Podręcznik użytkownika M.I.R.A.C.L

Streszczenie

Biblioteka MIRACL składa się z ponad 100 podprogramów, które zajmują się wszystkimi aspektami arytmetyki o dużej dokładności. Są zdefiniowane dwa nowe typy - big dla dużych liczb całkowitych i flash (skrót dla floating -slash) dla dużych liczb wymiernych. podprogramy dla dużych liczb całkowitych są oparte o algorytm Knuth'a, opisany w Rozdziale 4 jego klasycznej pracy 'The Art. Of Computer Programming'. Arytmetyka floating-slash, która pracuje z zaokrąglonymi ułamkami, został zaproponowana przez D.Matula i P.Kornerupa. Wszystkie podprogramy zostały gruntownie zoptymalizowane pod kątem szybkości i wydajności. Jednakże opcjonalna, alternatywna szybkość języka asemblera dla pewnych krytycznych czasowo podprogramów również została zawarta, szczególnie dla popularnych procesorów Intel 80x86. Dostarcza również interfejsu dla C++. Zawarte są pełne kody źródłowe.

1 WPROWADZENIE

Pamiętam kiedy jako młody ,naiwny użytkownik komputera, przyjmowałem stan wiedzy na temat mikrokomputerów; pamiętam swoją niecierpliwość w oczekiwaniu na włączenie komputera, teraz dostępnego w zasięgu ręki; pamiętam wszystkie te artykuły, które obiecywały ,że dzisiejsze mikrokomputery są tak potężne jak wczorajsze mainframe'y". Pamiętam powolne i żmudne wpisywanie pierwszego programu obliczającego 1000! (tj. 1000x999x998....x1) - obliczenie niewyobrażalne ręcznie

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

RUN

Po kilku sekundach pojawiał się wynik:-
Too big at line 30
Pamiętam swoje rozczarowanie
Teraz zbliża się MIRACL. MIRACL jest przenośną biblioteką C, która implementuje typy danych całkowite o dużej dokładności i wymierne, i dostarcza podprogramów do wykonania podstawowej arytmetyki na nich.

Uruchom program fact z dostarczonego dysku i wpisz 1000. To jest twoja odpowiedź - liczba z 2568 cyframi.

Teraz skompiluj i uruchom program roots i poproś go o obliczenie pierwiastka kwadratowego z 2. Praktycznie natychmiast twój komputer poda poprawną wartość do 100 miejsc po przecinku.

Następnie uruchom Public Key Cryptography program enciph. Kiedy poprosi cię o nazwę pliku do szyfrowania naciśnij return. Kiedy poprosi cię o nazwę pliku wyjściowego wpisz FRED, potem wciśnij return. Teraz wpisz jakąś wiadomość, kończąc CONTROL-Z. Twoja wiadomość będzie całkowicie zaszyfrowana w pliku FRED.BLG. teraz uruchom 'deciph' i wpisz FRED. Wciśnij return jako żądanie pliku wyjściowego. Twoja oryginalna wiadomość pojawi się na ekranie.

Ten typ szyfrowania, jest oparty na trudności w rozkładaniu na czynniki pierwsze dużych liczb, oferując dużo większe bezpieczeństwo i elastyczność niż bardziej tradycyjne metody.

Użyteczną demonstracją siły MIRACL jest program ratcalc, potężny kalkulator naukowy - dokładność do 36 miejsc po przecinku i niezwykła zdolność do działania na częściach ułamkowych.

Podręcznik ten zakłada, że czytelnik jest zaznajomiony z językiem C i swoim własnym komputerem. Przy pierwszym czytaniu Rozdziały 4,5 i 6 mogą być spokojnie opuszczone.

Analiza przykładowego kodu źródłowego może być bardzo pomocna.

2.INSTALACJA

Biblioteka MIRACL została pomyślnie zainstalowana na VAX11/780, na różnych stacjach roboczych UNIX (Sun, SPARC, Next, IBM RS/6000) na IBM PC używających kompilatorów Microsft C i C++, Borland Turbo C i Borland C++, kompilatora Watcom C i DJGPP GNU, na komputerach ARM i Apple Macintosh.

Kompletny kod źródłowy dla każdego modułu w bibliotece MIRACL i dla każdego przykładowego programu jest dostarczony na dyskietce. Większość jest napisana w Standardowym ANSI C i powinna zostać skompilowana w jakimś przyzwoitym kompilatorze ANSI C. Niektóre moduły zawierają obszerną ilość linijek w języku asemblera, używanych do optymalizacji wydajności dla pewnych kombinacji kompilator / procesor. Jednak są one wywoływane przez polecenia kompilacji warunkowej i nie są łączone z innymi kompilatorami. Plik wsadowy xxdoit.xxx zawiera polecenia używane dla tworzenia pliku bibliotecznego i przykładowych programów dla kilku kompilatorów. Wydrukuj i zanalizuj właściwy plik dla swojej konfiguracji.

Wcześniej skompilowane biblioteki do bezpośredniego użycia z pewnymi popularnymi kompilatorami znajdują się na dołączonym nośniku: wersje gotowe do uruchomienia tylko pewnych programów przykładowych, dla zaoszczędzenia miejsca.

Do stworzenia biblioteki będziesz potrzebował kompilatora, edytora tekstu, linkera, narzędzi bibliotecznych i asemblera (opcjonalnie). Przeczytaj dokumentację swojego kompilatora po więcej szczegółów. Plik mrmuldv.any, który zawiera specjalne asemblerowe wersje podprogramów krytycznych czasowo muldiv ,muldvd, muldvd2 i mldvm razem z przenośnymi wersjami w C, może być potrzebny dla dopasowania do twojej konfiguracji. Skopiuj go do pliku nazwanego mrmuldv.c i edytuj ten plik. Przeczytaj go dokładnie.

Sprzęt / kompilator określa plik mirdef.h , który musi być wyszczególniony. Pomaga w tym pięć przykładowych wersji nagłówkowych , które są dostępne: mirdef.h16 dla procesorów 16 bitowych, mirdef.h32 dla procesorów 32 bitowych, mirdef.haf jeśli używasz 32 bitowego procesora w 16 bitowym trybie i mirdef.hpc dla pseudo 32 bitowej pracy w 16 bitowym środowisku. Zauważmy, że pełna 32 bitowa wersja jest szybsza, ale tylko możliwa jeśli używamy prawdziwego 32 bitowego kompilatora z 32 bitowym procesorem. Wypróbuj mirdef.gcc stosując gcc i g++ w środowisku Unix (żadnego asemblera).

Pomocą w procesie konfiguracji jest plik config.c., Kiedy skompilujesz go i uruchomisz na docelowym procesorze automatycznie wygeneruje plik mirdef.h i poda ogólne rady co do konfiguracji. Wygeneruje również plik miracl.lst z listą modułów MIRACL będących zawartych w powiązanej ,budowanej bibliotece.

Eksperymentowanie z tym programem jest mocno zalecane. Kiedy kompilujemy ten program NIE UŻYWAMY żadnych optymalizacji kompilatora.

Plik mirdef.h zawiera kilka opcjonalnych definicji: Definiuje MR_NOFULLWIDTH jeśli nie jesteś w stanie dostarczyć wersji muldvd, muldvd2 i muldvm w mrmuldv.c. Definiuje MR_FLASH jeśli życzysz sobie używać zmiennych flash w programach. Musi być zdefiniowany albo MR_LITTLE_ENDIAN lub MR_BIG_ENDIAN . Program config.c automatycznie określa , który jest właściwszy dla procesora.

Poprzez pominięcie definicji MR_FLASH duże zmienne mogą być uczynione dużo większymi a biblioteka tworzona może być dużo mniejsza, prowadząc do bardziej skondensowanych plików wykonywalnych. Zdefiniowany MR_STRIPPED_DOWN pomija komunikaty o błędzie, oszczędzając więcej przestrzeni w tworzonym kodzie. Używaj ostrożnie!

