a program modified to work with a MIRACL library built in this way, see the program brent_mt.c. Note however that this solution does NOT apply to programs written using the MIRACL C++ wrapper described in Chapter 7. It only applies to C programs that access the MIRACL routines directly.

An alternative solution is to use Keys, which are a type of thread specific "global" variable. These Keys are not part of the C/C++ standard, but are operating system specific extensions, implemented via special function calls. MIRACL provides support for both Microsoft Windows and Unix operating systems. In the former case these Keys are called Thread-Local Storage. See [Richter] for more information. For Unix MIRACL supports the POSIX standard interface for multithreading. A very useful reference for both Windows and Unix is [Walmsley]. This support for threads is implemented in the module mrcore.c, at the start of the file and in the initialisation routine `mirsys`.

For Windows, define `MR_WINDOWS_MT` in mirdef.h, and for Unix define `MR_UNIX_MT`. In either case there are some programming implications.

[1]Review this.

In the first place the Key that is to maintain the *mip* must be initialised and ultimately destroyed by the programs primary thread. These functions are carried out by calls to the special routines `mr_init_threading` and `mr_end_threading` respectively.

In C++ programs these functions might be associated with the constructor and destructor of a global variable [Walmsley] — this will ensure that they are called at the appropriate time before new threads are forked off from the main thread. They must be called before any thread calls `mirsys` either explicitly, or implicitly by creating a thread-specific instance of the class `Miracl`.

It is strongly recommended that program development be carried out without support for threads. Only when a program is fully tested and debugged should it be converted into a thread.

Threaded programming may require other OS-specific measures, in terms of linking to special libraries, or access to special heap routines. In this regard it is worth pointing out that all MIRACL heap accesses are via the module mralloc.c.

See the example program threadwn.cpp for an example of Windows C++ multithreading. Read the comments in this program — it can be compiled and run from a Windows Command prompt. Similarly see threadux.cpp for an example of Unix multithreading.

2.4 Constrained environments

In version 5 of MIRACL there is new support for implementations in very small and constrained environments. Using the config utility it is now possible to allow various time/space trade-offs, but the main innovation is the possibility of building and using MIRACL in an environment which does not support a heap. Normally space for big variables is obtained from the heap, but by specifying in the configuration header `MR_STATIC`, a version of the library is built which will always attempt to allocate space not from the heap, but from static memory or from the stack.

The main downside to this is that the maximum size of big variables must be set at compile time, when the library is being created. As always it is best to let the config utility guide you through the process of creating a suitable mirdef.h configuration header.

For the C programmer, the allocation of memory from the stack for big variables proceeds as follows.

```
    big x, y, z;
    char mem[MR_BIG_RESERVE(3)];
    memset(mem, 0, MR_BIG_RESERVE(3));
```

This allocates space for 3 big variables on the stack, and set that memory to zero. Each individual big variable is then initialised as

```
    x = mirvar_mem(mem, 0);
    y = mirvar_mem(mem, 1);
    z = mirvar_mem(mem, 2);
```

Allocating all the space for multiple big variables from a single chunk of memory makes sense, as it leads to a faster initialization, and also gives complete control over variable alignment, which compilers sometimes get wrong. Note that in this mode the usual big number initialization function mirvar is no longer available, and allocation must be implemented as described above.

Finally this memory chunk may optionally be cleared before leaving a function by a final call to memset() — this may be important for security reasons. For an example see the program brent.c.

This mechanism may be particularly useful when trying to implement a very small program using elliptic curves, which anyway require much smaller big numbers than other cryptographic techniques. To allocate memory from the stack for an elliptic curve point

```
    epoint *x, *y, *z;
    char mem[MR_ECP_RESERVE(3)];
    memset(mem, 0, MR_ECP_RESERVE(3));
```

To initialize these points

```
    x = epoint_init_mem(mem, 0);
    y = epoint_init_mem(mem, 1);
    z = epoint_init_mem(mem, 2);
```

Again it may be advisable to clear the memory associated with these points before exiting the function.

This mechanism is fully supported for C++ programs as well, where it works in conjunction with the stack allocation method described in chapter 7. See pk-demo.cpp for an example of use.

In some extreme cases it may be desired to use only the stack for all memory allocation. This allows maximum use and re-use of memory, and avoids any fragmentation of precious RAM. This can be achieved for C programs by defining MR_GENERIC_MT in mirdef.h. See above for more details on this option.

A typical mirdef.h header in this case might look like:

```
/*
 *   MIRACL compiler/hardware definitions - mirdef.h
 *   Copyright (c) 1988-2005 Shamus Software Ltd.
 */

#define MR_LITTLE_ENDIAN
#define MIRACL 32
#define mr_utype int
#define MR_IBITS 32
#define MR_LBITS 32
#define mr_unsign32 unsigned int
#define mr_dltype __int64
#define mr_unsign64 unsigned __int64
#define MR_STATIC 7
#define MR_ALWAYS_BINARY
#define MR_NOASM
#define MAXBASE ((mr_small)1<<(MIRACL-1))
```

```
#define MR_BITSINCHAR 8
#define MR_SHORT_OF_MEMORY
#define MR_GENERIC_MT
#define MR_STRIPPED_DOWN
```

For examples of programs which use this kind of header, see ecsgen_s.c, ecsign_s.c and ecsver_s.c, and ecsgen2s.c, ecsign2s.c and ecsver2s. These programs implement very small and fast ECDSA key generation, digital signature, and verification on a Pentium using Microsoft C++. See ecdh.c for another nice example, which uses precomputation to speed up an EC Diffie-Hellman implementation.

NOTE: Doing without a heap is a little problematical. Structures can no longer be of variable size, and so various features of MIRACL become unavailable in this mode. For example precomputations such as required for application of the Chinese remainder theorem are no longer supported. However in a constrained environment it could be reasonably assumed that such precomputations are carried out off-line, and made available to the constrained program fixed in ROM.

The MIRACL modules are carefully designed so that an application will only pull in the minimal number of modules from the library for any given task. This helps to keep the program size down to a minimum. However if program size is a big issue then extra savings can sometimes be made by manually deleting from the modules functions that are not needed by your particular program (the linker will complain if the function is in fact needed).

2.5 Platform-specific information

AMD64

The AMD64 processor is now fully supported using Intel GCC Compiler.

Use a header file like

```
#define MR_LITTLE_ENDIAN
#define MIRACL 64
#define mr_utype long
#define MR_IBITS 32
#define MR_LBITS 64
#define mr_unsign32 unsigned int
#define mr_unsign64 unsigned long
#define MR_FLASH 52
#define MAXBASE ((mr_small)1<<(MIRACL-1))
#define BITSINCHAR 8
```

and use assembly language file mrmuldv.s64.

Note that the above header file assumes an LP64-compatible compiler. For an LLP64 compiler, change `mr_utype` to a 64-bit `long long` or `__int64`.

There is also a macro file amd64.mcs — see kcmcomba.txt and makemcs.txt. However when we tried it the `-O2` optimizer was broken when compiling mrcomba.c or mrkcm.c.

To build the MIRACL library, extract below into a file amd64 and execute

```
bash amd64
```

Contents of `amd64` should be

```
rm miracl.a
gcc -I. -c -O2 mrcore.c
gcc -I. -c -O2 mrarth0.c
gcc -I. -c -O2 mrarth1.c
gcc -I. -c -O2 mrarth2.c
gcc -I. -c -O2 mralloc.c
gcc -I. -c -O2 mrsmall.c
gcc -I. -c -O2 mrio1.c
gcc -I. -c -O2 mrio2.c
gcc -I. -c -O2 mrgcd.c
gcc -I. -c -O2 mrjack.c
gcc -I. -c -O2 mrxgcd.c
gcc -I. -c -O2 mrarth3.c
gcc -I. -c -O2 mrrand.c
gcc -I. -c -O2 mrprime.c
gcc -I. -c -O2 mrcrt.c
gcc -I. -c -O2 mrscrt.c
gcc -I. -c -O2 mrmonty.c
gcc -I. -c -O2 mrpower.c
gcc -I. -c -O2 mrcurve.c
gcc -I. -c -O2 mrfast.c
gcc -I. -c -O2 mrshs.c
gcc -I. -c -O2 mrshs256.c
gcc -I. -c -O2 mrshs512.c
gcc -I. -c -O2 mraes.c
gcc -I. -c -O2 mrlucas.c
gcc -I. -c -O2 mrstrong.c
gcc -I. -c -O2 mrbrick.c
gcc -I. -c -O2 mrebrick.c
gcc -I. -c -O2 mrecgf2m.c
gcc -I. -c -O2 mrflash.c
gcc -I. -c -O2 mrfrnd.c
gcc -I. -c -O2 mrdouble.c
gcc -I. -c -O2 mrround.c
gcc -I. -c -O2 mrbuild.c
gcc -I. -c -O2 mrflsh1.c
gcc -I. -c -O2 mrpi.c
gcc -I. -c -O2 mrflsh2.c
gcc -I. -c -O2 mrflsh3.c
gcc -I. -c -O2 mrflsh4.c
as mrmuldv.s64 -o mrmuldv.o
ar rc miracl.a mrcore.o mrarth0.o mrarth1.o mrarth2.o mralloc.o mrsmall.o
ar r miracl.a mrio1.o mrio2.o mrjack.o mrgcd.o mrxgcd.o mrarth3.o
ar r miracl.a mrrand.o mrprime.o mrcrt.o mrscrt.o mrmonty.o mrcurve.o
ar r miracl.a mrpower.o mrfast.o mrshs.o mrshs256.o mraes.o mrlucas.o mrstrong.o
ar r miracl.a mrflash.o mrfrnd.o mrdouble.o mrround.o mrbuild.o
ar r miracl.a mrflsh1.o mrpi.o mrflsh2.o mrflsh3.o mrflsh4.o
ar r miracl.a mrbrick.o mrebrick.o mrecgf2m.o mrshs512.o mrmuldv.o
gcc -I. -O2 factor.c miracl.a -lm -o factor
rm mr*.o
```

ARM

If developing for the ARM, or indeed any other new processor, you should first build a C-only library.

For the ARM, this mirdef.h header would be appropriate for an integer-only build of the library.

```
/*
 *   MIRACL compiler/hardware definitions - mirdef.h
 *   Copyright (c) 1988-2001 Shamus Software Ltd.
 */

#define MIRACL 32
#define MR_LITTLE_ENDIAN

/* or possibly
#define MR_BIG_ENDIAN
*/

#define mr_utype int
#define MR_IBITS 32
#define MR_LBITS 32
#define mr_dltype long long
#define mr_unsign32 unsigned int
#define mr_unsign64 unsigned long long
#define MAXBASE ((mr_small)1<<(MIRACL-1))

#define MR_NOASM
```

Assuming that the mirdef.h, miracl.h and mr*.c files are all in the same directory, then a suitable batch file for building a MIRACL library might look like this:

```
armcc -I. -c -O2 mrcore.c
armcc -I. -c -O2 mrarth0.c
armcc -I. -c -O2 mrarth1.c
armcc -I. -c -O2 mrarth2.c
armcc -I. -c -O2 mralloc.c
armcc -I. -c -O2 mrsmall.c
armcc -I. -c -O2 mrio1.c
armcc -I. -c -O2 mrio2.c
armcc -I. -c -O2 mrgcd.c
armcc -I. -c -O2 mrjack.c
armcc -I. -c -O2 mrxgcd.c
armcc -I. -c -O2 mrarth3.c
armcc -I. -c -O2 mrrand.c
armcc -I. -c -O2 mrprime.c
armcc -I. -c -O2 mrcrt.c
armcc -I. -c -O2 mrscrt.c
armcc -I. -c -O2 mrmonty.c
armcc -I. -c -O2 mrpower.c
armcc -I. -c -O2 mrsroot.c
armcc -I. -c -O2 mrcurve.c
armcc -I. -c -O2 mrfast.c
armcc -I. -c -O2 mrshs.c
armcc -I. -c -O2 mrshs256.c
armcc -I. -c -O2 mrshs512.c
armcc -I. -c -O2 mraes.c
armcc -I. -c -O2 mrlucas.c
armcc -I. -c -O2 mrstrong.c
armcc -I. -c -O2 mrbrick.c
armcc -I. -c -O2 mrebrick.c
armcc -I. -c -O2 mrecgf2m.c
armar rc miracl.a mrcore.o mrarth0.o mrarth1.o mrarth2.o mralloc.o
armar r  miracl.a mrsmall.o mrio1.o mrio2.o mrjack.o mrgcd.o mrxgcd.o
armar r  miracl.a mrarth3.o mrrand.o mrprime.o mrcrt.o mrscrt.o
armar r  miracl.a mrmonty.o mrcurve.o mrfast.o mrshs.o mraes.o mrlucas.o
armar r  miracl.a mrstrong.o mrbrick.o mrebrick.o mrecgf2m.o mrpower.o
armar r  miracl.a mrsroot.o mrshs256.o mrshs512.o
del mr*.o
armcc -I. -c pk-demo.c
armlink pk-demo.o miracl.a -o pk-demo.axf
```

This may be fast enough for you. If its not you can use the assembly language macros provided in arm.mcs or gccarm.mcs for greater speed. See kcmcomba.txt.

For faster RSA and DH implementations replace the `MR_NOASM` definition with `MR_KCM n` (where `n` is usually 4, 8 or 16 — experiment. `n*MIRACL` must divide the modulus size in bits exactly, which it will for standard moduli of 1024 bit for example). Compile and run the utility mex.c

```
    mex n arm mrkcm
```

(Yes, it's the same `n`). Rebuild the MIRACL library, but this time include the modules mrkcm.c and mrmuldv.c (you can find the latter in mrmuldv.ccc. This standard C version will do.)

For fast GF(p) elliptic curves, replace `MR_NOASM` with `MR_COMBA n`. This time `32*n` is exactly the size of p in bits (assuming 32-bit processor).

This approach is also optimal for 1024-bit RSA decryption using the Chinese Remainder Theorem. Set `n=16` ($512 = 16 \times 32$)

```
    mex n arm mrcomba
```

Rebuild the MIRACL library, but this time include the modules mrcomba.c and mrmuldv.c.

Still not fast enough? If the prime p is of a "special" form for an elliptic curve, define in mirdef.h `MR_SPECIAL`. Edit mrcomba.tpl to insert "special" code for modular reduction — it's quite easy and you will find examples there already. Run mex as before, and rebuild MIRACL again.

For processors other than the ARM, the basic procedure is the same. A C-only build is always possible. To go faster you will need to create a .mcs file for your processor, and then you can proceed as above.

An alternative is to do a C-only build and then go in and optimise the generated assembly language. The time-critical routines are usually `multiply()` and `redc()` which can be found in mrarth2.c and mrmonty.c.

This will probably not be as fast as the highly optimised approach outlined above.

NOTE: There is a nasty ARM compiler bug in the version I am using. It can cause problems, if for example using the C-only macros from c.mcs or c1.mcs.

Use this program to illustrate the bug, or to see if your compiler is affected.

```
/* Short program to illustrate ARM compiler bug
   works fine with -O0, gets wrong answer for -O1 and -O2 optimization
   Answer should be 0xffffffff00000001 but it gets 0x1
*/

#include <stdio.h>

int main()
{
    unsigned long long x;
    unsigned long a,b;
    a=0;
    b=0xFFFFFFFF;
    x=(unsigned long long)a-b;
    printf("x= %llx\n",x);
    return 0;
}
```

Another problem may arise with systems that do not fully support `unsigned long long` arithmetic (you may be getting linker errors with names like `__udivdi3` functions not found). In this case for a C only build delete the `#define MR_NOASM` from mirdef.h and use the Blakely-Sloan versions of `mrmuldiv` and `mrmuldvm` with the standard versions of `mrmuldvd` and `mrmuldvd2` (from mrmuldv.ccc) to create a file mrmuldv.c which should then be included in the library. Also insert an `#undef mr_dltype` at the start of mrxgcd.c.