Jeśli nie chcesz żadnego asemblera zdefiniuj MR_NOASM .Wygeneruje to standardowy kod C dla trzech krytycznych czasowo podprogramów i generuje go w rzędzie. Jest to szybsze - oszczędza na obciążeniu funkcji wywołującej - i również daje kompilatorowi optymalizującemu coś do „przeżucia". Zauważ, że jeśli MR_NOASM jest zdefiniowany, wtedy moduł mrmuldv nie jest wymagany w bibliotece MIRACL.

Jeśli używasz Microsft Visual C++, pomocną rzeczą może być przejrzenie pliku msvisual.txt. Jeśli używasz Linuksa sprawdź linux.txt. Użytkownicy kompilatorów Borlanda powinni spojrzeć do borland.txt.

W większości przypadków gdzie gotowe biblioteki lub określone rady z pliku .txt nie są dostępne, poniższa procedura pozwoli z sukcesem zbudować bibliotekę MIRACL:

1. Skompiluj i uruchom config.c na docelowym procesorze
2. Zmień nazwę wygenerowanego pliku z mirdef.tst na mirdef.h
3. Jeśli pomyślnie przeszedłeś program config, wyodrębnij odpowiedni plik mrmuldv.c z mrmuldv.any. Jeśli jest to czysty jeżyk asemblera, może być właściwe nazwać go mrmuldv.s lub mrmuldv.asm.
4. Jeśli została wyselekcjonowana szybka metoda KCM lub Comba do mnożenia modularnego (zobacz poniżej), skompiluj i uruchom mex.c na jakiejś stacji roboczej. Użyj go do automatycznego wygenerowania albo modułu mrcomba.c albo mrkcm.c. Będzie to wymagało procesora /kompilatora określającego plik xxx.mcs. Kompilator musi wspierać bezpośrednie asemblowanie
5. Upewnij się, że wszystkie pliki nagłówkowe MIRACL są dostępne kompilatorowi. Typowo znacznik -I lub /I pozwala tym nagłówkom być dostępnym z bieżącego katalogu.
6. Skompiluj moduły MIRACL wylistowany w wygenerowanym pliku miracl.lst i stwórz plik biblioteczny, zazwyczaj miracl.a lub miracl.lib. Może to być osiągnięte przez wyedytowanie miracl.lst do odpowiedniego pliku wsadowego lub pliku make. W Unix może to być tak proste jak:

```
gcc -Y. -c -02 mr*. C
ar  rc          miracl.a mr*.o
```

7. Jeśli używasz dla MIRACL otoczenia C++, skompiluj wymagane moduły, na przykład monty.cpp i / lub big.cpp itp.
8. Skompiluj i zlinkuj kod swojej aplikacji do jakiegoś modułu C++, wymaganego i do biblioteki MIRACL

Pamiętaj, że MIRACL jest oprogramowaniem przenośnym. Może być przeniesiony na każdy komputer, który wspiera kompilator ANSI C.

Zauważ, że MIRACL jest biblioteką C, nie C++. Powinna zawsze być budowana jako biblioteka C, inaczej mogą pojawić się błędy kompilatora. Zawierając podprogramy MIRACL w programie C, zawieramy plik nagłówkowy miracl.h na początku programu, po zawarciu standardowego pliku nagłówkowego C, stdio.h. Możemy również wywołać podprogramy MIRACL bezpośrednio z programu C++ poprzez:

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

chociaż w większości przypadków będzie bardziej pożądane zastosowanie otoczenia obiektowego C++ opisanego w rozdziale 7

2.1 Optymalizacja

W kontekście MIRACL znaczy to przyspieszenie spraw. Krytyczną decyzją podejmowaną kiedy konfigurujmy MIRACL jest określenie optymalnie odpowiedniego typu do użycia. Zazwyczaj będzie to typ int. Ogólnie rzecz biorąc próbujemy zdefiniować maksimum możliwych odpowiednich typów, jako wymaganych przez config. Jeśli mamy 64 bitowy procesor , powinniśmy móc określić 64 bitowy odpowiedni typ. W niektórych okolicznościach szybsze może być zastosowanie zmienno przecinkowego typu double.

Oczywiście MIRACL w C będzie wolniejszy (ale jeszcze szybki!). Jest również łatwiejszy do rozpoczęcia. Wymaga to typów danych całkowitych dwukrotnie szerszych od odpowiadających typów. W tym kontekście zauważmy, ze dzisiaj większość kompilatorów wspiera typ całkowity long long, który jest dwukrotnie szerszy od int. Czasami jest nazywany _int64 zamiast long long.

Jeśli Twój procesor jest skrajnym rodzajem RISC i nie wspiera całkowitych instrukcji mnożenia / dzielenia, lub jeśli używa bardzo dużych modułów wtedy technika Karatsuba - Montgomery - Comba dla szybkiego modularnego mnożenia może być szybsze dla potęgowania kryptosystemów. Ponownie program config będzie w tym pomocny .

Czasami szybsze jest zaimplementowanie modułu mrmuldv w języku asemblera. Nie wymaga to typów danych o podwójnej szerokości. Jeśli masz szczęście, Twój kompilator będzie wspierał automatyczne wywoływanie asemblacji wbudowanej, która przyspieszy sprawy w przyszłości. Zobacz miracl.h aby zobaczyć jakiekompilatory są obsługiwane w ten sposób.

Dla maksymalnej szybkości użyjemy ekstremalnej techniki zaimplementowanej w mrkcm.c. mrcomba.c. Zobacz kcmcomba.txt po instrukcję jak automatycznie wygenerować te pliki używając dostarczonego narzędzia mex. Zajrzyj do rozdziału 5 po więcej szczegółów.

2.2 Upgrade z wersji 3

Wersja 4.0 wprowadziła Miracl Instance Pointer lub mip. Poprzednia wersja używała kilku globalnych i statycznych zmiennych do przechowywania informacji o stanie wewnętrznym. Związane są z tym dwa problemy. Po pierwsze zmienne globalne muszą przybierać różne nazwy aby uniknąć „zderzenia" z innymi nazwami globalnymi. Po drugie czyni dużo trudniejszym rozwijanie aplikacji wielowątkowych . Wraz z wersją 4.0 wszystkie takie zmienne ,teraz odnosząc się do instancji zmiennej, są członkami struktury typu miracl i muszą być dostępne przez wskaźnik do instancji tej struktury. Ten globalny wskaźnik jest teraz tylko statyczną / globalną zmienną biblioteki MIRACL . Wartości te są zwracane przez podprogram mirsys, który inicjalizuje bibliotekę MIRACL

Programiści C++ powinni odnotować, zmiany w nazwie instancji klasy z miracl na Miracl. Mip może być znaleziony przez adres tej instancji

Miracl precision = 50;

-

mip = &precision

-

etc

2.3 Programowanie wielowątkowe

W wersji 4.4 MIRACL oferuje pełne wsparcie dla programowania wielowątkowego. Problem jaki musi być przezwyciężony jest taki, że MIRACL musi mieć dostęp do wielu instancji określonego stanu informacji poprzez swój mip. Idealnie byłoby gdyby nie było zmiennych globalnych, ale MIRACL ma jeden taki wskaźnik. Niestety każdy wątek jakiego używa MIRACL potrzebuje swojego mip'a, wskazujący jego własny niezależny stan. Jest to dobrze znany problem powiązany z wątkami.

Pierwszym rozwiązaniem jest modyfikacja MIRACL, aby mip, zamiast być przekazywanym globalnie, był przekazywany jako parametr do każdej funkcji MIRACL. Podprogramy MIRACL mogą być automatycznie modyfikowane do wspierania tego przez zdefiniowanie MR_GENERIC_MT w mirdef.h. Teraz (prawie wszystkie) podprogramy MIRACL są tak zmieniane aby mip był pierwszym parametrem każdej funkcji Niektóre proste funkcje są wyjątkami i nie wymagają parametru mip - są one oznaczone gwiazdkami w rozdziale 9. Jako przykład programu zmodyfikowanego do pracy z biblioteką MIRACL zbudowaną w ten

sposób zobacz program brent_mt.c. Zauważ jednak, że to rozwiązanie nie ma zastosowania do programów napisanych przy użyciu interfejsu obiektowego MIRACL C++ opisanego w rozdziale 7. Stosuje się to do programów C, mających dostęp bezpośredni do podprogramów MIRACL.