Borland

You have just downloaded the "free" and excellent Borland Compiler from `http://www.borland.com`, and you want to compile the MIRACL library and create some applications. If so, read on…

If you have the TASM assembler (which is not free) then unzip all the MIRACL files into one directory, read the comments at the start of bc32doit.bat and if happy execute the batch file. Some example commands to build some representative applications are at the end of the batch file.

If you don't have TASM then you can still build a C-only library (which will be slower). Proceed as follows.

1. Unzip MIRACL into a single directory — do not tick the "Use Folder Names" box if using WinZip
2. Use this header for mirdef.h. Note that Borland now supports a 64-bit data type called `__int64` (compatible with Microsoft C)

```
#define MIRACL 32
#define MR_LITTLE_ENDIAN
#define mr_utype int
#define MR_IBITS 32
#define MR_LBITS 32
#define mr_unsign32 unsigned int
#define mr_dltype __int64
#define mr_unsign64 unsigned __int64
#define MR_NOASM
#define MR_FLASH 52
#define MAXBASE ((mr_small)1<<(MIRACL-1))
```

3. Copy all the MIRACL header files into the directory where Borland C puts its standard headers. This may be c:
   borland
   bcc55
   include
4. Edit bc32doit.bat. Read the comments at the start. Remove all `-B` compiler flags (these invoke TASM, and you haven't got TASM). Delete all references to mrmuldv.c
5. Run the batch file.

Itanium

The Itanium processor is now fully supported using Intel C/C++ Compiler.

Use a header file like

```
#define MR_LITTLE_ENDIAN
#define MIRACL 64
#define mr_utype long
#define MR_IBITS 32
#define MR_LBITS 64
#define mr_unsign32 unsigned int
#define mr_unsign64 unsigned long
#define MR_FLASH 52
#define MAXBASE ((mr_small)1<<(MIRACL-1))
#define BITSINCHAR 8
```

and create file mrmuldv.c from Itanium source code in mrmuldv.any.

Note that this mrmuldv.c file only implements `muldiv(.)` and `muldvm(.)`. The other two functions — the time critical ones — `muldvd(.)` and `muldvd2(.)` are inlined — see miracl.h.

Note that the above header file assumes an LP64-compatible compiler. For an LLP64 compiler, change `mr_utype` to a 64-bit `long long` or `__int64`.

There is also a macro file itanium.mcs — see kcmcomba.txt and makemcs.txt.

To build the MIRACL library, extract below into a file itanium and execute

```
bash itanium
```

Contents of itanium:

```
rm miracl.a
icc -I. -c -O3 mrcore.c
icc -I. -c -O3 mrarth0.c
icc -I. -c -O3 mrarth1.c
icc -I. -c -O3 mrarth2.c
icc -I. -c -O3 mralloc.c
icc -I. -c -O3 mrsmall.c
icc -I. -c -O3 mrio1.c
icc -I. -c -O3 mrio2.c
icc -I. -c -O3 mrgcd.c
icc -I. -c -O3 mrjack.c
icc -I. -c -O3 mrxgcd.c
icc -I. -c -O3 mrarth3.c
icc -I. -c -O3 mrrand.c
icc -I. -c -O3 mrprime.c
icc -I. -c -O3 mrcrt.c
icc -I. -c -O3 mrscrt.c
icc -I. -c -O3 mrmonty.c
icc -I. -c -O3 mrpower.c
icc -I. -c -O3 mrcurve.c
icc -I. -c -O3 mrfast.c
icc -I. -c -O3 mrshs.c
icc -I. -c -O3 mrshs256.c
icc -I. -c -O3 mrshs512.c
icc -I. -c -O3 mraes.c
icc -I. -c -O3 mrlucas.c
icc -I. -c -O3 mrstrong.c
icc -I. -c -O3 mrbrick.c
icc -I. -c -O3 mrebrick.c
icc -I. -c -O3 mrecgf2m.c
icc -I. -c -O3 mrflash.c
icc -I. -c -O3 mrfrnd.c
icc -I. -c -O3 mrdouble.c
icc -I. -c -O3 mrround.c
icc -I. -c -O3 mrbuild.c
icc -I. -c -O3 mrflsh1.c
icc -I. -c -O3 mrpi.c
icc -I. -c -O3 mrflsh2.c
icc -I. -c -O3 mrflsh3.c
icc -I. -c -O3 mrflsh4.c
icc -I. -c -O3 mrmuldv.c
ar rc miracl.a mrcore.o mrarth0.o mrarth1.o mrarth2.o mralloc.o mrsmall.o
ar r miracl.a mrio1.o mrio2.o mrjack.o mrgcd.o mrxgcd.o mrarth3.o
ar r miracl.a mrrand.o mrprime.o mrcrt.o mrscrt.o mrmonty.o mrcurve.o
ar r miracl.a mrpower.o mrfast.o mrshs.o mrshs256.o mraes.o mrlucas.o
ar r mrstrong.o miracl.a mrflash.o mrfrnd.o mrdouble.o mrround.o
ar r mrbuild.o miracl.a mrflsh1.o mrpi.o mrflsh2.o mrflsh3.o mrflsh4.o
ar r miracl.a mrbrick.o mrebrick.o mrecgf2m.o mrshs512.o mrmuldv.o
icc -I. -O3 factor.c miracl.a -lm -o factor
rm mr*.o
```

Microsoft Visual C++

To create MIRACL applications under Microsoft Windows using Microsoft C++ version 6.0, follow these steps. Remember a “Release” build will be faster than a “Debug” build. For MS VC++ .NET see below.

1. Create a new empty project of type Win32 Console Application”. Give it the name of the associated main project file, e.g. limlee for limlee.cpp
2. Click on “Project”, and then on “Add to Project”, and then on “Files”
3. Select “library files (.lib)” from the “Files of type” drop-down list. Look for the precompiled MIRACL library file ms32.lib, and insert it into the project.
4. Add the main project file (e.g. limlee.cpp) into the project.
5. Only if it is a C++ main project file, also add into the project any other needed files, like big.cpp/zzn.cpp etc. Those required are specified in the comments at the start of the main project file, e.g. `* Requires: big.cpp zzn.cpp`. This step is not required for C projects.
6. Now click on “Project” and then on “Settings”. Click the C/C++ tab and select the “preprocessor” category. Find the box “Additional Include Directories”, and specify a path to the MIRACL header files, mirdef.h/miracl.h/big.h etc.
7. Make sure that any files needed by the application are available in the directory from which the application is run. For example *.key *.dss, or *.ecs files
8. Now build and run the project.

To create a new version of the MIRACL library,

1. Compile and run the config.c utility, and rename the generated mirdef.tst to mirdef.h. Note that Microsoft C has a 64-bit integer data type called `__int64`
2. Create a new project of type “Win32 Static Library”. Then click on “Finish”.
3. Add in the appropriate files mr*.c. The ones required are those listed in miracl.lst (also generated by the config utility).
4. Now click on “Project” and then on “Settings”. Click the C/C++ tab and select the “preprocessor” category. Find the box “Additional Include Directories”, and specify a path to the MIRACL header files mirdef.h and miracl.h
5. Now build the project and create the MIRACL library.

To create MIRACL applications under Microsoft Windows using Microsoft C++ .NET, follow these steps.

1. Click on New and Project, and select “Visual C++ Projects”, “Win32”, select “Win32 Console Project”, and give the project name, for example limlee, for the limlee.cpp example. Click on Finish.
2. Now things get a little tricky. An empty limlee.cpp file is provided. You must now cut and paste from the MIRACL provided file limlee.cpp into this one. Note that the main program is now called `_tmain`.
3. Click on Project, and on “Add existing Item”, and Select “All files” from the “Files of type” drop-down list. Look for the precompiled MIRACL library file ms32.lib, and insert it into the project.

4. Only if it is a C++ main project file, also add into the project any other needed files, like big.cpp/zzn.cpp etc. Those required are specified in the comments at the start of the main project file, e.g. `* Requires: big.cpp zzn.cpp`. This step is not required for C projects. Use "Project", and "Add existing Item".
5. Click on Project and then "Properties". Open up the C++ Tab, go to "Precompiled Headers", and select "Not using precompiled headers" from the drop-down box. Open the "General" Tab, and specify a path to the MIRACL header files, mirdef.h, miracl.h and big.h etc.
6. Build the project and run it by clicking "Debug" and "Start". The program runs to completion and exits.

To create a new version of the MIRACL library, first create a suitable mirdef.h as above, then

1. Click on New and Project, and select "Visual C++ Projects", "Win32", enter the name Miracl and click on "Win32 Project" and then OK.
2. Click on "Application Settings", and then select "Static Library". Then click on Finish
3. Add in the appropriate files mr*.c. (Project followed by "Add Existing Items").
4. Click on Project and then "Properties". Open up the C++ Tab, go to "Precompiled Headers", and select "Not using precompiled headers" from the drop-down box. Open the "General" Tab, and specify a path to the MIRACL header files, mirdef.h and miracl.h
5. Now build the project.

If using MIRACL within an MFC based Win32 project, here are some hints.

1. Build a MIRACL library with `MR_NO_STANDARD_IO` defined in mirdef.h. The config.c utility can be used as usual to create a suitable mirdef.h
2. If using the C++ MIRACL classes, don't forget to change the default project settings for the MIRACL implementation files such as big.cpp to "not using pre-compiled headers".
3. Don't forget that MIRACL is a C library, not C++. To call MIRACL routines directly from a C++ program you must:

   ```
   extern "C"
   {
        #include "miracl.h"
   }
   ```

   If using C++, its probably preferable to access MIRACL via the C++ wrapper classes such as implemented by big.h/big.cpp. See big.h file for more tips.
4. To display a big number, convert it first to a string. See the comments at the start of big.h

Microsoft Visual C++ 8.0 With Visual C++ V8.0, the supplied library file ms32.lib will not work, so you will first need to create a new one.

1. Select New Project, Win32 Console Application

```
Name: miracl
Location: d:\\myprojects (for example)
Solution name: miracl
```

2. Click OK
3. Click Application settings
4. Click on Static library.
5. Disable precompiled headers
6. Click on Finish
7. Click on Header Files in the left hand pane
8. Click on Project, and Add Existing Item
9. Add miracl.h and mirdef.h from wherever you have unzipped the MIRACL distribution
10. Click on Source Files in the left hand pane
11. Click on Project, and Add Existing Item
12. Add the following MIRACL source files from the MIRACL distribution to the project: mraes.c, mralloc.c, mrarth0.c, mrarth1.c, mrarth2.c, mrarth3.c, mrbits.c, mrbrick.c, mrbuild.c, mrcore.c, mrcrt.c, mrcurve.c, mrdouble.c, mrebrick.c, mrecgf2m.c, mrfast.c, mrflash.c, mrflsh1.c, mrflsh2.c, mrflsh3.c, mrflsh4.c, mrfrnd.c, mrgcd.c, mrio1.c, mrio2.c, mrjack.c, mrlucas.c, mrmonty.c, mrmuldv.c, mrpi.c, mrpower.c, mrprime.c, mrrand.c, mrround.c, mrscrt.c, mrshs.c, mrshs256.c, mrshs512.c, mrsmall.c, mrsroot.c, mrstrong.c, mrxgcd.c, mrzzn2.c.
13. Then Click on Build miracl. The library is created in directory d:
    myprojects
    miracl
    debug
    miracl.lib
14. Alternatively create a release version in the obvious way (if desired).
15. Close this project
16. Again Select New Project, Win32 Console Application

    ```
    Name: brent
    Location: d:\myprojects
    Solution name: brent
    ```

17. Click on OK, click on Application Settings, leave it as Console Application, and again disable precompiled headers.
18. Click on Finish.
19. Click on Header Files in the left hand pane
20. Click on Project, and Add Existing Item
21. Add miracl.h and mirdef.h from wherever you have unzipped the MIRACL distribution
22. Also add zzn.h and big.h (the files required here are indicated in the comment `/* Requires: big.cpp zzn.cpp */` at the start of brent.cpp)
23. Click on Source Files in the left hand pane
24. Right click on the automatically generated file brent.cpp, and exclude it from the project.
25. Click on Project, and Add Existing Item

26. Add the file brent.cpp from the MIRACL distribution
27. Add the files zzn.cpp and big.cpp from the MIRACL distribution
28. Click on Project, and Add Existing Item. Navigate to wherever the MIRACL library has been created (d:
myprojects
miracl
debug) and add miracl.lib to the project. Answer No to the dialog that appears.
29. Click on Build brent. The source files are compiled and linked to the MIRACL library. To run the program Click on Debug, and then on Start without Debugging.

SmartMIPS®

The SmartMIPS® is an example of the new generation of 32-bit smart cards. Rather than support specialised crypographic co-processors, these smart cards deploy enhanced instruction sets with instructions specially tailored to the requirements of multi-precision arithmetic over $GF(p)$ and $GF(2^m)$. This is a viable approach due to the increased speed and power of these devices. (Smart cards are NOT low powered devices. Power is taken from the card reader, and is thus not particularly limited)

These smart cards also support impressive amounts of ROM, Flash memory, EEPROM and RAM. Nonetheless the environment is heavily constrained compared to a desktop workstation, or even a PDA or hand-held mobile device.

One major constraint is a limited amount of RAM — typically just 16K bytes. In this context it does not make sense to divide this limited resource between static memory, a heap and a stack. Therefore a MIRACL build which requires only a stack is appropriate. Many big number libraries support elaborate mechanisms so that big numbers can grow without limit. In contrast MIRACL has always supported fixed size big numbers, and this is particularly appropriate in this context. By fixing big number sizes at compile time, memory for big numbers can be allocated very quickly from the stack, with minimal overhead. To do this define

```
#define MR_STATIC X
```

in mirdef.h, where `X` is the fixed size of the big numbers in 32-bit words.

A file smartmip.mcs is supplied so that optimal assembly language can be generated for big number modular multiplication, using the MIRACL macro mechanism (see makemcs.txt and kcmcomba.txt). This also supports very fast $GF(2^m)$ polynomial multiplication, using a special instruction. By setting

```
#define MR_COMBA2 X
```

in mirdef.h, and running the mex utility, very fast code will be generated to the file mrcomba2.c, which can be integrated into the MIRACL library. Here `X` is again the fixed size of the big numbers in 32-bit words (rounded up).

An example mirdef.h configuration header for implementing a fast elliptic curve cryptosystem over $GF(2^{283})$ might look like this:

```
/*
```

```
 *   MIRACL compiler/hardware definitions - mirdef.h
 *   Copyright (c) 1988-2006 Shamus Software Ltd.
 */

#define MR_LITTLE_ENDIAN
#define MIRACL 32
#define mr_utype int
#define MR_IBITS 32
#define MR_LBITS 32
#define mr_unsign32 unsigned int
#define mr_dltype long long
#define mr_unsign64 unsigned long long
#define MR_STATIC 9
#define MR_NOASM
#define MR_ALWAYS_BINARY
#define MR_STRIPPED_DOWN
#define MR_GENERIC_MT
#define MR_NO_STANDARD_IO
#define MR_NO_FILE_IO
#define MR_COMBA2 9
#define MAXBASE ((mr_small)1<<(MIRACL-1))
#define MR_BITSINCHAR 8
#define MR_SHORT_OF_MEMORY
```

SPARC

NOTE: On SPARCs with hardware support for quad-precision long doubles, it may be optimal to build a MIRACL library using a `double` underlying type rather than use the approach described here. See double.txt.