Alternatywnym rozwiązaniem jest zastosowanie Kluczy, które są typem określonych , wątkowych „globalnych" zmiennych. Klucze nie są częścią standardowego C/C++, ale specyficznymi rozszerzeniami systemu operacyjnego implementowanymi poprzez specjalne funkcje wywołujące. MIRACL dostarcza wsparcia dla systemu operacyjnego Microsft Windows i UNIX. Dawniej Klucze były nazywane Thread-Local Storage. Dla UNIX'a MIRACL wspiera standardowy interfejs POSIX dla wielowątkowości . Bardzo użytecznym odnośnikiem do Windowsa i UNIX'a jest [Walmsley], To wsparcie dla wątków jest zaimplementowane w module mrcore.c, przy starcie pliku i w podprogramie inicjalizującym mirsys

Dla Windows definiujemy MR_WINDOWS_MT w mirdef.h a dla UNIX'a definiujemy MR_UNIX_MT. Są tego jakieś implikacje programistyczne.

Na początku Klucz, który utrzymuje mip musi być zainicjalizowany i ostatecznie zniszczony przez podstawowy wątek programu. Funkcje te są wykonywane przez wywołanie ,odpowiednio, specjalnych podprogramów mr_init_threading i mr_end_threading.

W programach C++ funkcje te mogą być powiązane z konstruktorami i destruktorami zmiennej globalnej - zapewni to ,że będą wywoływane we właściwym czasie przed tym nim nowy wątek nie wyłoni się z wątku głównego. Muszą być wywoływane przed wywołaniami wątku mirsys albo jawnie albo niejawnie przez stworzenie instancji określonego wątku klasy Miracl.

Jest silnie zalecane aby program w trakcie rozwoju wykonywany był bez wsparcia dla wątków. Tylko kiedy program jest w pełni przetestowany i zdebuggowany powinien być skonwertowany z wątkami.

Programy wątkowe mogą wymagać innych cech specyficznych OS'a, pod względem linkowania specjalnych bibliotek, lub dostępu do specjalnych podprogramów sterty. Pod tym względem warto jest wskazać , że cała sterta MIRACL dostępna jest poprzez moduł mralloc.c

Zobaczmy program przykładowy threadwn.cpp dla przykładu wielowątkowości Windows w C++. Przeczytaj komentarze w tym programie - może być skompilowany i uruchomiony z linii poleceń Windows. Podobnie zobacz threadux.cpp dla przykładu wielowątkowości UNIX'a

3 INTERFEJS UŻYTKOWNIKA

PRZYKŁAD

```
/*
•   Program do obliczania silni .
*/

#include <stdio.h>
#include „miracl.h"                              /* dołączenie systemu MIRACL */

void main ()
{ /* obliczenie silni liczby */
    big nf;                                      /* deklaracja „dużej" zmiennej nf */
    int  n;
    miracl   *mip = mirsys(5000, 10);
/* podstawa 10, 5000 cyfr dla big */
    nf = mirvar (1)                              /* inicjalizacja zmiennej big nf = 1 */
    printf(„program silnia\n");
    printf(„liczba  wejściowa  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);                   /* wynik wyjściowy */
}
```

Program ten może być użyty do szybkiego obliczania i wydrukowania 1000! (2568 cyfrowa liczba) w mniej niż sekundę na komputerze 60MHz'owym ,zadanie wykonywane przez H.S.Uhlera używając kalkulatora i mając dużo cierpliwości zajęło kilka lat' [Knuth73]. Wiele innych przykładowych programów jest opisanych w Rozdziale 8.

Każdy program, który chciałby używać systemu MIRACL musi mieć instrukcję #include „miracl.h". Mówi ona kompilatorowi aby wprowadził plik nagłówkowy C miracl.h do pliku źródłowego programu głównego przed dokonaniem kompilacji. Plik ten zawiera deklaracje wszystkich podprogramów MIRACL dostępnych użytkownikowi. Mały pod - plik nagłówkowy mirdef.h zawiera określone detale sprzęt / kompilator.

W programie głównym system MIRACL musi być zainicjalizowany przez wywołanie podprogramu mirsys, który ustawia podstawę liczbową i maksymalny rozmiar zmiennych big i flash. Inicjalizuje również losowy system liczbowy i tworzy kilka przestrzeni roboczych zmiennych big dla ich własnego użytku Wartością zwracaną jest Miracl Instant Pointer lub mip. Ten wskaźnik może być zastosowany dla uzyskania dostępu do różnych wewnętrznych parametrów powiązanych z bieżącą instancją MIRACL'a. Na przykład ustawienie flagi ERCON można zapisać

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

Początkowe wywołanie mirsys również inicjalizuje system śledzenia błędów, który jest zintegrowany z pakietem MIRACL. Gdy tylko jest wykryty błąd sekwencja podprogramu ściąga podprogram który raportuje generowany błąd tak jak i sam błąd. Typowy komunikat o błędzie może być taki

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

Taki raport błędu ułatwia debuggowanie i pomaga nam podczas rozwijania tego podprogramu. Skojarzona instancja zmiennej TRACE, zainicjalizowana OFF, jest ustawiona przez użytkownika przez ON, spowoduje śledzenie postępu programu przez podprogramy MIRACL wyprowadzane na ekran komputera.

Instancja flagi ERNUM, zainicjalizowana zerem, zapisuje liczbę ostatnich występujących wewnętrznych błędów MIRACL. Jeśli flaga ERCON jest ustawiona na FALSE (domyślnie) komunikat o błędzie jest kierowany do stdout a program przerywany przez wywołanie systemowego podprogramu exit(0). Jeśli twój system nie dostarcza takiego podprogramu, programista musi dostarczyć coś w zamian. Jeśli ERCON jest ustawiony na TRUE żaden komunikat o błędzie nie będzie wyemitowany, zamiast tego cały ciężar spada na programistę ,na wykrycie i obsługę błędu. W tym przypadku jest kontynuowane wykonywanie. Programista musi wybrać pomiędzy zajęciem się błędem a resetem ERNUM do zera. Jednak błędy są zazwyczaj fatalne, i jeśli ERNUM nie jest zerem wszystkie podprogramy MIRACL wywołują dalsze „błędy' i wychodzą natychmiast. Spojrzyj do miracl.h po listę wszystkich możliwych błędów.

Każda zmienna big i flash w programie użytkowym musi być inicjalizowana przez wywoływanie podprogramu mirvar, który również pozwala zmiennej być początkową małą wartością całkowitą.

Pełny zbiór podprogramów arytmetycznych i liczbowych zadeklarowany w miracl.h może być zastosowany z tymi zmiennymi. Dozwolona jest pełna elastyczność (prawie zawsze) w parametrach używanych z tymi podprogramami. Na przykład wywołanie multiply (x,y,z) mnoży zmienną big x przez zmienną big y przechowując wynik w zmiennej big z. Równoważne może być multiply(x,y,x), multiply(y,y,x) lub multiply(x,x,x). Ten ostatni po prostu daje Ci kwadrat z x. Zauważ ,że pierwsze parametry są przez konwencję zawsze (zazwyczaj) wstawiane do podprogramów. Podprogramy te nie tylko pozwalają na liczby big i flash ale również pozwala tym zmiennym wykonywać arytmetykę z wbudowanymi całkowitymi i typami danych o podwójnej precyzji.

Dostarczone podprogramy konwersji konwertują z jednego typu na inny. Po szczegóły każdego podprogramu zajrzyj do odnośnej dokumentacji w Rozdziale 9 a po przykłady programów do Rozdziału 8.