These comments apply to the standard 32-bit SPARC (Version 8) processor with hardware 32-bit multiplication. For 64-bit SPARC (Version 9) see below.

If developing for the SPARC, or indeed any other new processor, you should first build a C-only library.

For the SPARC, this mirdef.h header would be appropriate for an integer-only build of the library.

```
/*
 *   MIRACL compiler/hardware definitions - mirdef.h
 *   Copyright (c) 1988-2001 Shamus Software Ltd.
 */

#define MIRACL 32
#define MR_BIG_ENDIAN
#define mr_utype int
#define MR_IBITS 32
#define MR_LBITS 32
#define mr_dltype long long
#define mr_unsign32 unsigned int
#define mr_unsign64 unsigned long long
#define MAXBASE ((mr_small)1<<(MIRACL-1))
```

```
#define MR_NOASM
```

Assuming that the mirdef.h, miracl.h and mr*.c files are all in the same directory, then a suitable batch file for building a MIRACL library might look like this:

```
gcc -I. -c -O2 mrcore.c
gcc -I. -c -O2 mrarth0.c
gcc -I. -c -O2 mrarth1.c
gcc -I. -c -O2 mrarth2.c
gcc -I. -c -O2 mralloc.c
gcc -I. -c -O2 mrsmall.c
gcc -I. -c -O2 mrio1.c
gcc -I. -c -O2 mrio2.c
gcc -I. -c -O2 mrgcd.c
gcc -I. -c -O2 mrjack.c
gcc -I. -c -O2 mrxgcd.c
gcc -I. -c -O2 mrarth3.c
gcc -I. -c -O2 mrrand.c
gcc -I. -c -O2 mrprime.c
gcc -I. -c -O2 mrcrt.c
gcc -I. -c -O2 mrscrt.c
gcc -I. -c -O2 mrmonty.c
gcc -I. -c -O2 mrpower.c
gcc -I. -c -O2 mrsroot.c
gcc -I. -c -O2 mrcurve.c
gcc -I. -c -O2 mrfast.c
gcc -I. -c -O2 mrshs.c
gcc -I. -c -O2 mrshs256.c
gcc -I. -c -O2 mrshs512.c
gcc -I. -c -O2 mraes.c
gcc -I. -c -O2 mrlucas.c
gcc -I. -c -O2 mrstrong.c
gcc -I. -c -O2 mrbrick.c
gcc -I. -c -O2 mrebrick.c
gcc -I. -c -O2 mrecgf2m.c
ar rc miracl.a mrcore.o mrarth0.o mrarth1.o mrarth2.o mralloc.o mrsmall.o
ar r  miracl.a mrio1.o mrio2.o mrjack.o mrgcd.o mrxgcd.o mrarth3.o mrsroot.o
ar r  miracl.a mrrand.o mrprime.o mrcrt.o mrscrt.o mrmonty.o mrcurve.o
ar r  miracl.a mrfast.o mrshs.o mraes.o mrlucas.o mrstrong.o mrbrick.o
ar r  miracl.a mrebrick.o mrecgf2m.o mrpower.o
ar r  miracl.a mrshs256.o mrshs512.o
del mr*.o
gcc -I.-O2 pk-demo.c miracl.a -o pk-demo
```

This may be fast enough for you. If its not you can use the assembly language macros provided in sparc32.mcs for greater speed. See kcmcomba.txt.

For faster RSA and DH implementations replace the `MR_NOASM` definition with `MR_KCM n` (where `n` is usually 4, 8 or 16 — experiment. `n*MIRACL` must divide the modulus size in bits exactly, which it will for standard moduli of 1024 bit for example). Compile and run the utility mex.c

```
c:\miracl>mex n sparc32 mrkcm
```

(Yes it's the same `n`). Rebuild the MIRACL library, but this time include the modules mrkcm.c and mrmuldv.c (you can find the latter as mrmuldv.ccc. This standard C version will do, although the SPARC asm versions from mrmuldv.any are faster. These would need to be assembled rather than compiled)

For fast $GF(p)$ elliptic curves, replace `MR_NOASM` with `MR_COMBA n`. This time 32*`n` is exactly the size of the modulus in bits (assuming 32-bit processor).

```
c:\miracl>mex n sparc32 mrcomba
```

Rebuild the MIRACL library, but this time include the modules mrcomba.c and mrmuldv.c.

Still not fast enough? If the prime modulus is of a "special" form, define in mirdef.h `MR_SPECIAL`. Edit mrcomba.tpl to insert "special" code for modular reduction — it's quite easy and you will find examples there already. Run mex as before, and rebuild MIRACL again.

For processors other than the SPARC, the basic procedure is the same. A C-only build is always possible. To go faster you will need to create a .mcs file for your processor, and then you can proceed as above.

An alternative is to do a C-only build and then go in and optimise the generated assembly language. The time-critical routines are usually `multiply()` and `redc()` which can be found in mrarth2.c and mrmonty.c.

This will probably not be as fast as the highly optimised approach outlined above.

64-bit SPARC (Version 9). Alas not a "real" 64-bit processor in the sense that there is no 64x64=128 bit multiply instruction.

The standard C header files mirdef.h should in this case look like

```
/*
 *   MIRACL compiler/hardware definitions - mirdef.h
 *   Copyright (c) 1988-2001 Shamus Software Ltd.
 */

#define MIRACL 32
#define MR_BIG_ENDIAN
#define mr_utype int
#define MR_IBITS 32
#define MR_LBITS 32
#define mr_dltype long
#define mr_unsign32 unsigned int
#define mr_unsign64 unsigned long
#define MAXBASE ((mr_small)1<<(MIRACL-1))

#define MR_NOASM
```

Compile as above, but include compiler flag `-m64`. Also you may need to change `-O2` to `-O1` — when I tried it `-O2` optimization was broken.

For faster assembly language implementation proceed as above, but this time use macros from sparc64.mcs.

SSE2

If you have a modern Pentium 4 or clone processor that supports the SSE2 extensions, then using these instructions can be faster.

The file sse2.mcs is provided as a plug-in alternative for ms86.mcs, and gccsse2.mcs is provided as an alternative for gcc386.mcs.

Using the COMBA or KCM methods and these provided macros, PCs will execute big number code up to 60% faster. Ideal for a Pentium 4 based Crypto server. See kcmcomba.txt.

It is the programmer's responsibility to ensure that their hardware and their compiler supports SSE2 extensions.

Tested with latest Microsoft (use sse2.mcs) and GCC compilers (V3.3 or greater — use gccsse2.mcs).

The key instruction is `PMULUDQ` which multiplies two pairs of 32-bit numbers in a single instruction. Unfortunately trying to exploit this capability is very difficult. But even just using it for a single multiplication is faster than the standard x386 `MUL` instruction. However SSE2 instructions do not support a carry flag. But the `PADDQ` instruction adds 64-bit numbers.

Consider the following trick:

The 64-bit result of a `PMULUDQ` is written to a 128-bit SSE2 register thus

```
< 32 bits >

+--------+---------+----------+-----------+
|        |         |          |           |
|00000000|000000000|    Hi    |     Lo    |
|        |         |          |           |
+--------+---------+----------+-----------+

<--------------- 128 bits --------------->
```

Now shuffle this (using `PSHUFD`) so it becomes

```
+--------+---------+----------+-----------+
|        |         |          |           |
|00000000|    Hi   |0000000000|     Lo    |
|        |         |          |           |
+--------+---------+----------+-----------+
```

Now accumulate (by simple addition) partial products like these (see makemcs.txt) in an SSE2 register, using the `PADDQ` instruction

```
+--------+---------+----------+-----------+
|        |         |          |           |
|00000CHi|  SumHi  |0000000CLo|   SumLo   |
|        |         |          |           |
+--------+---------+----------+-----------+
```

where `CHi` and `CLo` are accumulated carries from each half.

At the bottom of each column of partial products, the sum for the column is `SumLo`, and the Carry for the next column is the sum of

```
+--------+---------+----------+-----------+
|        |         |          |           |
|   0    |    0    |0000000CHi|   SumHi   |
|        |         |          |           |
+--------+---------+----------+-----------+
```

and

```
+--------+---------+----------+-----------+
|        |         |          |           |
|   0    |    0    |    0     |00000000Clo|
|        |         |          |           |
+--------+---------+----------+-----------+
```

This can easily be achieved using the available shift instructions and `PADDQ`.

Unix

RedHat Linux 6.0+ MIRACL i386 installation. Also works OK for Solaris if its x386/Pentium based.

1. Unzip the MIRACL.ZIP file using the utility unzip, into an empty directory

   ```
   unzip -j -aa -L miracl.zip
   ```

   The `-j` ignores the directory structure inside MIRACL.ZIP. The `-aa` converts all text files to Unix format, and `-L` ensures that all filenames are lower-case.
2. Perform a tailored build of the MIRACL library by opening an X-Term, and typing

   ```
   bash linux
   ```

3. All the MIRACL applications (except RATCALC) can then be built, as desired. Remember to link all C applications to the miracl.a library. C++ applications must be linked as well to one or more of big.o zzn.o ecn.o crt.o flash.o object files etc. See the xxx.bat files for examples. Some applications that require floating-point support may also require `-lm` in the compile command line.

Some programs may require some small changes. For example in schoof.cpp search for the comment about "platforms".

Note that Linux already has (a rather pathetic) factor program. To avoid name clashes you might rename MIRACL's factor program to facter, or somesuch.

Chapter 3

The User Interface

AN EXAMPLE

```
/*
 *   Program to calculate factorials.
 */

#include <stdio.h>
#include "miracl.h"             /* include MIRACL system */

void main()
{
    /* calculate factorial of number */
    big nf;                     /* declare "big" variable nf */
    int n;
    miracl *mip = mirsys(5000, 10);
                                /* base 10, 5000 digits per big */
    nf = mirvar(1);             /* initialise big variable nf=1 */
    printf("factorial program\n");
    printf("input number n= \n");
    scanf("%d", &n);
    getchar();
    while (n > 1)
        premult(nf, n--, nf);   /* nf=n!=n*(n-1)*...2*1  */
    printf("n!= \n");
    otnum(nf, stdout);          /* output result */
}
```

This program can be used to quickly calculate and print out 1000! (a 2568-digit number) in less a second on a 60MHz Intel Pentium-based computer, a task first performed 'by H.S. Uhler using a desk calculator and much patience over a period of several years' [Knuth73]. Many other example programs are described in Chapter 8.

Any program that wishes to make use of the MIRACL system must have an `#include "miracl.h"` statement. This tells the compiler to include the C header file miracl.h with the main program source file before proceeding with the compilation. This file contains declarations of all the MIRACL routines available to the user. The small sub-header file mirdef.h contains hardware/compiler-specific details.

In the main program the MIRACL system must be initialised by a call to the routine `mirsys`, which sets the number base and the maximum size of the big and flash variables. It also initialises the random number system, and creates several workspace big variables for its own internal use. The return value is the Miracl Instance Pointer, or *mip*. This pointer can be used to access various internal parameters associated with the current instance of MIRACL. For example to set the `ERCON` flag, one might write

```
    mip->ERCON = TRUE;
```

The initial call to `mirsys` also initialises the error tracing system which is integrated with the MIRACL package. Whenever an error is detected the sequence of routine calls down to the routine which generated the error is reported, as well as the error itself. A typical error message might be

```
    MIRACL error from routine powltr
        called from isprime
        called from your program
    Raising integer to a negative power
```

Such an error report facilitates debugging, and assisted us during the development of these routines. An associated instance variable `TRACER`, initialised to `OFF`, if set by the user to `ON`, will cause a trace of the program's progress through the MIRACL routines to be output to the computer screen.

An instance flag `ERNUM`, initialised to zero, records the number of the last internal MIRACL error to have occurred. If the flag `ERCON` is set to `FALSE` (the default), an error message is directed to `stdout` and the program aborts via a call to the system routine `exit(0)`. If your system does not supply such a routine, the programmer must provide one instead. If `ERCON` is set to `TRUE` no error message is emitted and instead the onus is on the programmer to detect and handle the error. In this case execution continues. The programmer may choose to deal with the error, and reset `ERNUM` to zero. However errors are usually fatal, and if `ERNUM` is non-zero all MIRACL routines called subsequently will "fall-through" and exit immediately. See miracl.h for a list of all possible errors.

Every big or flash variable in the users program must be initialised by a call to the routine `mirvar`, which also allows the variable to be given an initial small integer value.

The full set of arithmetic and number-theoretic routines declared in miracl.h may be used on these variables. Full flexibility is (almost always) allowed in parameter usage with these routines. For example the call `multiply(x,y,z)`, multiplies the big variable `x` by the big variable `y` to give the result as big variable `z`. Equally valid would be `multiply(x,y,x)`, `multiply(y,y,x)`, or `multiply(x,x,x)`. This last simply squares `x`. Note that the first parameters are by convention always (usually) the inputs to the routines. Routines are provided not only to allow arithmetic on big and flash numbers, but also to allow these variables to perform arithmetic with the built-in integer and double precision data-types.

Conversion routines are provided to convert from one type to another. For details of each routine see the relevant documentation in the reference manual and the example programs of Chapter 8.

Input and output to a file or I/O device is handled by the routines `innum`, `otnum`, `cinnum` and `cotnum`. The first two use the fixed number base specified by the user in the initial call of `mirsys`. The latter pair work in conjunction with the instance variable `IOBASE` which can be assigned dynamically by the user. A simple rule is that if the program is CPU bound, or involves changes of base, then set the base initially to `MAXBASE` (or 0 if a full-width base is possible — see Chapter 4) and use `cinnum` and `cotnum`. If on the other hand the program is I/O bound, or needs access to individual digits of numbers (using `getdig`, `putdig` and `numdig`), use `innum` and `otnum`.

Input and output to/from a character string is also supported in a similar fashion by the routines `instr`, `otstr`, `cinstr` and `cotstr`. The input routines can be used to set big or flash numbers to large constant values. By outputting to a string, formatting can take place prior to actual output to a file or I/O device.

Numbers to bases up to 256 can be represented. Numbers up to base 60 use as many of the symbols 0–9, A–Z, a–x as necessary.

A number base of 64 enforces standard base64 encoding. On output base64 numbers are padded with trailing `=` symbols if needed, but not otherwise formatted. On input white-space characters are skipped, and padding ignored. Do not use base64 with flash numbers. Do not use base64 for outputting negative numbers, as the sign is ignored.

If the base is greater than 60 (and not 64), the symbols used are the ASCII codes 0–255.

A base of 256 is useful when it is necessary to interpret a line of text as a large integer, as is the case for the Public Key Cryptography programs described in Chapter 8. The routines `big_to_bytes` and `bytes_to_big` allow for direct conversion from the internal big format to/from pure binary.

Strings are normally zero-terminated. However a problem arises when using a base of 256. In this case every digit from 0–255 can legitimately occur in a number. So a 0 does not necessarily indicate the end of the string. On input another method must be used to indicate the number of digits in the string.

By setting the instance variable `INPLEN = 25` (for example), just prior to a call to `innum` or `instr`, input is terminated after 25 bytes are entered. `INPLEN` is initialised to 0, and reset to 0 by the relevant routine before it returns.

For example, initialise MIRACL to use bigs of 400 bytes

```
    miracl *mip = mirsys(400, 256);
```

Internal calculations are very efficient using this base.