Wejście i wyjście do pliku lub urządzenia I/O jest obsługiwany przez podprogramy innum, otnum, cinnum i cotnum. Pierwsze dwa używają bazowej stałej liczby określonej przez użytkownika w pierwszym wywołaniu mirsys. Druga para działa wraz z instancją zmiennej IOBASE, która może być dołączana dynamicznie przez użytkownika Prosta zasada jest taka, że jeśli program jest ograniczony CPU lub wymaga zmiany bazy, wtedy ustawiamy bazę początkową MAXBASE (lub 0 jeśli jest możliwa baza o pełnej szerokości - zobacz Rozdział 4)

i używamy cinnum i cotnum. Jeśli, z drugiej strony, program jest ograniczony przez I/O, lub potrzebujemy dojścia do pojedynczej cyfry liczby (używając getdig, putdig i numdig), używamy innum i otnum.

Wejście i wyjście do / z ciągów znaków jest również wsparte w podobny sposób przez podprogramy instr, otstr, cinstr i cotstr . Podprogramy wejściowe mogą być używane do ustawiania liczb big lub flash dużymi stałymi wartościami. Przez wyjście do ciągu, formatowanie może mieć miejsce przed rzeczywistym wyprowadzeniem do pliku lub urządzenia I/O.

Liczby bazowe do 256 mogą być przedstawiane. Liczby o podstawie 60 używają tak wiele symboli 0-9,A-Z, a-x jak to konieczne. Podstawa liczbowa 64 wymusza standard kodowania o podstawie 64. Na wyjściu liczby o podstawie 64 są uzupełniane symbolem = jeśli jest to konieczne, ale nie formatowania. Na wejściu białe znaki są pomijane, a uzupełnienie ignorowane. Nie używa się podstawy 64 z liczbami flash. Nie używa się podstawy 64 dla wyjściowych liczb ujemnych ponieważ znak jest ignorowany.

Jeśli podstawa jest większa niż 60 (ale nie 64), używane symbole to kody ASCII od 0 - 255.

Podstawa 256 jest użyteczna kiedy konieczna jest interpretacja lini tekstu jako dużej liczby całkowitej, tak jak jest w przypadku programu Kryptografii Klucza Publicznego opisanego w Rozdziale 8 Podprogram big_to_byte i byte_to_big pozwala bezpośrednio konwertować z wewnętrznego formatu big do/ z czystej postaci binarnej. Ciągi są zazwyczaj zakończone zerem. Jednak problem powstaje kiedy używamy podstawy 256. W tym przypadku każda cyfra od 0 - 255 może wystąpić w liczbie. Więc 0 nie musi koniecznie wskazywać końca ciągu. Na wejściu inna metoda musi być użyta do wskazania liczby cyfr w ciągu.

Przez ustawienie instancji zmiennej INPLEN = 25 (na przykład), przed wywołaniem innum lub instr, wprowadzanie jest zakończone po 25 bajtach. INPLEN jest inicjalizowany 0 i resetowany do zera przez odnośny podprogram przed powrotem.

Na przykład, inicjalizujemy MIRACL używając 400 bajtowych bigs

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

Wewnętrzne obliczenia są bardzo wydajne przy użyciu tej podstawy.

Wprowadzamy ciąg ASCII jako podstawową 256 liczbę. To będzie ciąg zakończony zerem, więc nie trzeba INPLEN

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

Teraz jest wymagane wprowadzanie dokładnie 1024 losowych bitów

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

Ale chcemy uzyskać dane wyjściowe w HEX

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

Teraz w podstawie 64

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

Liczby wymierne mogą być wprowadzane przy użyciu albo kropki pozycyjnej (np. 0.3333) albo ułamka (np. 1/3) Obie formy mogą być używane na w wyjściu przez ustawienie instancji zmiennej RPOINT = ON lub =OFF

4. WEWNĘTRZNA REPREZENTACJA

Konwencjonalna arytmetyka komputerowa dostarczana przez większość kompilatorów języków komputerowych dostarcza jednego lub dwóch typów danych zmienno przecinkowych (tj. o pojedynczej i podwójne precyzji) do przedstawiania wszystkich liczb rzeczywistych, razem z jednym lub więcej typami całkowitymi dla przedstawiania wszystkich liczb. Te wbudowane typy danych są blisko powiązane z odnośną architekturą komputerową, która jest stworzona do szybkiego działania z dużą ilością małych liczb zamiast wolnego działania z małą ilością dużych liczb (danymi jako stałe alokacje pamięci). Zmienno przecinkowość pozwala na stosunkowo małe liczby binarne (np. 32 bitowe) do przedstawiania liczb rzeczywistych z wystarczającą precyzją (np. 7 miejsc po przecinku) w dużym , dynamicznym zakresie. Typy całkowite

pozwalają na przedstawiania małych liczb całkowitych bezpośrednio w ich odpowiedniku binarnym, lub postaci uzupełnienia do dwóch jeśli są ujemne. Niemniej jednak to konwencjonalne podejście do arytmetyki komputerowej ma kilka wad.

- Typy danych całkowite i zmienno przecinkowe są niekompatybilne. Zauważmy, że zbiór całkowity, chociaż nieskończony, jest podzbiorem liczb wymiernych (tj. ułamkowych), który okazuje się podzbiorem liczb rzeczywistych. A zatem każda liczba całkowita ma swoją odpowiednią reprezentację zmienno przecinkową. Niestety te dwie reprezentacje generalnie będą różne. Na przykład małe , dodatnie liczby całkowite będą przedstawiane przez ich binarne odpowiedniki jako całkowite, a przez oddzielną mantysę i wykładnik jako zmienno przecinkowe. Sugeruje to potrzebę podprogramów konwersji do konwersji z jeden j postaci do drugiej.
- Większość liczb wymiernych nie może być wyrażonych dokładnie (np. 1/3).Zamiast tego, system zmienno przecinkowy może wyrazić dokładnie tylko te liczby wymierne, których mianowniki są wielokrotnością współczynnika odpowiedniej podstawy. Na przykład nasz dobrze znany system dziesiętny może wyrażać dokładnie tylko te liczby wymierne, których mianowniki są wielokrotnościami 2 i 5; 1/20 to dokładnie 0.05, 1/21 to 0.0476190476190...
- Zaokrąglanie w systemie zmienno przecinkowym jest uzależnione od podstawy i źródłem niejasnych błędów
- Fakt, że rozmiar typów danych całkowitych i zmienno przecinkowych jest narzucony przez architekturę komputera, udaremnia wysiłek twórców języka uczynienia go naprawdę przenośnym
- Liczby mogą być tylko przedstawiane w stałej uzależnionej od maszyny precyzji. W wielu aplikacjach może to być wadą wyniszczającą, na przykład w nowej , rozwijającej się kryptografii klucza publicznego.
- Zjawisko uzależnienia od bazy nie jest łatwe do zrozumienia. Na przykład bezie trudno uzyskać dostęp do poszczególnych cyfr liczby dziesiętnej, przedstawionej w tradycyjnym całkowitym typie danych.

Tutaj jest opisany zbiór standardowych podprogramów C, które manipulują bezpośrednio liczbami wymiernymi o dużej dokładności, z liczbami całkowitymi o dużej dokładności jako kompatybilnym podzbiorem. Może być również wykona przybliżona arytmetyka liczb rzeczywistych.

Dwa nowe typy danych to big i flash. Ten pierwszy jest używany do przechowywania liczb całkowitych o dużej dokładności, a drugi przechowuje liczby ułamkowe jako licznik i mianownik w formacie „floating- slash”. Oba przybierają formę tablicy cyfr o stałej długości, ze znakiem i informacją o długości zakodowaną w oddzielnej 32 bitowej liczbie całkowitej. Typ danej zdefiniowanej jako mr_small będzie używał do przechowywania cyfr liczby jednego z typów wbudowanych, na przykład int, long lub nawet double. Jest to odniesienie do „odpowiedniego typu”.

Oba nowe typy mogą być wprowadzone do składni języka C poprzez instrukcje C

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

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

Teraz zmienne big i flash mogą być zadeklarowane podobnie jak każdy wbudowany typ danych np. big

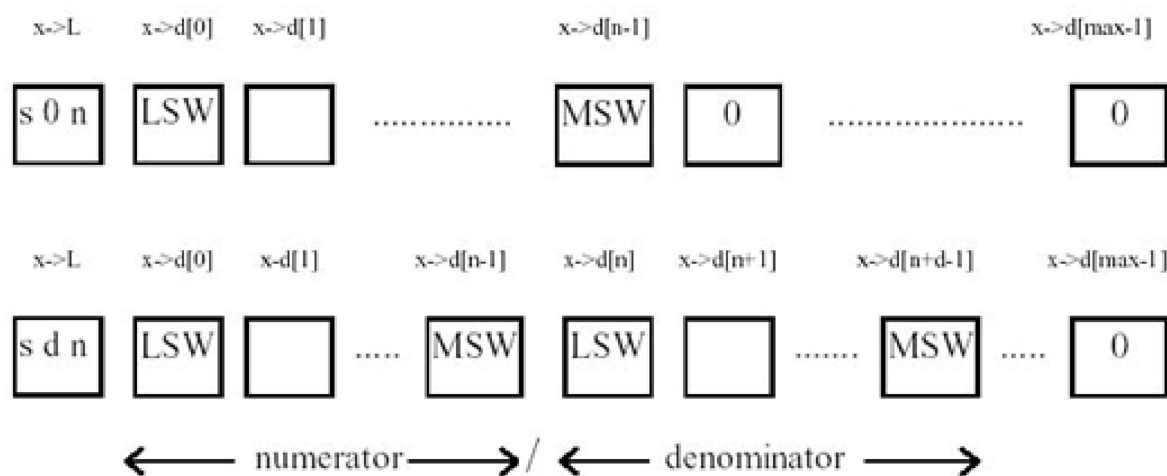
x, y [10], z [10][10];

Zauważ, że big jest wskaźnikiem. Pamięć potrzebna dla każdej instancji liczby big lub flash jest pobierana ze sterty . Dlatego też każda liczba big lub flash musi być zainicjalizowana przed użyciem, i wymaga pamięci do niej przydzielonej.

Zauważ, że użytkownik tych typów danych nie zajmuje się tą wewnętrzną reprezentacją.; podprogramy biblioteczne pozwalają na bezpośrednie manipulowanie liczbami big i flash.

Struktura big i flash jest pokazana na rysunku (4.1)

Struktura ta łączy łatwość stosowania z przedstawioną wydajnością. Mianownik o długości zero (d=0), zakłada rzeczywisty mianownik jeden ; podobnie licznik o długości zero (n=0) zakłada licznik jeden. Samo zero jest zdefiniowane wyjątkowo jako liczba , której pierwszym elementem jest zero (tj. d=0 i n = 0).



Rysunek 4.1: Struktura typów danych big i flash, gdzie s jest znakiem liczby, n i d są długościami ,odpowiednio licznika i mianownika, a LSW i MSW znaczą odpowiednio „Najmniej znaczące słowo" i „Bardziej znaczące słowo.

Zauważ, że ukośnik w danej typu flash nie ma stałej pozycji i może „pływać" w zależności od rzeczywistego rozmiaru licznika i mianownika.

Liczba flash jest określana przez podział jej na oddzielne składowe big licznik i big mianownik. Liczba big jest określana przez wydzielenie i działanie na każdym z jej całkowitym elemencie składowym .Unikając możliwego przepełnienia, liczby w każdym elemencie są zazwyczaj ograniczane do pewnego mniejszego zakresu niż do pełnej długości słowa ,tzn. 0 do 32767 (=2^{15} - 1) na 16 bitowym komputerze. Jednak przy ostrożnym programowaniu może być również użyta pełna rozpiętość podstawy 2^{16} , ponieważ język C nie raportuje błędów czasu wykonania przy przepełnieniu całkowitym.

Kiedy system jest inicjalizowany użytkownik określa stałą liczbę słów (lub bajtów) przypisanych do wszystkich zmiennych big lub flash. Każda baza może być użyta do maksimum, które zależy od długości słowa używanego komputera .Jeśli życzysz sobie użyć podstawę b, system, dla optymalnej wydajności, będzie używał podstawy b^n, gdzie n jest największą liczbą całkowitą, taką, że b^n mieści się w pojedynczym słowie komputerowym. Programy, generalnie, będą się wykonywały szybciej jeśli jest używana barza o pełnej szerokości (osiągana przez wyszczególnienie bazy 0 w wywołaniu inicjalizującym mirsys). Zauważ , że ten tryb może być wsparty przez obszerny wbudowany język asemblera dla pewnych powszechnych połączeń kompilator / procesor, w pewnych krytycznych czasowo podprogramach, na przykład jeśli używamy Borland / Turbo C z procesorem 80x86. Przeanalizuj na przykład kod źródłowy w module mrarth1.c.

Zakodowanie informacji o znaku , rozmiarze licznika i mianownika w pojedynczym słowie jest możliwe , ponieważ język C ma standardową budowę dla manipulacji bitami.

5. IMPLEMENTACJA

Nie będzie oryginalnym stwierdzeniem, ze podprogramy używają implementacji arytmetycznej w danej typu big. Używane algorytmy są wiernie oddane w opisie Knutha. Jednakże została uczyniona próba optymalizacji implementacji pod katem szybkości. Podczas czasochłonnych podprogramów mnożenia i dzielenia, zazwyczaj , musimy mnożyć razem cyfrę każdego operandu, dodać „przeniesienie" z poprzedniej operacji a potem oddzielić cyfry wyniku i „przenieść" do następnej operacji. Dla ilustracji rozważmy mnożenie dziesiętne:

8723536221
x 9
78511825989

Poprawnie przetwarzamy kolumnę z 5, mnożymy 5x9 =45,dokładamy „przeniesienie" z poprzedniej kolumny (3), co daje 48, wpisujemy 8 jako wynik do tej kolumny i przenosimy 4 do następnej kolumny.

Ta podstawowa, elementarna operacja jest zasadniczo obliczeniem ilorazu (a.b+c) / m i jego reszty. Dla powyższego przykładu a= 5, b = 9, c =3 i m. = 10. Ta operacja ma zdecydowanie uniwersalne zastosowanie, a ponieważ znajduje się w pętli wewnętrznej algorytmu arytmetycznego, jego wydajna implementacja jest niezbędna.

Są trzy główne trudności z implementacją operacji MAD (Multiply, Add i Divide) w języku wysokiego poziomu

- Będzie wolna
- Iloraz i reszta nie są dostępne równocześnie jako wynik operacji dzielenia. Dlatego też obliczenie musi by wykonane zasadniczo dwa razy, raz dla pobrania ilorazu i drugi dla reszty
- Chociaż wynik operacji mieści się w dwóch pojedynczych cyfrach, pośredni wynik (a.b+c) może być podwójnej długości. Istotnie taki podprogram Multiply - Add and Divide może być użyty przy okazji, kiedy może być wymagana podwójna wielkość przez podstawowy algorytm arytmetyczny. Zauważ, że język C jest obdarzony typem danej całkowitej 'long', który może faktycznie być zdolnym do czasowego przechowywania takiego wyniku.

Z tych powodów najlepiej zaimplementować te kluczowe operacje w języku asemblera używanego komputera, chociaż jest możliwa przenośna wersja w C. Na poziomie kodu maszynowego chwilowy wynik o podwójnej długości może być często stosowany, nawet jeśli typ danej long C sam nie jest podwójnej długości (ponieważ jest to przypadek dla większości kompilatorów C zaimplementowanych na 32 bitowych komputerach, dlatego ints i longs , oba, są wielkościami 32 bitowymi). Po więcej szczegółów zajrzyj do dokumentacji w pliku mrmuldv.any

Krytyka systemu MIRACL może być taka, że używa tablic o stałej długości z danymi typu big i flash Jest to zrobione po to aby uniknąć trudności i czasochłonnych problemów alokacji pamięci i odzyskiwania pamięci, które wystąpiły by przy zmiennej długości. Jednakże, znaczy to ,że kiedy robimy obliczenia na całkowitym big, wtedy wynik wszystkich pośrednich kalkulacji musi być mniejszy niż lub równy stałemu rozmiarowi początkowo określonemu w mirsys.