Input an ASCII string as a base 256 number. This will be zero-terminated, so no need for `INPLEN`.

```
    innum(x, stdin);
```

Now it is required to input exactly 1024 random bits

```
    mip->INPLEN = 128;
    innum(y, stdin);
```

But we want to see output in HEX

```
    mip->IOBASE = 16;
    cotnum(w, stdout);
```

Now in base64

```
    mip->IOBASE = 64;
    cotnum(w, stdout);
```

Rational numbers may be input using either a radix point (e.g 0.3333) or as a fraction (e.g. $^1/_3$). Either form can be used on output by setting the instance variable `RPOINT=ON` or `=OFF`.

Chapter 4

Internal Representation

Conventional computer arithmetic facilities as provided by most computer language compilers usually provide one or two floating-point data types (e.g. single and double precision) to represent all the real numbers, together with one or more integer types to represent whole numbers. These built-in data-types are closely related to the underlying computer architecture, which is sensibly designed to work quickly with large amounts of small numbers, rather than slowly with small amounts of large numbers (given a fixed memory allocation). Floating-point allows a relatively small binary number (e.g. 32 bits) to represent real numbers to an adequate precision (e.g. 7 decimal places) over a large dynamic range. Integer types allow small whole numbers to be represented directly by their binary equivalent, or in 2's complement form if negative. Nevertheless this conventional approach to computer arithmetic has several disadvantages.

- Floating-point and integer data-types are incompatible. Note that the set of integers, although infinite, is a subset of the rationals (i.e. fractions), which is in turn a subset of the reals. Thus every integer has an equivalent floating-point representation. Unfortunately these two representations will in general be different. For example a small positive whole number will be represented by its binary equivalent as an integer, and as separated mantissa and exponent as a floating-point. This implies the need for conversion routines, to convert from one form to the other.
- Most rational numbers cannot be expressed exactly (e.g. $1/3$). Indeed the floating-point system can only express exactly those rationals whose denominators are multiples of the factors of the underlying radix. For example our familiar decimal system can only represent exactly those rational numbers whose denominators are multiples of 2 and 5; $1/20$ is 0.05 exactly, $1/21$ is $0.0476190476190\ldots$
- Rounding in floating-point is base-dependant and a source of obscure errors.
- The fact that the size of integer and floating-point data types are dictated by the computer architecture, defeats the efforts of language designers to keep their languages truly portable.
- Numbers can only be represented to a fixed machine-dependent precision. In many applications this can be a crippling disadvantage, for example in the new and growing field of Public-Key cryptography.

- Base-dependent phenomena cannot easily be studied. For example it would be difficult to access a particular digit of a decimal number, as represented by a traditional integer data-type.

Herein is described a set of standard C routines which manipulate multi-precision rational numbers directly, with multi-precision integers as a compatible subset. Approximate real arithmetic can also be performed.

The two new data-types are called `big` and `flash`. The former is used to store multi-precision integers, and the latter stores multi-precision fractions as numerator and denominator in 'floating-slash' form. Both take the form of a fixed length array of digits, with sign and length information encoded in a separate 32-bit integer. The data type defined as `mr_small` used to store the number digits will be one of the built in types, for example `int`, `long` or even `double`. This is referred to as the "underlying type".

Both new types can be introduced into the syntax of the C language by the C statements

```
struct bigtype
{
    mr_unsign32 L;
    mr_small *d;
};

typedef struct bigtype *big;
typedef struct bigtype *flash;
```

Now `big` and `flash` variables can be declared just like any built-in data type, e.g.

```
    big x, y[10], z[10][10];
```

Observe that a `big` is just a pointer. The memory needed for each `big` or `flash` number instance is taken from the heap (or from the stack). Therefore each `big` or `flash` number must be initialised before use, and the required memory assigned to it.

Note that the user of these data-types is not concerned with this internal representation; the library routines allow `big` and `flash` numbers to be manipulated directly.

The structure of `big` and `flash` numbers is illustrated in Figure 4.

These structures combine ease of use with representational efficiency. A denominator of length zero ($d = 0$), implies an actual denominator of one; and similarly a numerator of length zero ($n = 0$) implies a numerator of one. Zero itself is uniquely defined as the number whose first element is zero (i.e. $n = d = 0$).

Note that the slash in the `flash` data-type is not in a fixed position, and may 'float' depending on the relative size of numerator and denominator.

A `flash` number is manipulated by splitting it up into separate `big` numerator and denominator components. A `big` number is manipulated by extracting and operating on each of its component integer elements. To avoid possible overflow, the numbers in each element are normally limited to a somewhat smaller range than that of the full word-length, e.g. 0 to 32767 ($= 2^{15} - 1$) on a 16-bit computer. However with

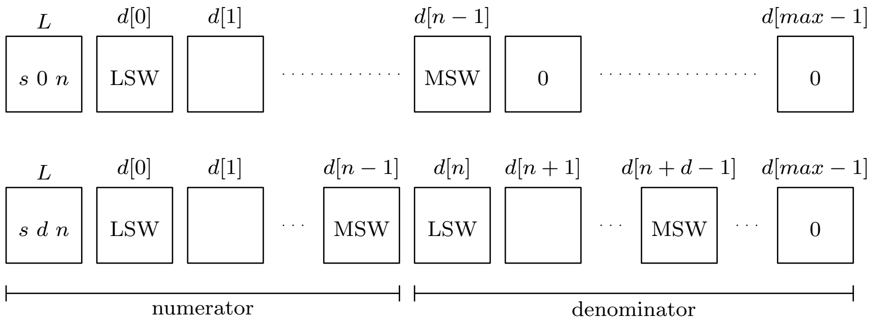

Figure 4.1: Structure of `big` and `flash` data-types where s is the sign of the number, n and d are the lengths of the numerator and denominator respectively, and LSW and MSW mean 'Least significant word' and 'Most significant word' respectively

careful programming a full-width base of 2^{16} can also be used, as the C language does not report a run-time error on integer overflow [Scott89b].

When the system is initialised the user specifies the fixed number of words (or bytes) to be assigned to all `big` or `flash` variables, and the number base to be used. Any base can be used, up to a maximum which is dependant on the wordlength of the computer used. If requested to use a small base b, the system will, for optimal efficiency, actually use base bn, where n is the largest integer such that bn fits in a single computer word. Programs will in general execute fastest if a full-width base is used (achieved by specifying a base of 0 in the initial call to `mirsys`). Note that this mode may be supported by extensive in-line assembly language for certain popular compiler/processor combinations, in certain time-critical routines, for example if using Borland/Turbo C with an 80x86 processor. Examine, for example, the source code in module mrarth1.c.

The encoding of the sign and numerator and denominator size information into a single word is possible, as the C language has standard constructs for bit manipulation.

Chapter 5

Implementation

No great originality is claimed for the routines used to implement arithmetic on the `big` data-type. The algorithms used are faithful renditions of those described by Knuth [Knuth81]. However some effort was made to optimise the implementation for speed. At the heart of the time-consuming multiply and divide routines there is, typically, a need to multiply together a digit from each operand, add in a 'carry' from a previous operation, and then separate the total into a digit of the result, and a 'carry' for the next operation. To illustrate consider this base 10 multiplication:

$$\begin{array}{r} 8723536221 \\ \times 9 \\ \hline 78511825989 \end{array}$$

To correctly process the column with the 5 in it, we multiply $5 \times 9 = 45$, add in the 'carry' from the previous column (a 3), to give 48, keep the 8 as the result for this column, and carry the 4 to the next column.

This basic primitive operation is essentially the calculation of the quotient $(ab+c)/m$ and its remainder. For the example above $a = 5$, $b = 9$, $c = 3$ and $m = 10$. This operation has surprisingly universal application, and since it lies at the innermost loop of the arithmetic algorithms, its efficient implementation is essential.

There are three main difficulties with a high-level language general base implementation of this MAD (Multiply, Add and Divide) operation.

- It will be slow.
- Quotient and remainder are not available simultaneously as a result of the divide operation. Therefore the calculation must be essentially done twice, once to get the quotient, and once for the remainder.
- Although the operation results in two single digit quantities, the intermediate product $ab + c$ may be double-length. Indeed such a Multiply-Add and Divide routine can be used on all occasions when a double-length quantity would be required by the basic arithmetic algorithms. Note that the C language is blessed with a 'long' integer data-type which may in fact be capable of temporarily storing this product.

For these reasons it is best to implement this critical operation in the assembly language of the computer used, although a portable C version is possible. At machine-code level a transitory double-length result can often be dealt with, even if the C long data-type is not itself double-length (as is the case for most C compilers as implemented on 32-bit computers, for which `int`s and `long`s are both 32-bit quantities). For further details see the documentation in the file mrmuldv.any.

A criticism of the MIRACL system might be its use of fixed length arrays for its `big` and `flash` data types. This was done to avoid the difficult and time-consuming problems of memory allocation and garbage collection, which would be needed by a variable-length representation. However it does mean that when doing a calculation on `big` integers that the results of all intermediate calculations must be less than or equal to the fixed size initially specified to `mirsys`.

In practise most numbers in a stable integer calculation are of more or less the same size, except when two are multiplied together in which case a double-length intermediate product is created. This is usually immediately reduced again by a subsequent divide operation. A classic example of this would be in the Pollard-Brent factoring program (Chapter 8).

Note that this is another manifestation, on a macro level, of the problem mentioned above. It would be a pity to have to specify each variable to be twice as large as necessary, just to cope with these occasional intermediate products. For this reason a special Multiply, Add and Divide routine mad has been included in the MIRACL library. It has proved very useful when implementing large programs (like the Pomerance-Silverman-Montgomery factoring program, Chapter 8) on computers with limited memory.

As well as the basic arithmetic operations, routines are also provided:

1. to generate and test `big` prime numbers, using a probabilistic primality test [Knuth81]
2. to generate `big` and `flash` random numbers, based on the subtract-with-borrow generator [Marsaglia]. Note however that the basic random number generator implemented internally is not cryptographicly secure. In a real cryptographic application it would not be adequate. A Cryptographicly strong generator is provided in the module mrstrong.c
3. to calculate powers and roots
4. to implement both the normal and extended Euclidean GCD (Greatest Common Divisor) algorithm [Knuth81]
5. to implement the 'Chinese Remainder Theorem' [Knuth81], and to calculate the Jacobi Symbol [Reisel].
6. to multiply extremely large numbers, using the Fast Fourier Transform method [Pollard71].

When performing extensive modular arithmetic, a time-critical operation is that of 'Modular Multiplication', that is multiplication of two numbers followed by reduction to the remainder when divided by a fixed n, the modulus. One obvious solution would be to use the mad routine described above. However Montgomery [Monty85] has proposed an alternative method. This requires that numbers are first converted to a special n-residue form. However once in this form modular multiplication is somewhat faster, using a special routine that requires no division whatsoever. When the calculation is complete, the answers can be converted back to normal form. Note that modular addition and subtraction of n-residues proceeds as usual, using the same routines as used for normal arithmetic. Given the requirement for conversion of variables to/from n-residue format, Montgomery's method should only be considered when a calculation requires an extensive amount of modular arithmetic using the same modulus. It is in fact much more convenient to use in a C++ environment, which hides these difficult details. See Chapter 7.

Montgomery arithmetic is used internally by many of the MIRACL library routines that require extensive modular arithmetic, such as the highly optimised modular exponentiation function powmod, and those functions which implement GF(p) Elliptic Curve arithmetic. Details can be found in the reference manual.

For the fastest possible modular arithmetic, one must alas resort to assembly language, and to methods optimised for a particular modulus, or moduli of a particular size. A number of different techniques are supported and can be used. The first two methods, the Comba and KCM methods, are implemented in the files mrcomba.c and mrkcm.c respectively. These files are created from template files mrcomba.tpl and mrkcm.tpl by inserting macros defined in a .mcs file. This is done automatically using the supplied macro expansion utility mex. Compile and run config.c on your target system to automatically create a suitable mirdef.h and for advise on how to proceed. Also read kcmcomba.txt. To get the fastest possible performance for your embedded application it is recommended that you should develop your own x.mcs file, if one is not already provided for your processor/compiler.

Two other rather more experimental techniques are implemented in the files mr87v.c and mr87f.c for the Intel 80x86 family of processors only, using the Borland C++ compiler.

If conditions are right the appropriate code will be automatically invoked by calling for example powmod.

It is important to note that the four techniques described require a compiler that supports in-line assembly. Furthermore the latter two techniques have only been tested with the Borland C++ V4.5 compiler for the 80x86 family of processors.

The first idea is to completely unravel and reorganise the program loops implicit in the multiplication and reduction process, as first advocated by [Comba] and modified by [Scott96]. See mrcomba.tpl. A fixed length modulus must be used and specified at compile time by defining MR_COMBA to the modulus size (in words) in mirdef.h. This works well for small to medium size moduli, particularly as used in GF(p) elliptic curve cryptography. For even more speed, the modular reduction algorithm can be optimised for a modulus that has a particularly simple form. This can be done by manually inserting the appropriate code into mrcomba.tpl. Example code for the case of a modulus $p = 2^{192} - 2^{64} - 1$ is given there in the routine comba_redc. To invoke this special code MR_SPECIAL must be defined in mirdef.h.

This technique can be combined with Karatsuba's idea for fast multiplication [Knuth81] to speed up modular multiplication for larger moduli [WeiDai]. This Karatsuba-Comba-Montgomery (KCM) method is invoked by defining MR_KCM in mirdef.h. The modulus size in computer words is restricted to be equal to MR_KCM*$2n$ for any positive n (within reason). This is a consequence of using Karatsuba's algorithm. For example defining MR_KCM to be 8 on a 32-bit computer allows popular modulus sizes of 512, 1024, 2048, . . . bits.

Another alternative is to exploit the floating point co-processor (if there is one), as its multiplication instruction is often faster than that of the integer unit [Rubin]. This is the case for the original Intel Pentium processor whose embedded co-processor takes only 3 cycles to perform a multiplication, compared with the 10 required for an integer multiply, although this is not true of the Pentium Pro, II, or III. Also the co-processor has eight extra registers, and can manipulate 64-bit numbers directly. These features allow the programmer some extra flexibility, which can be used to

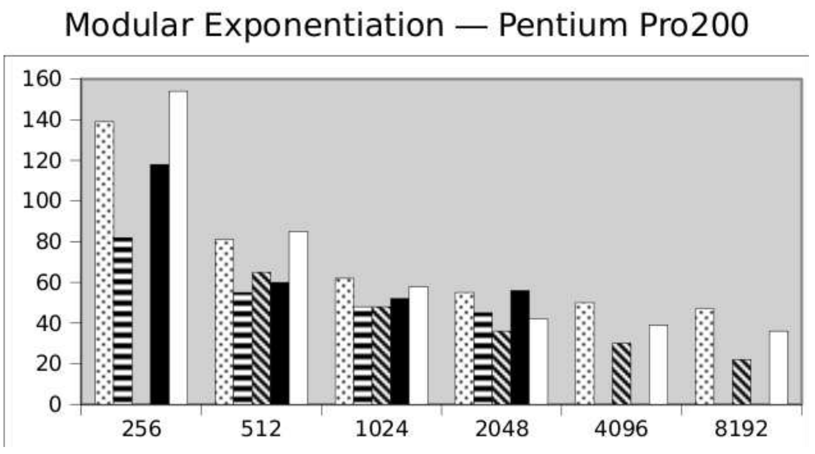

Figure 5.1: Modular exponentiation on a Pentium Pro200

advantage. Some experimental code has been written in the modules mr87f.c and mr87v.c, which may be exploited by defining `MR_PENTIUM` in mirdef.h. Use config.c to generate mirdef.h — this time the underlying type must be chosen as `double`. The module mr87v.c implements compact looping code, which will work with any modulus less than a certain maximum. The module mr87f.c unrolls the loops for more speed, but is bulkier and requires a fixed size modulus. Note that these modes of operation are incompatible with a full-width base, and work best with a number base of (usually) 2^{28} or 2^{29} — config.c will work it out for you. Note also that although this method will speed modular exponentiation on a Pentium, it may actually be slower for most other 80x86 processors, so use with care. In one test a 2048 bit number was raised to a 2048-bit power, mod a 2048 bit modulus. This took 2.4 seconds on a 60MHz Pentium.