Praktycznie większość liczb w stałym obliczeniu całkowitym jest mniej więcej tego samego rozmiaru, z wyjątkiem kiedy dwie są mnożone razem w przypadku tworzenia pośredniego iloczynu o podwójnej długości. Jest to zazwyczaj ponownie redukowane przez późniejszą operację dzielenia. Klasycznym tego przykładem może być program rozkładania na czynniki pierwsze Pollarda - Brenta (Rozdział 8).

Zauważ, że jest inny objaw, na poziomie makra, problemu wspomnianego powyżej .Szkoda byłoby określać każdą zmienna dwa razy większą niż to konieczne, kopiując te sporadyczne pośrednie iloczyny. Z tego powodu specjalny podprogram Multiply, Add and Divide, mad został wprowadzony w bibliotece MIRACL Okaże się ona użyteczna kiedy będziemy implementować duże programy (takie jak program rozkładania na czynniki pierwsze Pomerance'a - Silverman'a - Montgomery'ego, Rozdział 8) na komputerach z ograniczoną pamięcią.

Tak jak podstawowe operacje arytmetyczne , również podprogramy uwzględniają:

1. generowanie i testowanie liczb pierwszych big, przy użyciu probabilistycznego testu pierwszości
2. generowanie liczb losowych big i flash, w oparciu generatora odejmowania z pożyczką. Zanotuj jednak, że podstawowy generator liczb losowych zaimplementowany wewnętrznie nie jest chroniony kryptograficznie. W rzeczywistej aplikacji kryptograficznej nie będzie to wystarczające. Silny kryptograficznie generator jest dostarczony w module mrstrong.c
3. obliczają potęgę i pierwiastki
4. implementują normalny i rozszerzony algorytm Euclidean GCD
5. implementują „Chińskie Twierdzenie o Resztach" i obliczają Symbol Jacobi'ego
6. mnożą skrajnie duże liczby używając metody szybkiej transformacji Fourier;a

Kiedy wykonujemy arytmetykę modularną, operacją krytyczną czasową jest 'Mnożenie Modularne' to znaczy mnożenie dwóch liczb po których następuje redukcja reszty kiedy dzielimy przez stały n współczynnik. Jedynym oczywistym byłoby użycie podprogramu mad opisany powyżej. Jednak Montgomery zaproponował metodę alternatywą. Wymaga ona tego aby liczba została najpierw skonwertowana do specjalnej postaci n-resztkowej. Jakkolwiek w tej postaci mnożenie modularne jest nieco szybsze, używamy specjalnego podprogramu, który nie wymaga jakiegokolwiek dzielenia. Kiedy obliczenie jest skończone, odpowiedź może być skonwertowana z powrotem do normalnej postaci. Zauważ, że modularne dodawanie i odejmowanie n-resztkowe jest kontynuowane, jak zwykle, przy użyciu takich samych podprogramów jakie są używane dla normalnej arytmetyki. Przy danych wymaganiach dla konwersji zmiennych do/ z formatu n-resztkowego, metoda Montgomery'ego powinna być rozpatrywana tylko, kiedy obliczenia wymagają obszernej ilości arytmetyki modularnej używającej takich samych modułów. Faktycznie jest to dużo bardziej dogodne do użycia w środowisku C++, które ukrywa takie trudne szczegóły. Zobacz Rozdział 7.

Arytmetyk Montgomery'ego jest używana wewnętrznie przez wiele z podprogramów biblioteki MIRACL, które wymagają rozległej arytmetyki modularnej, takie jak wysoce zoptymalizowana modularna funkcja

potęgowania powmod i te funkcje, które implementują arytmetykę Krzywej Eliptycznej GF(p). Szczegóły w rozdziale 9.2.

Dla możliwie najszybszej arytmetyki modularnej, musimy niestety uciec się do języka asemblera, i metod optymalizacji dla poszczególnych modułów lub modułów o określonych rozmiarach. Jest wiele różnych technik wspierających to i mogących być użytymi. Pierwsze dwie metody to Comba i KCM, zaimplementowane odpowiednio w plikach mrcomba,c i mrkcm.c. Pliki te są stworzone z pliku szablonu mrcomba.tpl i mrkcm.tpl przez wprowadzenie makr zdefiniowanych w pliku .mcs. Jest to robione automatycznie przez stosowanie dostarczonego narzędzia mex (macro expansion). Kompilacja i uruchomienie config.c w systemie docelowym automatycznie stworzy odpowiedni mirdef.h i doradzi jak kontynuować. Również odczyta kcmcomba.txt. Aby uzyskać możliwie najszybszą wydajność dla Twojej osadzonej aplikacji jest zalecane abyś rozwinął swój własny plik x.mcs, jeśli już nie jest dostarczony do Twojego procesora/ kompilatora

Dwie inne , raczej eksperymentalne techniki są zaimplementowane w plikach mr87v.c i mr87f.c , tylko dla procesorów Intela z rodziny 80x86, używając kompilatora Borland C++.

Jeśli warunki są poprawne, właściwy kod będzie automatycznie wywoływany przez wywołanie na przykład powmod.

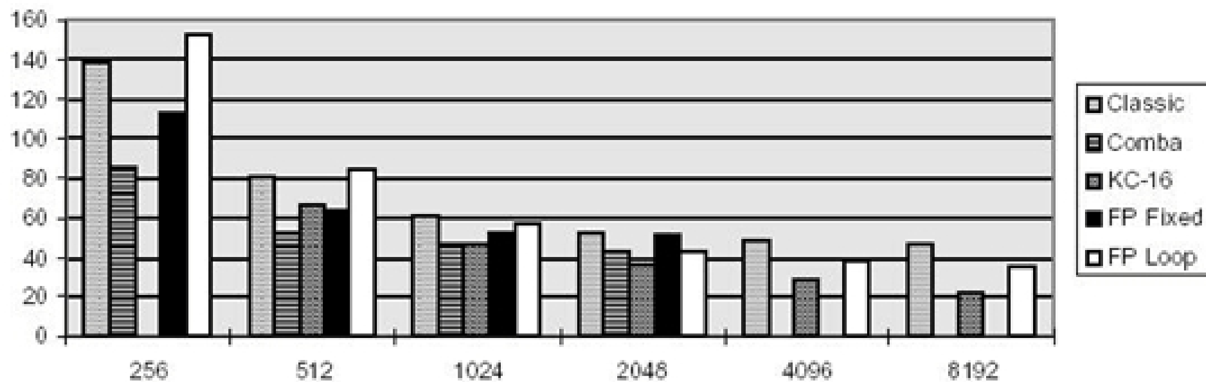
Ważne jest do zapamiętania, że te cztery opisane techniki wymagają kompilatora , który wspiera asemblację wbudowaną. Co więcej te ostatnie dwie techniki mogą być testowane tylko z kompilatorem Borland C++ v 4.5 z rodziną procesorów 80x86.

Pierwszym pomysłem jest całkowite rozwinięcie i reorganizacja ukrytych pętli programu w mnożeniu i redukcja przetwarzania, jako pierwsze zalecane przez [Comba] i modyfikowany przez [Scott96]. Zobacz mrcomba.tpl. Moduł o stałej długości musi być użyty i określony w czasie kompilacji przez zdefiniowanie rozmiaru modułu MR_COMBA (w słowach) w mirdef.h. Działa to dobrze dla małych i średnich rozmiarów modułów, jakich używa się w kryptografii krzywej eliptycznej GF(p). Dla uzyskania większej szybkości, algorytm redukcji modularnej może być zoptymalizowany do modułu, który ma szczególnie prostą postać. Może to zostać wykonane przez ręczne wprowadzenie właściwego kodu do mrcomba.tpl. Przykład kodu dla przypadku współczynnika $p = 2^{192} - 2^{64} -1$ jest podany w podprogramie comba_redc. Wywołanie tego specjalnego kodu MR_SPECIAL musi być zdefiniowane w mirdef.h.