Figure 5 illustrates the relative timings required by each method on a Pentium Pro 200MHz processor when compiled with the Borland C 32 bit compiler. The base line "Classic" method refers to the assembly language code implemented directly in mrarth2.c and mrmonty.c. The Comba and KCM implementations use assembly language from the ms86.mcs file. The modulus sizes are on the x axis, and the scaled time in seconds on the y axis. Note that in the calculation of $xy \pmod{n}$ it is assumed that x, y and n are randomly generated, all of same length in bits, and of no special form. It is assumed for example that the Comb optimisation technique (See [HAC] and brick.c) does not apply (that is x is a variable). The times shown are correct for the 8192 bit modulus. Times for smaller moduli are cumulatively scaled up by 8. So the times shown for a 4096 bit modulus should be divided by 8, for a 2048 bit modulus divided by 64, etc. Completely unrolled code is impracticable for the larger moduli, and hence timings for these methods are not given.

Note that the Comba method is optimal for moduli of 512 bits and less. This implies that it will be the optimal technique for fast GF(p) elliptic curve implementations, and for 1024-bit RSA decryption (which requires two 512-bit exponentiations and an application of the Chinese Remainder theorem). However these conclusions are processor-dependent, and may not be globally true. Also the Comba method can generate a lot of code, and this may be an important consideration in some applications. In some circumstances (for example when the instruction cache is very small), it may in fact be advisable to take the working unrolled assembly language and carefully, manually, re-roll it.

From Version 5.20 of MIRACL, a new data type is supported directly in C. This is called a `zzn2` type, and basically it consists of two `big`s in n-residue format

```
typedef struct
{
    big a;
    big b;
} zzn2;
```

where `a` and `b` can be considered as the real and imaginary parts respectively. The value of a `zzn2` is $a + ib$, where i is the imaginary square root of a quadratic non-residue. A `zzn2` variable is a representation of an element of a quadratic extension field with respect to a prime modulus p. For example if $p \equiv 3 \pmod 4$, then i can be taken as $\sqrt{-1}$, and the analogy to complex numbers with their real and imaginary parts becomes clear. They are particularly useful in implementations of cryptographic pairings. For an example of use, see the example program cardona.cpp which solves a cubic equation. A default value for the quadratic non-residue (which depends on the modulus) is stored in the instance variable `qnr`. Only the values -1 and -2 are currently supported.

To assist programmers generating code for a processor in a non-standard environment (e.g. an embedded controller), the code for dynamic memory allocation is always invoked from the module mralloc.c. By default this calls the standard C run-time functions `calloc` and `free`. However it can easily be modified to use an alternative user-defined memory allocation mechanism. For the same reason all screen/keyboard output and input is via the standard run-time functions `fputc` and `fgetc`. By intercepting calls to these functions, I/O can be redirected to non-standard devices.

Chapter 6

Floating-Slash numbers

The straightforward way to represent rational numbers is as reduced fractions, as a numerator and denominator with all common factors cancelled out. These numbers can then be added, subtracted, multiplied and divided in the obvious way and the result reduced by dividing both numerator and denominator by their Greatest Common Divisor. An efficient GCD subroutine, using Lehmers modification of the classical Euclidean algorithm for multiprecision numbers [Knuth81], is included in the MIRACL package.

An alternative way to represent rationals would be as a finite continued fraction [Knuth81]. Every rational number $^{p}/_{q}$ can be written as

$$\frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 \dots}}}$$

or more elegantly as $^{p}/_{q} = [a_0/a_1/a_2/\dots/a_n]$ where the a_i are positive integers, usually quite small.

For example

$$\frac{277}{642} = [0/2/3/6/1/3/3]$$

Note that the a_i elements of the above continued fraction representation are easily found as the quotients generated as a by-product when the Euclidean GCD algorithm is applied to p and q.

As we are committed to fixed length representation of rationals, a problem arises when the result of some operation exceeds this fixed length. There is a necessity for some scheme of truncation, or rounding. While there is no obvious way to truncate a large fraction, it is a simple matter to truncate the continued fraction representation. The resulting, smaller, fraction is called a best rational approximation, or a convergent, to the original fraction.

Consider truncating $^{277}/_{642} = [0/2/3/6/1/3/3]$. Simply drop the last element from the CF representation, giving $[0/2/3/6/1/3] = {}^{85}/_{197}$, which is a very close approximation to $^{277}/_{642}$ (error = 0.0018%). Chopping more terms from the CF expansion gives the successive convergents as $^{22}/_{51}$, $^{19}/_{44}$, $^{3}/_{7}$, $^{1}/_{2}$, $^{0}/_{1}$. As the fractions get smaller, the error increases. Obviously the truncation rule for a computer implementation should be to choose the biggest convergent that fits the computer representation.

The type of rounding described above is also called 'Mediant rounding'. If $^p/_q$ and $^r/_s$ are two neighbouring representable slash numbers astride a gap, then their mediant is the unrepresentable $^{p+r}/_{q+s}$. All larger fractions between $^p/_q$ and the mediant will round to $^p/_q$, and those between $^r/_s$ and the mediant will round to $^r/_s$. The mediant itself rounds to the 'simpler' of $^p/_q$ and $^r/_s$.

This is theoretically a very good way to round, much better than the rather arbitrary and base-dependent methods used in floating-point arithmetic, and is the method used here. The full theoretical basis of floating-slash arithmetic is described in detail by Matula and Kornerup [Matula85]. It should be noted that our `flash` representation is in fact a cross between the fixed- and floating-slash systems analysed by Matula and Kornerup, as our slash can only float between words, and not between bits. However the characteristics of the `flash` data-type will tend to those of floating-slash, as the precision is increased.

The MIRACL routine mround implements mediant rounding. If the result of an arithmetic operation is the fraction $^p/_q$, then the Euclidean GCD algorithm is applied as before to p and q. However this time the objective is not to use the algorithm to calculate the GCD per se, but to use its quotients to build successive convergents to $^p/_q$. This process is stopped when the next convergent is too large to fit the flash representation. The complete algorithm is given below (Kornerup and Matula [Korn83])

Given $p \geq 0$ and $q \geq 1$,

$$\begin{array}{lll} b_{-2} = p & x_{-2} = 0 & y_{-2} = 1 \\ b_{-1} = q & x_{-1} = 1 & y_{-1} = 0\,. \end{array}$$

Now for $i = 0, 1, \ldots$ and for $b_{i-1} > 0$, find the quotient a_i and remainder b_i when b_{i-2} is divided by b_{i-1}, such that

$$b_i = -a_i b_{i-1} + b_{i-2}\,.$$

Then calculate

$$\begin{aligned} x_i &= a_i x_{i-1} + x_{i-2} \\ y_i &= a_i y_{i-1} + y_{i-2}\,. \end{aligned}$$

Stop when $^{x_i}/_{y_i}$ is too big to fit the `flash` representation, and take $^{x_{i-1}}/_{y_{i-1}}$ as the rounded result.

If applied to $^{277}/_{642}$, this process will give the same sequence of convergents as stated earlier.

Since this rounding procedure must be applied to the result of each arithmetic operation, and since it is potentially rather slow, a lot of effort has been made to optimise its implementation. Lehmer's idea of operating only with the most significant piece of each number for as long as possible [Knuth81] is used, so that for most of the iterations only single-precision arithmetic is needed. Special care is taken to avoid the rounded result overshooting the limits of the `flash` representation [Scott89a]. The application of the basic arithmetic routines to the calculation of elementary functions such as $\log(x)$, $\exp(x)$, $\sin(x)$, $\cos(x)$, $\tan(x)$ etc., uses the fast algorithms described by Brent [Brent76].

In many cases the result given by a program can be guaranteed to be exact. This can be checked by testing the instance variable `EXACT`, which is initialised to `TRUE` and is only set to `FALSE` if any rounding takes place.

A disadvantage of using a `flash` type of variable to approximate real arithmetic is the non-uniformity in gap-size between representable values (Matula and Kornerup [Matula85]).

To illustrate this consider a floating-slash system which is constrained to have the product of numerator and denominator less than 256. Observe that the first representable fraction less than $^1/_1$ in such a system is $^{15}/_{16}$, a gap of $^1/_{16}$. The next fraction larger than $^0/_1$ is $^1/_{255}$, a gap of $^1/_{255}$. In general, for a k-bit floating-slash system, the gap size varies from smaller than 2^{-k} to a worst case $2^{-k/2}$. In practise this means that a real value that falls into one of the larger gaps, will be represented by a fraction which will be accurate to only half its usual precision. Fortunately such large gaps are rare, and increasingly so for higher precision, occurring only near simple fractions. However it does mean that real results can only be completely trusted to half the given decimal places. A partial solution to this problem would be to represent rationals directly as continued fractions. This gives a much better uniformity of gap-size (Kornerup and Matula [Korn85]), but would be very difficult to implement using a high level language.

Arithmetic on `flash` data-types is undoubtedly slower than on an equivalent sized multiprecision floating-point type (e.g. [Brent78]). The advantages of the flash approach are its ability to exactly represent rational numbers, and do exact arithmetic on them. Even when rounding is needed, the result often works out correctly, due to the tendency of mediant-rounding to prefer a simple fraction over a complex one. For example the roots program (Chapter 8) when asked to find the square root of 2 and then square the result, comes back with the exact answer of 2, despite much internal rounding.

WARNING! Do NOT mix `flash` arithmetic with the built-in double arithmetic. They don't mix well. If you decide to use flash arithmetic, use it throughout, and convert all constants at the start to type `flash`. Even better specify such constants if possible as fractions. So (in C++) it is much preferable to write

```
    x = Flash(5, 8);   // x = 5/8
```

rather than

```
    x = .625;
```

Chapter 7

The C++ Interface

Many users of the MIRACL package would be disappointed that they have to calculate

$$t = x^2 + x + 1$$

for a flash variable x by the sequence

```
    fmul(x, x, t);
    fadd(t, x, t);
    fincr(t, 1, 1, t);
```

rather than by simply

```
    t = x*x + x + 1;
```

Someone could of course use the MIRACL library to write a special purpose C compiler which could properly interpret such an instruction (see Cherry and Morris [Cherry] for an example of this approach). However such a drastic step is not necessary. A superset of C, called C++ has gained general acceptance as the natural successor to C. The enhancements to C are mainly aimed at making it an object-oriented language. By defining `big` and `flash` variables as 'classes' (in C++ terminology), it is possible to 'overload' the usual mathematical operators, so that the compiler will automatically substitute calls to the appropriate MIRACL routines when these operators are used in conjunction with `big` or `flash` variables. Furthermore C++ is able to look after the initialisation (and ultimate elimination) of these data-types automatically, using its constructor/destructor mechanism, which is included with the class definition. This relieves the programmer from the tedium of explicitly initialising each `big` and `flash` variable by repeated calls to `mirvar`. Indeed once the classes are properly defined and set up, it is as simple to work with the new data-types as with the built-in double and int types. Using C++ also helps shield the user from the internal workings of MIRACL.

The MIRACL library is interfaced to C++ via the header files big.h, flash.h, zzn.h, gf2m.h, ecn.h and ec2.h. Function implementation is in the associated files big.cpp, flash.cpp, zzn.cpp, gf2m.cpp, ecn.cpp and ec2.cpp, which must be linked into any application that requires them. The Chinese Remainder Theorem is also elegantly implemented as a class, in files crt.h and crt.cpp. See decode.cpp for an example of use. The Comb method for fast modular exponentiation with precomputation [HAC] is implemented in brick.h. See brick.cpp for an example of use. The GF(p) elliptic curve equivalents are in ebrick.h and ebrick.cpp and the GF(2^m) elliptic curve equivalents in ebrick2.h and ebrick2.cpp respectively.

EXAMPLE

```
/*
 *   Program to calculate factorials.
 */

#include <iostream>
#include "big.h"   /* include MIRACL system */

using namespace std;

Miracl precision(500,10); // This makes sure that MIRACL
                          // is initialised before main()
                          // is called
void main()
{
    /* calculate factorial of number */
    Big nf = 1;       /* declare "Big" variable nf */
    int n;
    cout << "factorial program\n";
    cout << "input number n= \n";
    cin >> n;
    while (n > 1)
        nf *= (n--);  /* nf = n! = n*(n-1)*(n-2)*....3*2*1  */
    cout << "n!= \n" << nf << "\n";
}
```

Compare this with the C version of Chapter 3. Note the neat use of a dummy class `Miracl` used to set the precision of the `big` variables. Its declaration at global scope ensures that MIRACL is initialised before `main()` is called. (Note that this would not be appropriate in a multi-threaded environment.) When compiling and linking this program, don't forget to link in the `Big` class implementation file big.cpp.

Conversion to/from internal `Big` format is quite important:

To convert a hex character string to a `Big`

```
    Big x;
    char c[100];
    ...
    mip->IOBASE = 16;
    x = c;
```

To convert a `Big` to a hex character string

```
    mip->IOBASE = 16;
    c << x;
```

To convert to/from pure binary, use the `from_binary()` and `to_binary()` friend functions.

```
int len;
char c[100];
...
Big x = from_binary(len, c);
    // creates Big x from len bytes of binary in c
len = to_binary(x, 100, c, FALSE);
    // converts Big x to len bytes binary in c[100]
len = to_binary(x, 100, c, TRUE);
    // converts Big x to len bytes binary in c[100]
    // (right justified with leading zeros)
```

In many of the example programs, particularly the factoring programs, all the arithmetic is done $\pmod n$. To avoid the tedious reduction $\pmod n$ required after each operation, a new C++ class `ZZn` has been used, and defined in the file zzn.h. This class `ZZn` (for $\mathbb{Z}_n$ or the ring of integers $\pmod n$) has its arithmetic operators defined to automatically perform the reduction. The function `modulo(n)` sets the modulus. In an analogous fashion the C++ class `GF2m` deals with elements of the field defined over GF(2^m). In this case the "modulus" is set via `modulo(m,a,b,c)`, which also specifies either a trinomial basis $t^m + t^a + 1$, (and set $b = c = 0$), or a pentanomial basis $t^m + t^a + t^b + t^c + 1$. See the IEEE P1363 documentation for details: `http://grouper.ieee.org/groups/1363/draft.html`.

Internally the `ZZn` class uses Montgomery representation. See zzn.h. Note that the internal implementation of `ZZn` is hidden from the application programmer, a classic feature of C++. Thus the awkward internals of Montgomery representation need not concern the C++ programmer.

The class `ECn` defined in ecn.h makes manipulation of points on GF(p) elliptic curves a simple matter, again hiding all the grizzly details. The class `EC2` defined in ec2.h does the same for GF(2^m) elliptic curves.

Almost all of MIRACL's functionality is accessible from C++. Programming can often be done intuitively, without reference to the manual, using familiar C syntax as illustrated above. Other functions are accessed using the 'obvious' syntax — as in for example `x=gcd(x,y)`, or `y=sin(x)`. For more details examine the header files and example programs.

C++ versions of most of the example programs are included in the distribution media, with the file extensions .cpp.

One problem with manipulating large objects in C++ is the tendency of the compiler to generate code to create/destroy/copy multiple temporary objects. By default MIRACL obtains memory for `Big` and `Flash` variables from the heap. This can be quite time-consuming, and all such objects need ultimately to be destroyed. It would be faster to assign memory instead from the stack, especially for relative small big numbers. This can now be achieved by defining `BIGS=m` at compilation time. For example if using the Microsoft C++ compiler from the command line:

```
C:\miracl>cl /O2 /GX /DBIGS=50 brent.cpp big.cpp zzn.cpp miracl.lib
```

Note that the value of `m` should be the same as or less than the value of `n` that is specified in the call to `mirsys(n,0)` or in `Miracl precision=n` in the main program.

When using finite-field arithmetic, valid numbers are always less than a certain fixed modulus. For example in the finite field (mod n), the class defined in zzn.h and zzn.cpp might handle numbers with respect to a 512-bit modulus n, which is set by `modulo(n)`. In this case one can define `ZZNS=16` so that all elements are of a size $16 \times 32 = 512$, and are created on the stack. (This works particularly well in combination with the Comba mechanism described in Chapter 5)

In a similar fashion, when working over the field GF(2^{283}), one can define `GF2MS=9`, so that all elements in the field are stored in a fixed memory allocation of 9 words taken from the stack.

In these latter two cases the precision `n` specified in the call to `mirsys(n,0)` or in `Miracl precision=n` in the main program should be at least 2 greater than the `m` that specified in the `ZZNS=m` or `GF2MS=m` definition.

This is not recommended for program development, or if the objects are very large. It is only relevant with C++ programs. See the comments in the sample programs ibe_dec.cpp and dl.cpp for examples of the use of this mechanism. However the benefits can often be substantial — programs may be up to twice as fast.

Finally here is a more elaborate C++ program to implement a relatively complex cryptographic protocol. Note the convention of using capitalised variables for field elements.

```
/*
 *   Gunthers's ID based key exchange - Finite field version
 *   See RFC 1824
 *   r^r variant (with Perfect Forward Security)
 */

#include <iostream>
#include <fstream>
#include "zzn.h"

using namespace std;

Miracl precision = 100;

char *IDa = "Identity 1";
char *IDb = "Identity 2";

// Hash function
Big H(char *ID)
{ // hash character string to 160-bit big number
    int b;
    Big h;
    char s[20];
    sha sh;
    shs_init(&sh);
    while (*ID != 0) shs_process(&sh, *ID++);
    shs_hash(&sh, s);
    h = from_binary(20, s);
    return h;
}

int main()
```

```
{
    int bits;
    ifstream common("common.dss");    // construct file stream
    Big p,q,g,x,k,ra,rb,sa,sb,ta,tb,wa,wb;
    ZZn G,Y,Ra,Rb,Ua,Ub,Va,Vb,Key;
    ZZn A[4];
    Big b[4];
    long seed;
    miracl *mip = &precision;
    cout << "Enter 9 digit random number seed  = ";
    cin >> seed; irand(seed);

    // get common data. Its in hex. G^q mod p = 1
    common >> bits;
    mip->IOBASE = 16;
    common >> p >> q >> g;
    mip->IOBASE = 10;
    modulo(p);    // set modulus

    G = (ZZn) g;

    cout << "Setting up Certification Authority ... " << endl;

    // CA generates its secret and public keys

    x = rand(q);        // CA secret key, 0 < x < q
    Y = pow(G, x);      // CA public key, Y=G^x

    cout << "Visiting CA ...." << endl;

    // Visit to CA - a
    k = rand(q);
    Ra = pow(G, k);
    ra = (Big) Ra % q;
    sa = (H(IDa) + (k * ra) % q);
    sa = (sa * inverse(x, q)) % q;

    // Visit to CA - b
    k = rand(q);
    Rb = pow(G, k);
    rb = (Big) Rb % q;
    sb = (H(IDb) + (k * rb) % q);
    sb = (sb * inverse(x, q)) % q;

    cout << "Offline calculations .... " << endl;

    // offline calculation - a
    wa = rand(q);
    Va = pow(G, wa);
    ta = rand(q);
    Ua = pow(Y, ta);

    // offline calculation - b
    wb = rand(q);
    Vb = pow(G, wb);
    tb = rand(q);
    Ub = pow(Y, tb);

    // Swap ID, R, U, V
    cout << "Calculate Key ... " << endl;

    // calculate key  a
    // Key = Vb^wa.Ub^sa.G^[(H(IDa)*tb)%q].Rb^[(rb*ta)%q] mod p

    rb = (Big) Rb % q;
    A[0] = Vb; A[1] = Ub; A[2] = G; A[3] = Rb;
    b[0] = wa; b[1] = sa; b[2] = (H(IDb) * ta) % q;
    b[3] = (rb * ta) % q;

    Key = pow(4, A, b);    // extended exponentiation
    cout << "Key= \n" << Key << endl;

    // calculate key - b
    ra = (Big) Ra % q;
    A[0] = Va; A[1] = Ua; A[2] = G; A[3] = Ra;
    b[0] = wb; b[1] = sb; b[2] = (H(IDa) * tb) % q;
    b[3] = (ra * tb) % q;

    Key = pow(4, A, b);    // extended exponentiation
    cout << "Key= \n" << Key << endl;
    return 0;
}
```

MIRACL has evolved quite a complex class hierarchy — see the diagram below. Where possible classes are built directly on top of the C/assembly core. Note the support for polynomials, power series and extension fields.

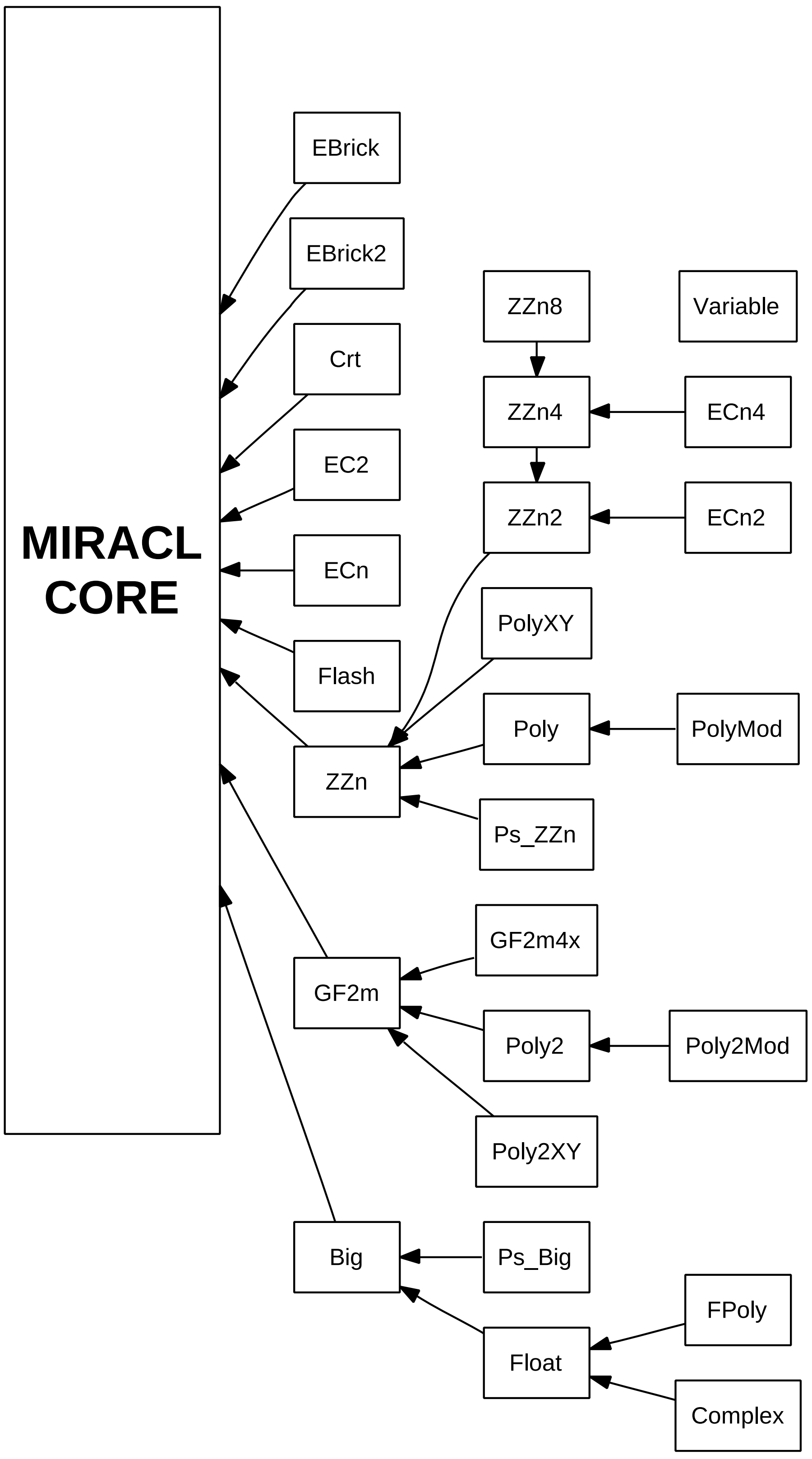

Figure 7.1: MIRACL's class diagram

Chapter 8

Example Programs

Note: The programs described here are of an experimental nature, and in many cases are not completely "finished off". For further information read the comments associated with the appropriate source file.

8.1 Simple Programs

hail.c

This program allows you to investigate so-called hailstone numbers, as described by Gruenberger [Gruen]. The procedure is simple. Starting with any number apply the following rules:

1. If it is odd, multiply it by 3 and add 1.
2. If it is even, divide it by 2.
3. Repeat the process, until the number becomes equal to 1, in which case stop.

It would appear that for any initial number this process always eventually terminates, although it has not been proved that this must happen, or that the process cannot get stuck in an infinite loop. What goes up, it seems, must come down. Try the program for an initial value of 27. Then try it using much bigger numbers, like 10709980568908647 (which has interesting behaviour).

palin.c

This programs allows one to investigate palindromic reversals [Gruen]. A palindromic number is one which reads the same in both directions. Start with any number and apply the following rules.

1. Add the number to the number obtained by reversing the order of the digits. Make this the new number.
2. Stop the process when the new number is palindromic.

It appears that for most initial numbers this process quickly terminates. Try it for 89. Then try it for 196.

mersenne.c

This program attempts to generate all prime numbers of the form $2^n - 1$. The largest known primes have always been of this form because of the efficiency of this Lucas-Lehmer test. The routine `fft_mult` is used, as it is faster for very large numbers.

8.2 Factoring Programs

Six different Integer Factorisation programs are included, covering all modern approaches to this classical problem. For more background and information on the algorithms used, see [Scott89c].

brute.c

This program attempts to factorise a number by brute force division, using a table of small prime numbers. When attempting a difficult factorisation it makes sense to try this approach first. Factorise 12345678901234567890 using this program. Then try it on bigger random numbers.

brent.c

This program attempts to factorise a number using the Brent-Pollard method. This method is faster at finding larger factors than the simple-minded brute force approach. However it will not always succeed, even for simple factorisations. Use it to factorise R17, that is 11111111111111111 (seventeen ones). Then try it on larger numbers that would not yield to the brute force approach.

pollard.c

Another factoring program, which implements Pollard's (p-1) method, specialises in quickly finding a factor p of a number N for which $p - 1$ has itself only small factors. Phase 1 of this method will work if all these small factors are less than `LIMIT1`. If Phase 1 fails then Phase 2 searches for just one final larger factor less than `LIMIT2`. The constants `LIMIT1` and `LIMIT2` are set inside the program.

williams.c

This program is similar to Pollards method, but can find a factor p of N for which $p + 1$ has only small factors. Again two phases are used. In fact this method is sometimes a $p + 1$ method, and sometimes a $p - 1$ method, so several attempts are made to hit on the $p + 1$ condition. The algorithm is rather more complex than that used in Pollards method, and is somewhat slower.

lenstra.c

Lenstra [Monty87] has discovered a new method of factorisation, generically similar to the Pollard and Williams methods, but potentially much more powerful. It works by randomly generating an Elliptic Curve, which can then be used to find a factor p of N, for which $p + 1 - \delta$ has only small factors, where δ depends on the particular curve chosen. If one curve fails then another can be tried, an option not possible with the Pollard/Williams methods. Again this is a two phase method, and although it has very good asymptotic behaviour, it is much slower than the Pollard/Williams methods for each iteration.

qsieve.c

This is a sophisticated Pomerance-Silverman-Montgomery [Pomerance], [Silverman] factoring program. which factors $F7 = 2^{128} + 1$

$$340282366920938463463374607431768211457$$

in less than 30 seconds, running on a 60MHz Pentium-based computer. When this number was first factored, it took 90 minutes on an IBM 360 mainframe (Morrison and Brillhart [Morrison]), albeit using a somewhat inferior algorithm.

Its speciality is factoring all numbers (up to about sixty digits long), irrespective of the size of the factors. If the number to be factored is N, then the program actually works with a number kN, where k is a small Knuth-Schroepel multiplier. The program itself works out the best value of k to use. Internally, the program uses a 'factor base' of small primes. The larger the number, the bigger will be this factor base. The program works by accumulating information from a number of simpler factorisations. As it progresses with these it prints out `working...n`. When it thinks it has enough information it prints out trying, but these tries may be premature and may not succeed. The program will always terminate before the number `n` in `working...n` reaches the size of the factor base.

This program uses much more memory than any of the other example programs, particularly when factoring bigger numbers. The amount of memory that the program can take is limited by the values defined for `MEM`, `MLF` and `SSIZE` at the beginning of the program. These limit the number of primes in the factor base, the number of 'larger' primes used by the so-called large-prime variation of the algorithm, and the sieve size respectively. They should be increased if possible, or reduced if your computer has insufficient memory. See [Silverman] for more details.

Use qsieve to factor 10000000000000000000000000000000009 (thirty-five digits).

factor.c

This program combines the above algorithms into a single general purpose program for factoring integers. Each method is used in turn in the attempt to extract factors. The number to be factored is given in the command line, as in factor 11111111111. The number can alternatively be specified as a formula, using the switch `-f`, as in `factor -f (10#11-1)/9`. The symbol # here means 'to the power of' (# is used instead of âs the latter symbol has a special meaning for DOS on an IBM PC). Type factor on its own for a full description of this and other switches that can be used to control the input/output of this program.

8.3 Discrete Logarithm Programs

Two programs implement Pollards algorithms [Pollard78] for extracting discrete logarithms. The discrete logarithm problem is to find x given y, r and n in

$$y = r^x \pmod{n}$$

The above is a good example of a one-way function. It is easy to calculate y given x, but apparently extremely difficult to find x given y. Pollard's algorithms however perform quite well under certain circumstances, if x is known to be small or if n is a prime p for which $p - 1$ has only small factors.

kangaroo.c

This program finds x in the above, assuming that x is quite small. The value of r is fixed (at 16), and the modulus n is also fixed inside the program. Initially a 'trap' is set. Subsequently the discrete logarithm can be found (almost certainly) for any number, assuming its discrete logarithm is less than a certain upper limit. The number of steps required will be approximately the square root of this limit.

genprime.c

A prime number p with known factorisation of $p - 1$ is generated by this program, for use by the index.c and identity.c programs described below. The factors of $p - 1$ are output to a file prime.dat.

index.c

This program implements Pollard's rho algorithm for extracting discrete logarithms, when the modulus n in the above equation is a prime p, and when $p - 1$ has only relatively small factors. The number of steps required is a function of the square root of the largest of these factors.