Technika ta może być połączona z pomysłem Karatsubty dla szybkiego mnożenia przyspieszając mnożenie modularne dla dużych modułów. Metoda Karatsuba-Comba-Mobntgomery (KCM) jest wywoływana przez zdefiniowany MR_KCM w mirdef.h Rozmiar modułu w słowie komputerowym jest ograniczony do równania MR_KCM $*2^n$ dla każdego dodatniego n (w granicach zdrowego rozsądku). Jest to konsekwencja stosowania algorytmu Karatsuby. Na przykład definiując MR_KCM na 8 na 32 bitowym komputerze pozwalamy na powszechne rozmiary modułów 512, 1024, 2048.....bitów.

Inną alternatywą jest wykorzystanie koprocesora zmienno przecinkowego (jeśli jest), ponieważ jego instrukcje mnożenia są często szybsze niż te z jednostki całkowitej. Jest to przypadek dla oryginalnego procesora Pentium, którego koprocesor pobiera tylko 3 cykle do wykonania mnożenia w porównaniu z 10 wymaganymi dla mnożenia całkowitego, chociaż nie jest to prawda dla Pentium Pro, II lub III. Koprocesor ma również osiem dodatkowych rejestrów i może manipulować bezpośrednio 64 bitowymi liczbami. Cechy te pozwalają programiście na dodatkową elastyczność, która może być uznana za zaletę. Niektóre eksperymentalne kody mogą być napisane w module mr87f.c i mr87v.c, które mogą być wykorzystane przez zdefiniowanie MR_PENTIUM w mirdef.h. Używając config.c do wygenerowania mirdef.h - tym razem jako odpowiedni musi być wybrany double. Moduł mr87v.c małych rozmiarów kod zapętlony, który będzie działał z każdym modułem mniejszym niż pewno maksimum. Moduł mr87f.c rozwija pętle dla większej szybkości, ale jest to nieporęczne i wymagające modułów o stałych rozmiarach. Zauważ ,że te tryby działania są niekompatybilne z podstawą o pełnej szerokości i działają najlepiej z podstawą liczby(zazwyczaj) 2^{28} lub 2^{29} - config.c będzie rozpracowywał to dla ciebie. Zauważ również, że chociaż ta metoda przyspieszy modularne potęgowanie na Pentium, może w rzeczywistości być wolniejsza dla większości innych procesorów 80x86, więc należy używać jej ostrożnie. W jednym teście 2048 bitowa liczba została podniesiona do 2048 bitowej potęgi, mod 2048 bitowe moduły. Zajęło to 2.4 sekundy na Pentium 60 MHz.

Fig 3: Modular Exponentiation - Pentium Pro200



Ten diagram ilustruje względne czasy wymagane przez każdą metodę na procesorze Pentium Pro 200 MHz, kiedy kompilujemy 32 bitowym kompilatorem Borland C. Metoda w lini podstawowej „Classic" odnosi się do kodu asemblerowego, zaimplementowanego bezpośrednio w mrarth2.c i mrmonty.c. Implementacje Comba i KCM używają asemblera z pliku ms86.mcs. Rozmiary modułów są na osi x, a czas wyskalowany w sekundach na osi y. Zauważ, że w obliczeniu x^y mod n założono, że x, y i n są generowane losowo, wszystkie o takiej samej długości w bitach, i w żadnej specjalnej postaci. Założono na przykład, że technika optymalizacji Brickell'a (zobacz [Brick] i brick.c) nie ma zastosowania. Czasy są pokazywane poprawnie dla 8192 bitowych modułów. Czasy dla mniejszych modułów są stopniowo zwiększane o 8. Więc czasy pokazane dla modułów 4096 bitowych powinny być dzielone przez 8, dla modułów 2048 bitowych dzielone przez 64, itd. Kompletnie rozwinięty jest niewykonalny w praktyce dla dużych modułów i dlatego czasy dla tych metod nie zostały podane.

Zauważ, że metoda Comba jest optymalna dla modułów 512 bitowych i mniejszych. Implikuje to, że będzie optymalną technika dla szybkich implementacji krzywej eliptycznej GF(p) i dla 1024 bitowego deszyfrowania RSA (które wymaga dwóch 512 bitowych potęgowań i zastosowania Chińskiego Twierdzenia o Resztach. Jednakże te wnioski zależą od procesora i globalnie mogą nie być prawdziwe. Również metoda Comba może generować dużo kodu a może to być ważnym czynnikiem w jakiejś aplikacji.

Wspierając programistę przy generowaniu kodu dla procesora w niestandardowym środowisku (np. wbudowany sterownik), kod dla dynamicznego alokowania pamięci jest zawsze wywoływany z modułu mralloc.c Domyślnie wywołuje to standardowe funkcje czasu wykonania C calloc i free. Jednakże łatwo może to być zmodyfikowane przy zastosowaniu alternatywnie zdefiniowanego przez użytkownika mechanizmu alokacji pamięci. Z tego samego powodu wszystkie wyjścia i wejścia ekran / klawiatura następują przez funkcje czasu wykonania fputc i fgetc. Przez przechwycenie wywołań do tych funkcji, I/O może być przekierowane do niestandardowego urządzenia.

6 LICZBY FLOATING - SLASH

Prostym sposobem przedstawiania liczb ułamkowych jest redukcja ułamków, ponieważ licznik i mianownik ze wspólnym współczynnikiem znoszą się. Liczby te mogą potem być dodawane, odejmowane ,mnożone i dzielone w sposób oczywisty a wynik zredukowany przez podzielenie licznika i mianownika przez ich największy wspólny dzielnik. Skuteczny podprogram GCD, stosujący modyfikację Lehmera klasycznego algorytmu euklidesowego dla liczb o dużej dokładności, jest zawarty w pakiecie MIRACL.

Alternatywnym sposobem przedstawienia liczb ułamkowych byłaby metoda ułamków łańcuchowych. Każda liczba ułamkowa może być zapisana jako

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

lub bardziej elegancko jako p/q = $[a_0/a_1/a_2/.../a_n]$ gdzie a_i są dodatnimi liczbami całkowitymi, zazwyczaj małymi. Na przykład

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

Zauważ, że powyższe elementy a_i przedstawiające ułamek łańcuchowy są łatwo znajdowane jako iloraz generowany jako produkt uboczny, kiedy jest stosowany algorytm euklidesowy GCD do p i q.

Ponieważ jesteśmy zaangażowani w przedstawianie liczb wymiernych o stałej długości, problem powstaje kiedy wynik jakiejś operacji przekracza tą stałą długość. Jest taka konieczność przy pewnych schematach obcinania lub zaokrąglania. Ponieważ nie ma oczywistego sposobu obcięcia dużej liczby ułamkowej jest prosty sposób obcinania reprezentacji ułamków łańcuchowych. Wynikowy, mniejszy ułamek jest zwany najlepszym wymiernym przybliżeniem lub zbieżnością oryginalnego ułamka.

Rozważmy obcięcie 277 / 642 = [0/2/3/5/1/3/3]. Po prostu opuszczamy ostatni element z reprezentacji CF, co daje [0/2/3/6/1/3] = 85 / 197, co jest bardzo bliskim przybliżeniem do 277 / 642 (błąd = 0,0018%). Przycinanie kolejnych elementów z CF daje nam kolejne zbieżności jako 22/51, 19/44, 3/7 ,1/2 ,0/1. Ponieważ ułamek robi się mniejszy wzrasta błąd. Oczywiście zasada obcinania dla implementacji komputerowej powinna wybrać największą zbieżność , która mieści się w reprezentacji komputerowej.

Typ zaokrąglania opisany powyżej jest również nazywany „zaokrąglaniem Medianty". Jeśli p/q i r/s są dwoma sąsiadującymi przedstawionymi liczbami slash , wtedy ich medianta jest przedstawiana jako (p+r)/(q+s). Wszystkie większe ułamki pomiędzy p/q a mediantą będą zaokrąglane do p/q a te pomiędzy r/s a mediantą do r/s. Sama medianta zaokrągla do „prostszego" z p/r i r/s.