8.4 Public-Key Cryptography

Public Key Cryptography is a two key cryptographic system with the very desirable feature that the encoding key can be made publicly available, without weakening the strength of the cipher. The first example program demonstrates many popular public-key techniques. Then two functional Public-Key cryptography systems, whose strength appears to depend on the difficulty of factorisation, are presented. The first is the classic RSA system (Rivest, Shamir and Adleman [RSA]). This is fast to encode a message, but painfully slow at decoding. A much faster technique has been invented by Blum and Goldwasser. This probabilistic Public Key system is also stronger than RSA in some senses. For more details see [Brassard], who describes it as 'the best that academia has had to offer thus far'. For both methods the keys are constructed from 'strong' primes to enhance security. Closely associated with PK Cryptography, is the concept of the Digital Signature. A group of example programs implement the Digital Signature Standard, using classic finite fields and elliptic curves over both the fields GF(p) and $GF(2^m)$.

pk-demo.c

This program carries out a 1024-bit Diffie-Hellman key exchange, and then another Diffie-Hellman type key exchange, but this time based on a 160-bit prime and an elliptic curve. Next a test string is encrypted and decrypted using the El Gamal method. The program finishes with a 1024 bit RSA encryption/decryption of the same string. For a good description of all these techniques see [Stinson]. Anyone attempting to implement a PK system using MIRACL is strongly encouraged to examine this file, and its C++ counter-part pk-demo.cpp

bmark.c/imratio.c

The benchmarking program bmark.c allows the user to quickly determine the time that will be required to implement any of the popular public key methods. It can be compiled and linked with any of the variants of the MIRACL library, as specified in mirdef.h, to determine which gives the best performance on a particular platform for a particular PK method. The program imratio.c when compiled and run calculates the significant ratios S/M, I/M and J/M, where S is the time for a modular squaring, M the time for a modular multiplication, I the time for a modular inversion, and J the time for a Jacobi symbol calculation.

genkey.c

This program generates the 'public' encoding key and 'private' decoding keys that are necessary for both the original Rivest-Shamir-Adleman PK system and the superior Blum-Goldwasser method [Brassard]. These keys can take a long time to generate, as they are formed from very large prime numbers, which must be generated carefully for maximum security.

The size of each prime in bits is set inside the program by a `#define`. The security of the system depends on the difficulty of factoring the encoding 'public' key, which is formed from two such large primes. The largest numbers which can be routinely factored using hundreds of powerful computers are 430 bits long (1996). So a minimum size of 512 bits for each prime gives plenty of security (for now!)

After this program has run, the two keys are created in files PUBLIC.KEY and PRIVATE.KEY.

encode.c

Messages or files may be encoded with this program, which uses the 'public' encoding key from the file PUBLIC.KEY, generated by the program genkey, which must have been run prior to using this program. When run, the user is prompted for a file to encipher. Either supply the name of a text file, or press return to enter a message directly from the keyboard. In the former case the encoded output is sent to a file with the same name, but with the extension .RSA. In the latter case a prompt is issued for an output filename, which must be given. Text entered from the keyboard must be terminated by a `CONTROL-Z` (end-of-file character). Type out the encoded file and be impressed by how indecipherable it looks.

decode.c

Messages or files encoded using the RSA system may be decoded using this program, which uses the 'private' decoding key from the file PRIVATE.KEY generated by the program genkey which must have been run at some stage prior to using this program.

When run, the user is prompted for the name of the file to be decoded. Type in the filename (without an extension — the program will assume the extension .RSA) and press return. Then the user is asked for an output filename. Either supply a filename or press return, in which case the decoded output will be sent straight to the screen. A problem with the RSA system becomes immediately apparent — decoding takes quite a relatively long time! This is particularly true for larger key sizes and long messages.

enciph.c

This program works in an identical fashion to the program 'encode', except that it prompts for a random seed before encrypting the data. This random seed is then used internally to generate a larger random number. The encryption process depends on this random number, which means that the same data will not necessarily produce the same cipher-text, which is one of the strengths of this approach. As well as creating a file with a .BLG extension containing the encrypted data, a second small file (with the .KEY extension) is also produced.

deciph.c

This program works in an identical fashion to the program 'decode'. However it has the advantage that it runs much more quickly. There will be a significant initial delay while a rather complex calculation is carried out. This uses the private key and the data in the .KEY file to recover the large random number used in the encryption process. Thereafter deciphering is as fast as encipherment.

dssetup.c

A standard method for digital signature has been proposed by the American National Institute of Standards and Technology (NIST), and fully described in the Digital Signature Standard [DSS]. This program generates a prime q, another much larger prime $p = 2nq + 1$, (where n is random) and a generator g. This information is made common to all. This program generates the common information $\{p, q, g\}$ into a file common.dss.

limlee.c

It has been shown by Lim and Lee [LimLee] that for certain Discrete Logarithm based protocols (but not for the Digital Signature Standard) there is a weakness associated with primes of the kind generated by the dssetup.c program described above. To avoid these problems they recommend that p is of the form $p = 2p_1p_2p_3 \cdots q + 1$, where the p_i are primes greater than q. This program generates the values $\{p, q, g\}$ into a file common.dss, and can be used in place of dssetup.c. It is a little slower.

dssgen.c

Each individual user who wishes to digitally sign a computer file randomly generates their own private key $x < q$ and makes available a public key $y = g^x \pmod p$. The security of the system depends on the sizes of p and q (at least 512 bits and 160 bits respectively). This program generates a single public/private key pair in the files public.dss and private.dss respectively.

dssign.c

This program uses the private key from private.dss to 'sign' a document stored in a file. First the file data is 'hashed' down to a 160 bit number using SHA, the Standard Hash Algorithm. This is also specified by the NIST and is implemented in the provided module mrshs.c. The 160-bit hash is duly 'signed' as described in [DSS], and the signature, in the form of two 160-bit numbers, written out to a file. This file has the same name as the document file, but with the extension .dss.

dssver.c

This program uses the public key from public.dss to verify the signature associated with a file, as described in [DSS].

ecsgen.c, ecsign.c, ecsver.c

The Digital Signature technique can also be implemented using Elliptic Curves over the field GF(p) [Jurisic]. Common domain information in the order $\{p, A, B, q, X, Y\}$ is extracted from the file common.ecs created using one of the point-counting algorithms described below. These values specify an initial point (X, Y) on an elliptic curve $y^2 = x^3 + Ax + B \pmod p$ which has q points on it. The advantages are a much smaller public key for the same level of security. Smaller numbers can be used as the discrete logarithm problem is apparently much more difficult in the context of an elliptic curve. This in turn implies that elliptic curve arithmetic is also potentially faster. However the use of smaller numbers is somewhat offset by the more complex calculations involved.

This set of programs has the same functionality as those described above for the standard DSS. Note however that the file extension .ecs is used for all the generated files. Read the comments in the source files for more information.

ecsgen2.c, ecsign2.c, ecsver2.cpp

These programs provide the same functionality as those provided above, but use elliptic curves defined over the field GF(2^m). Domain information in this case is extracted from the file common2.ecs in the order $\{m, A, B, q, X, Y, a, b, c\}$, where (X, Y) specifies an initial point on the elliptic curve $y^2 = x^3 + Ax^2 + B$ defined over GF(2^m). The parameters of a trinomial or pentanomial basis are also specified, $t^m + t^a + 1$ or $t^m + t^a + t^b + t^c + 1$ respectively. In the former case b and $c$are zero. Finally $cf \times q$ specifies the number of points on the curve, the product of a large prime factor q

and a small cofactor cf. The latter is normally 2 or 4. The file common2.ecs can be created by the schoof2 program described below.

cm.cpp, schoof.cpp, mueller.cpp, process.cpp, sea.cpp, schoof2.cpp

A problem with Elliptic curve cryptography is the construction of suitable curves. This is actually much more difficult than the equivalent problem in the integer finite field as implemented by the program dssetup.c/dssetup.cpp. One approach is the Complex Multiplication method, as described in the Annex to the IEEE P1363 Standard Specifications for Public Key Cryptography (available from the Web). This is implemented here by the C++ program cm.cpp and its supporting modules float.cpp, complex.cpp, flpoly.cpp, poly.cpp, and associated header files.

The program when run uses command line arguments. Type `cm` on its own to get instructions. For example

```
cm -f 2#224-2#96+1 -o common.ecs
```

generates the common information needed to implement elliptic curve cryptography into the file common.ecs.

As an alternative to the CM method, a random curve can be generated, and the points on the curve directly counted. This is more time-consuming than complex multiplication, but may lead to more secure, less structured curves. The basic algorithm is due to Schoof [Sch],[Blake] and is only practical due to the use of Fast Fourier Transform methods [Shoup] for the multiplication/division of large degree polynomials. See mrfast.c. Its still very slow, much slower than cm. Type `schoof` on its own to get instructions. For example

```
schoof f 2#192-2#64-1 3 35317045537
```

counts the points on the curve $y^2 = x^3 - 3x + 35317045537 \pmod{2^{192} - 2^{64} - 1}$.

This curve is randomly selected (actually 35317045537 is my international phone number). The answer is the prime number

$$6277101735386680763835789423127240467907482257771524603027$$

Be prepared to wait, or. . .

Use the suite of programs mueller, process, and sea, which together implement the superior, but more complex, Schoof-Elkies-Atkin method for point counting. See [Blake] for details.

First of all the mueller program should be run, to generate the required Modular Polynomials. This needs to be done just once — ever. The greater your collection of Modular Polynomials, the greater the size of prime modulus that can be used for the elliptic curves of interest. Note that this program is particularly hard on memory resources, as well as taking a long time to run. However after an hour at most you should have enough Modular Polynomials to start experimenting. As with all these programs, simply typing the program name without parameters generates instructions for use. Also be sure to read the comments at the start of the source file, in this case mueller.cpp.

Next run the process application, which processes the file of raw modular polynomials output by mueller, for use with a specified prime modulus.

Finally run sea to count the points on the curve, and optionally to create a .ecs file as described above.

For example:

```
mueller 0 120 o mueller.raw
process f 65112*2#144-1 i mueller.raw o test160.pol
sea 3 49 i test160.pol
```

generates all the modular polynomials for primes from 0 to 120, and outputs them to the file mueller.raw. Then these polynomials are processed with respect to the prime $p = 65112 \times 2^{144} - 1$, to create the file test160.pol. Finally the main sea application counts the points on the curve $y^2 = x^3 - 3x + 49 \pmod{p}$.

This may be more complicated to use, but its much faster than schoof.

Read the comments at the start of sea.cpp for more information.

For elliptic curves over $\mathrm{GF}(2^m)$, the program schoof2 can be used, which is quite similar to schoof. It is even slower, but just about usable on contemporary hardware. For example

```
schoof2 1 52 191 9 o common2.ecs
```

counts the points on the curve $y^2 + xy = x^3 + x^2 + 52$, over the field $\mathrm{GF}(2^{191})$. A suitable irreducible basis must also be specified, in this case $t^{191} + t^9 + 1$. Tables of suitable bases can be found in many documents, for example in Appendix A of the IEEE P1363 standard. See [Menezes] for a description of the method.

For more information on building these applications see the files cm.txt, schoof.txt, schoof2.txt and sea.txt.

crsetup.cpp, crgen.cpp, crencode.cpp, crdecode.cpp

Public key schemes should ideally be immune from adaptive chosen cipher-text attacks, whereby an attacker is able to obtain decryptions of any presented cipher-texts other than the particular one they are interested in. Recently Cramer and Shoup [CS] have come up with a Public Key encryption method that is provably immune to such powerful attacks. The program crsetup creates various global parameters, and crgen generates one set of public and private keys in the files public.crs and private.crs respectively. To encrypt an ASCII file called for example fred.txt, run the crencode program that generates a random session key, and uses it to encrypt the file. This session key is in turn encrypted by the public key and stored in the file fred.key. The binary encrypted file itself is stored as fred.crs. To decrypt the file, run the crdecode program, which uses the private key to recover the session key, and hence decode the text to the screen.

A couple of points are worth highlighting. First of all the bulk encryption is carried out using a block cipher method. Such hybrid systems are standard practise, as block ciphers are much faster than public key methods. The block cipher scheme used is the new Advanced Encryption Standard block cipher, which is implemented in mraes.c.

Examination of the source code crdecode.cpp reveals that decryption is a two-pass process. On the first pass the program determines the validity of the cipher-text, and only after that is known to be valid does the program go on to decrypt the file. So the decryption procedure will not respond at all to arbitrary bit strings concocted by an attacker.

brick.c, ebrick.c, ebrick2.c

Certain Cryptographic protocols require the exponentiation of a fixed number g, that is the calculation of $g^x \pmod{n}$, where g and n are known in advance. In this case the calculation can be substantially speeded up by a precomputation which generates a small table of big numbers. The method was first described by Brickell et al [Brick]. The example program brick.c illustrates the method. The GF(p) elliptic curve equivalent is provided in ebrick.c and the GF(2^m) equivalent in ebrick2.c. In a typical application the precomputed tables might be generated using one of these programs (see commented-out code in ebrick2.c), which then might be transferred to ROM in an embedded program. The embedded program might use a static build of MIRACL to make use of these tables.

identity.c

This is a program that allows individuals, issued with certain secret information, to establish mutual keys by performing a calculation involving only the other correspondents publicly known identity. No interchange of data is required [Maurer], and so this is called Non-Interactive Key Exchange. Note that the 'publicly known identity' might, for example, be simply an email address. For a full description see [Scott92]. This example program generates the secret data from the proffered Identity. However before this program is run, the program genprime.c must be run twice, to generate a pair of suitable trap-door primes. Copy the output of the program, prime.dat, first to trap1.dat and then to trap2.dat. The product of these primes will be used as the composite modulus used for subsequent calculations.

Pairing-Based Cryptography

A number of experimental programs are provided to implement cryptographic protocols based on pairings. Notably there are examples of Identity-Based Encryption (IBE) and authenticated key exchange. Read the files pairings.txt, ake.txt and ibe.txt for details.

8.5 Flash Programs

Several programs demonstrate the use of `flash` variables. One gives an implementation of Gaussian elimination to solve a set of linear equations, involving notoriously ill-conditioned Hilbert matrices. Others show how rational arithmetic can be used to approximate real arithmetic, in, for example the calculation of roots and π. The former program detected an error in the value for the square root of 5 given in Knuth's appendix A [Knuth81]. The correct value is

$$2.236067977499789696409173668731276235440 6$$

The error is in the tenth last digit, which is a 2, and not a 1.

The roots program runs particularly fast when calculating the square roots of single precision integers, as a simple form of continued fraction generator can be used. In one test the golden ratio $(1 + \sqrt{5})/2$ was calculated to 100,000 decimal places in 3 hours of CPU time on a VAX11/780.

The sample program was used to calculate π correct to 1000 decimal places, taking less than a minute on a 25MHz 80386-based IBM PC to do so.

roots.c

This program calculates the square root of an input number, using Newton's method. Try using it to calculate the square root of two. The accuracy obtained depends on the size of the `flash` variables, specified in the initial call to `mirsys`. The tendency of flash arithmetic to prefer simple numbers can be illustrated by requesting, say, the square root of 7. The program calculates this value and then squares it, to give 7 again exactly. On your pocket calculator the same result will only be obtained if clever use is made of extra (hidden) guard digits.

hilbert.c

Traditionally the inversion of 'Hilbert' matrices is regarded as a tough test for any system of arithmetic. This programs solves the set of linear equations $Hx = b$, where H is a Hilbert matrix and b is the vector $[1, 1, 1, 1, \ldots, 1]$, using the classical Gaussian Elimination method.

sample.c

This program is the same as that used by Brent [Brent78] to demonstrate some of the capabilities of his Fortran Multiprecision arithmetic package. It calculates π, $\exp(\pi\sqrt{163/9})$, and $\exp(\pi\sqrt{163})$.

ratcalc.c

As a comprehensive and useful demonstration of `flash` arithmetic this program simulates a standard full-function scientific calculator. Its unique feature (besides its 36-digit accuracy) is its ability to work directly with fractions, and to handle mixed calculations involving both fractions and decimals. By using this program the user will quickly get a feel for `flash` arithmetic and its capabilities. Note that this program contains some non-portable code (screen handling routines) that must be tailored to each individual computer/terminal combination. The version supplied works only on standard PCs using DOS, or a command prompt window in Windows NT/98.

Chapter 9

Instance variables

These variables are all member of the miracl structure defined in miracl.h. They are all accessed via the *mip* — the Miracl Instance Pointer.

`BOOL EXACT` Initialised to `TRUE`. Set to `FALSE` if any rounding takes place during `flash` arithmetic.

`int INPLEN` Length of input string. Must be used when inputting binary data.

`int IOBASE` The "printable" number base to be used for input and output. May be changed at will within a program. Must be greater than or equal to 2 and less than or equal to 256

`int IOBSIZ` Size of I/O buffer.

`BOOL ERCON` Errors by default generate an error message and immediately abort the program. Alternatively by setting `mip->ERCON=TRUE` error control is left to the user.

`int ERNUM` Number of the last error that occurred.

`char IOBUFF[]` Input/Output buffer.

`int NTRY` Number of iterations used in probabalistic primality test by isprime. Initialised to 6.

`int *PRIMES` Pointer to a table of small prime numbers.

`BOOL RPOINT` If set to `TRUE` numbers are output with a radix point. Otherwise they are output as fractions (the default).

`BOOL TRACER` If set to `ON` causes debug information to be printed out, tracing the progress of all subsequent calls to MIRACL routines. Initialised to `OFF`.

Chapter 10

MIRACL Error Messages

MIRACL error messages, diagnosis and response.

10.1 Number base too big for representation

Diagnosis: An attempt has been made to input or output a number using a number base that is too big. For example outputting using a number base of 2^{32} is clearly impossible. For efficiency the largest possible internal number base is used, but numbers in this format should be input/output to a much smaller number base ≤ 256. This error typically arises when using using `innum()` or `otnum()` after `mirsys(.,0)`.

Response: Perform a change of base prior to input/output. For example set the instance variable `IOBASE` to 10, and then use `cinnum()` or `cotnum()`. To avoid the change in number base, an alternatively is to initialise MIRACL using something like `mirsys(400,16)` which uses an internal base of 16. Now Hex I/O can be performed using `innum()` and `otnum()`. Note that this will not impact performance on a 32-bit processor, as 8 Hex digits will be packed into each computer word.

10.2 Division by zero attempted

Diagnosis: Self-explanatory

Response: Don't do it!

10.3 Overflow — Number too big

Diagnosis: A number in a calculation is too big to be stored in its fixed length allocation of memory.

Response: Specify more storage space for all `big` and `flash` variables, by increasing the value of `n` in the initial call to `mirsys(n,b)`.

10.4 Internal Result is Negative

Diagnosis: This is an internal error that should not occur using the high level MIRACL functions. It may be caused by user-induced memory over-runs.

Response: Report to `mike@computing.dcu.ie`[1]

10.5 Input Format Error

Diagnosis: The number being input contains one or more illegal symbols with respect to the current I/O number base. For example this error might occur if `IOBASE` is set to 10, and a Hex number is input.

Response: Re-input the number, and be careful to use only legal symbols. Note that for Hex input only upper-case A–F are permissible.

10.6 Illegal number base

Diagnosis: The number base specified in the call to `mirsys()` is illegal. For example a number base of 1 is not allowed.

Response: Use a different number base.

10.7 Illegal parameter usage

Diagnosis: The parameters used in a function call are not allowed. In certain cases certain parameters must be distinct — for example in `divide()` the first two parameters must refer to distinct `big` variables.

Response: Read the documentation for the function in question.

10.8 Out of space

Diagnosis: An attempt has been made by a MIRACL function to allocate too much heap memory.

Response: Reduce your memory requirements. Try using a smaller value of n in your initial call to `mirsys(n,b)`.

10.9 Even root of a negative number

Diagnosis: An attempt has been made to find, for example, the square root of a negative number.

Response: Don't do it!

[1]Might change the domain of the e-mail address

10.10 Raising integer to negative power

Diagnosis: Self-explanatory.

Response: Don't do it!

10.11 Integer operation attempted on flash number

Diagnosis: Certain functions should only be used with `big` numbers, and do not make sense for `flash` numbers. Note that this error message is often provoked by memory problems, where for example the memory allocated to a `big` variable is accidentally over-written.

Response: Don't do it!

10.12 Flash overflow

Diagnosis: This error is provoked by `flash` overflow or underflow. The result is outside of the representable dynamic range.

Response: Use bigger `flash` numbers. Analyse your progam carefully for numerical instability.

10.13 Numbers too big

Diagnosis: The size of `big` or `flash` numbers requested in your call to `mirsys()` are simply too big. The length of each `big` and `flash` is encoded into a single computer word. If there is insufficient room for this encoding, this error message occurs.

Response: Build a MIRACL library that uses a bigger "underlying type". If not using `flash` arithmetic, build a library without it — this allows much bigger big numbers to be used.

10.14 Log of a non-positive number

Diagnosis: An attempt has been made to calculate the logarithm of a non-positive `flash` number.

Response: Don't do it!

10.15 Flash to double conversion failure

Diagnosis: An attempt to convert a `flash` number to the standard built-in C `double` type has failed, probably because the `flash` number is outside of the dynamic range that can be represented as a `double`.

Response: Don't do it!

10.16 I/O buffer overflow

Diagnosis: An input output operation has failed because the I/O buffer is not big enough.

Response: Allocate a bigger buffer by calling `set_io_buffer_size(.)` after calling `mirsys()`.

10.17 MIRACL not initialised — no call to mirsys()

Diagnosis: Self-explanatory

Response: Don't do it!

10.18 Illegal modulus

Diagnosis: The modulus specified for use internally for Montgomery reduction, is illegal. Note that this modulus must not be even.

Response: Use an odd positive modulus.

10.19 No modulus defined

Diagnosis: No modulus has been specified, yet a function which needs it has been called.

Response: Set a modulus for use internally.

10.20 Exponent too big

Diagnosis: An attempt has been made to perform a calculation using a pre-computed table, for an exponent (or multiplier in the case of elliptic curves) bigger than that catered for by the pre-computed table.

Response: Re-compute the table to allow bigger exponents, or use a smaller exponent.

10.21 Number base must be power of 2

Diagnosis: A small number of functions require that the number base specified in the initial call to `mirsys()` is a power of 2.

Response: Use another function, or specify a power-of-2 as the number base in the initial call to `mirsys()`.

10.22 Specified double-length type isn't

Diagnosis: MIRACL has determined that the double length type specified in mirdef.h is in fact not double length. For example if the underlying type is 32-bits, the double length type should be 64 bits.

Response: Don't do it!

10.23 Specified basis is not irreducible

Diagnosis: The basis specified for GF(2^m) arithmetic is not irreducible.

Response: Don't do it!

Chapter 11

The Hardware/Compiler Interface

Hardware/compiler details are specified to MIRACL in this header file mirdef.h.

For example:

```
/*
 *   MIRACL compiler/hardware definitions - mirdef.h
 *   This version suitable for use with most 32-bit
 *   computers
 *
 *   Copyright (c) 1988-1999 Shamus Software Ltd.
 */

#define MIRACL_32
#define MR_LITTLE_ENDIAN
                     /* this may need to be changed        */
#define mr_utype int /* the underlying type is usually int *
                      * but see mrmuldv.any                */
#define mr_unsign32 unsigned long
                     /* 32 bit unsigned type               */
#define MR_IBITS  32 /* number of bits in an int */
#define MR_LBITS  32 /* number of bits in a long */

#define MR_FLASH 52  /* delete this definition if integer  *
                      * only version of MIRACL required    *
                      * Number of bits per double mantissa */
#define MAXBASE ((mr_small)1<<(MIRACL-1))
#define MRBITSINCHAR 8
                     /* Number of bits in char type        */

/* #define MR_NOASM   * define this if using C code only   *
                      * Note: mr_dltype MUST be defined    */
/* #define mr_dltype long long
                      * double-length type                 */
/*
#define MR_STRIPPED_DOWN * define this to minimize size    *
```

```
                      * of library - all error messages    *
                      * lost! USE WITH CARE - see mrcore.c */
```

This file must be edited if porting to a new hardware environment. Assembly language versions of the time-critical routines in mrmuldv.any may also have to be written, if not already provided, although in most cases the standard C version mrmuldv.ccc can simply be copied to mrmuldv.c.

It is best where possible to use the mirdef.h file that is generated automatically by the interactive config.c program.

Chapter 12

Bibliography

[Blake] BLAKE, SEROUSSI, and SMART. Elliptic Curves in Cryptography, London Mathematical Society Lecture Notes Series 265, Cambridge University Press. ISBN 0 521 65374 6, July 1999

[Brassard] BRASSARD, G. Modern Cryptology. Lecture Notes in Computer Science, Vol. 325. Springer-Verlag 1988.

[Brent76] BRENT, R.P. Fast Multiprecision Evaluation of Elementary Functions. J. ACM, 23, 2 (April 1976), 242–251.

[Brent78] BRENT, R.P. A Fortran Multiprecision Arithmetic Package. ACM Trans. Math. Software 4,1 (March 1978), 57–81.

[Brick] BRICKELL, E, et al, Fast Exponentiation with Precomputation, Proc. Eurocrypt 1992, Springer-Verlag 1993.

[Cherry] CHERRY, L. and MORRIS, R. BC — An Arbitrary Precision Desk-Calculator Language. in ULTRIX-32 Supplementary Documents Vol. 1 General Users. Digital Equipment Corporation 1984.

[Comba] COMBA, P.G. Exponentiation Cryptosystems on the IBM PC. IBM Systems Journal, 29,4 (1990), pp 526–538

[CS] CRAMER, R. and SHOUP, V. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack Proc. Crypto 1998, Springer-Verlag 1999

[DSS] Digital Signature Standard, Communications of the ACM, July 1992, Vol. 35 No. 7

[Gruen] GRUENBERGER, F. Computer Recreations. Scientific American, April 1984

[Jurisic] JURISIC, A and MENEZES A.H. Elliptic Curves and Cryptography, Dr. Dobbs Journal, #264, April 1997

[Knuth73] KNUTH, D.E. The Art of Computer Programming, Vol 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass., 1973.

[Knuth81] KNUTH, D.E. The Art of Computer Programming, Vol 2: Seminumerical Algorithms. Addison-Wesley, Reading, Mass., 1981.

[Korn83] KORNERUP, P. and MATULA, D.W. Finite Precision Rational Arithmetic:

An Arithmetic Unit. IEEE Trans. Comput., C-32, 4 (April 1983), 378–387.

[Korn85] KORNERUP, P. and MATULA, D.W. Finite Precision Lexicographic Continued Fraction Number Systems. Proc. 7th Sym. on Comp. Arithmetic, IEEE Cat. #85CH2146-9, 1985, 207–214.

[LimLee] LIM, C.H. and LEE, P.J. A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup. Advances in Cryptology, Crypto '97, Springer-Verlag 1998

[Marsaglia] MARSAGLIA, G.M. and ZAMAN, A. A New Class of Random Number Generators. The Annals of Applied Probability, Vol. 1, 3, 1991, 462–480

[Matula85] MATULA, D.W. and KORNERUP, P. Finite Precision Rational Arithmetic: Slash Number Systems. IEEE Trans. Comput., C-34, 1 (January 1985), 3–18.

[Maurer] MAURER, U.M. and YACOBI, Y. Non-Interactive Public Key Cryptography. Advances in Cryptography, Eurocrypt '91, Springer Verlag, 1992

[Menezes] MENEZES, A.J. Elliptic Curve Public key Crtyptosystems, Kluwer Academic Publishers, 1993

[HAC] Handbook of Applied Cryptography, CRC Press, 2001

[McCurley] McCURLEY, K.S. A Key Distribution System Equivalent to Factoring. J. Cryptology, Vol. 1. No. 2, 1988

[Monty85] MONTGOMERY, P. Modular Multiplication Without Trial Division. Math. Comput., 44, (April 1985), 519–521

[Monty87] MONTGOMERY, P. Speeding the Pollard and Elliptic Curve Methods. Math. Comput., 48, (January 1987), 243–264

[Morrison] MORRISON, M.A. and BRILLHART, J. A Method of Factoring and the Factorization of F7. Math. Comput., 29, 129 (January 1975), 183–205.

[Pollard71] POLLARD, J.M. Fast Fourier Transform in a Finite Field. Math. Comput., 25, 114 (April 1971), 365–374

[Pollard78] POLLARD, J.M. Monte Carlo Methods for Index Computation (mod p). Math. Comp. Vol. 32, No. 143, pp 918–924, 1978

[Pomerance] POMERANCE, C. The Quadratic Sieve Factoring Algorithm. In Advances in Cryptology, Lecture Notes in Computer Science, Vol. 209, Springer-Verlag, 1985, 169–182

[Reisel] REISEL, H. Prime Numbers and Computer methods for Factorisation. Birkhauser. 1987

[Richter] RICHTER, J. Advanced Windows. Microsoft Press.

[RSA] RIVEST, R., SHAMIR, A. and ADLEMAN, L. A Method for obtaining Digital Signatures and Public-Key Cryptosystems. Comm. ACM, 21,2 (February 1978), 120–126.

[Rubin] RUBIN, P. Personal Communication

[Sch] SCHOOF, R. Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p. Math. Comp. Vol. 44, No. 170. April 1985, pp 483–494

[Scott89a] SCOTT, M.P.J. Fast rounding in multiprecision floating-slash arithmetic. IEEE Transactions on Computers, July 1989, 1049–1052.

[Scott89b] SCOTT, M.P.J. On Using Full Integer Precision in C. Dublin City University Working Paper CA 0589, 1989.

[Scott89c] SCOTT, M.P.J. Factoring Large Integers on Small Computers. National Institute for Higher Education Working Paper CA 0189, 1989

[Scott92] SCOTT, M.P.J. and SHAFAAMRY, M. Implementing an Identity-based Key Exchange algorithm. Available from `ftp://ftp.computing.dcu.ie/pub/crypto/ID-based_key_exchange.ps`

[Scott93] SCOTT, M.P.J. Novel Chaining Methods for Block Ciphers, Dublin City University, School of Computer Applications Working Paper CA-1993

[Scott96] SCOTT, M.P.J. Comparison of methods for modular multiplication on 32-bit Intel 80x86 processors. Available from `ftp://ftp.computing.dcu.ie/pub/crypto/timings.ps`

[Shoup] SHOUP, V. A New Polynomial Factorisation Algorithm and Its Implementation. Jl. Symbolic Computation, 1996

[Stinson] STINSON, D.R. Cryptography, Theory and practice. CRC Press, 1995

[Silverman] SILVERMAN, R.D. The Multiple Polynomial Quadratic Sieve, Math. Comp. 48, 177, (January 1987), 329–339

[Walmsley] WALMSLEY, M., Multi-Threaded Programming in C++. Springer-Verlag 1999.

[WeiDai] DAI, W. Personal Communication