Teoretycznie jest to bardzo dobry sposób zaokrąglania, dużo lepszy niż raczej arbitralna metoda używana w arytmetyce zmienno przecinkowej, i jest metodą używaną tutaj. Pełna teoretyczna podstawa arytmetyki floating - slash jest opisana szczegółowo przez Matule & Kornerupa. Powinieneś zauważyć, że nasze przedstawienie flash jest faktycznie skrzyżowaniem, systemów fixed i floating - slash analizowanymi przez Matulę i Kornerupa, ponieważ nasz slash może przepływać miedzy słowami a nie między bitami. Jednak charakterystyka typu danej flash będzie zmierzała do tych floating-slash, ponieważ wzrasta precyzja .

Podprogram miracl mround implementuje zaokrąglanie medianty. Jeśli wynik operacji arytmetycznej jest ułamkiem p/q, wtedy euklidesowski algorytm GCD jest przed p i q. Jednak tym razem celem nie jest użycie algorytmu do obliczenia GCD, ale użycie jego współczynnika do zbudowania kolejnych zbieżności do p/q. Proces ten jest zatrzymany kiedy następną zbieżność jest zbyt duża aby pomieścić reprezentację flash Kompletny algorytm jest dany poniżej

Dane są $p \geq 0$ i $q \geq 1$

$b_{-2} = p$ $\quad x_{-2} = 0$ $\quad y_{-2} = 1$
$b_{-1} = q$ $\quad x_{-1} = 1$ $\quad y_{-1} = 0$

Teraz dla i =0,1,..... i $b_{i-1} > 0$ znajdujemy iloraz a_i i resztę b_i kiedy b_{i-2} jest dzielone przez b_{i-1}, tak, że

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

wtedy obliczamy

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

Zatrzymujemy kiedy x_i / y_i jest zbyt duże aby pomieścić reprezentację flash i pobiera x_{i-1} / y_{i-1} jako wynik zaokrąglony.

Jeśli użyjemy 277 / 642, to przetwarzanie da taką samą sekwencję zbieżności jak podana wcześniej.

Ponieważ ta procedura zaokrąglania musi być stosowana do każdej arytmetycznej operacji, i ponieważ jest potencjalnie raczej wolna, dużym wysiłkiem było uczynienie optymalizacji tej implementacji. Pomysł Lehmera działania tylko z najbardziej znaczącą częścią każdej liczby tak długo jak to możliwe jest używana, dlatego dla większości iteracji potrzebna jest tylko arytmetyka o pojedynczej precyzji Specjalnie troszczyć się trzeba aby uniknąć zaokrąglania wyniku wybiegającego daleko poza ograniczenia dla przedstawiania flash. Zastosowanie podstawowych podprogramów arytmetycznych do obliczania funkcji elementarnych takich jak log(x), exp9x), sin(x), cos(x), tan(x) itd., wymaga szybkich algorytmów opisanych przez Brenta.

W wielu przypadkach wynik dany przez program może gwarantować dokładność. Może to być sprawdzone przez przetestowanie instancji zmiennej EXACT, który jest inicjalizowany przez TRUE i jest ustawiany na FALSE tylko jeśli zaokrąglanie ma miejsce.

Wadą użycia zmiennej typu flash do przybliżenia arytmetyki rzeczywistej jest nie -ujednolicenie rozmiaru przerwy pomiędzy reprezentowanymi wartościami.

Dla zilustrowania tego rozważymy system floating-slash który jest ograniczony do iloczynu licznika i mianownika mniejszego niż 256. Zauważ, że pierwszy reprezentowalny ułamek mniejszy niż 1/1 w takim systemie to 15/16, przerwa 1/16. Następny ułamek większy niż 0/1 to 1/255, przerwa 1/255. Generalnie rzecz biorąc, dla k-bitowego systemu floating-slash, rozmiar przerwy zmienia się od mniejszego niż 2^{-k} do najbardziej niekorzystnego przypadku $2^{-k/2}$. W praktyce oznacza to, że rzeczywista wartość, która wypadła w jednej z większych przerw, będzie przedstawiona przez ułamek, który będzie dokładny tylko w połowie swojej zwykłej precyzji. Na szczęście takie duże przerwy są rzadkie, i coraz częściej dla wyższej precyzji, występują tylko blisko prostych ułamków. Jednakże to znaczy, że wynik rzeczywisty może być tylko całkowicie zaufany do połowy danych miejsc dziesiętnych. Częściowym rozwiązaniem tego problemu byłoby przedstawienie liczb wymiernych bezpośrednio jako ułamki łańcuchowe. Daje to dużo lepsze ujednolicenie rozmiarów przerw, ale byłoby bardzo trudno zaimplementować przy użyciu języków wysoko poziomowych.

Arytmetyka na typach danych flash jest bez wątpienia wolniejsza niż na odpowiednich typach zmienno przecinkowych o podwyższonej precyzji. Zaletą metody flash jest jej zdolność do dokładnego przedstawiania liczb wymiernych, i dokładnej arytmetyki na nich. Nawet kiedy zaokrąglanie jest konieczne, wynik często obliczany jest poprawny, wskutek tego zaokrąglanie medianą ma skłonność do preferowania prostego ułamka niż złożonego. Na przykład program roots (Rozdział 8), kiedy prosi o znalezienie pierwiastka kwadratowego z 2 a potem podnosi do kwadratu wynik, wraca z dokładną odpowiedzią 2, pomimo duzego wewnętrznego zaokrąglania.

UWAGA! NIE mieszaj arytmetyki flash z wbudowaną arytmetyka double. One nie łączą się dobrze. Jeśli decydujesz się użyć arytmetyki flash, użyj jej wszędzie i skonwertuj wszystkie stałe na starcie do typu flash. Nawet lepiej określić taką stałą, jeśli to możliwe jako ułamek. Więc (w C++) jest bardziej pożądany zapis

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

zamiast

```
x = .625 ;
```

7. INREFEJS C++

Wielu użytkowników pakietu MIRACL jest zawiedzionych, ze muszą obliczać

$t = x^2 + x +1$

dla zmiennej flash x przez sekwencję

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

zamiast po prostu

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

Ktoś oczywiście może użyć biblioteki MIRACL do napisania specjalizowanego kompilatora C, który mógłby poprawnie zinterpretować taką instrukcję. Jednakże taki drastyczny krok nie jest konieczny. Nadzbiór C, nazwany C++ uzyskał ogólną akceptację jako naturalny następca C. Rozszerzenia C są głownie skierowane w kierunku uczynienia z niego języka zorientowanego obiektowo. Poprzez zdefiniowanie zmiennych big i flash jako klas (w terminologii C++) jest możliwe 'przeładowanie' zwykłych matematycznych operatorów aby kompilator automatycznie zastąpił wywołanie do właściwych podprogramów MIRACL, kiedy te operatory są używane wraz ze zmiennymi big i flash. Ponadto C++ może zająć się inicjalizacja (i ostateczną eliminacją) tych typów danych automatycznie, używając mechanizmu konstruktora / dekonstruktora, który jest zawarty w definicji klasy. Uwalnia to programistę od nudnego wyraźnego inicjalizowania każdej zmiennej big i flash przez powtarzanie wywołania mirvar. Istotnie, kiedy klasy są zdefiniowane i ustawione właściwie, łatwiej jest pracować z nowymi typami danych podobnie jak z wbudowanymi typami danych double i int. Stosowanie C++ również pomaga osłonić użytkownika od wewnętrznych warunków pracy MIRACL.

Biblioteka MIRACL jest połączony do C++ poprzez pliki nagłówkowe big.h, flash.h, monty.h, gf2m.h, elliptic.h i ec2.h. Implementacja funkcji jest w skojarzonych plikach big.cpp, flash.cpp, monty.cpp, gf2.cpp, elliptic.cpp i

ec2.cpp, które muszą być powiązane z aplikacjami, które ich wymagają. Chinskie Twierdzenie o Resztach jest również elegancko zaimplementowane jako klasa, w plikach crt.h i crt.cpp. Zobacz decode.cpp jako przykład zastosowania. Metoda Brickella i innych dla szybkiego modularnego potęgowania z wcześniejszym obliczaniem jest zaimplementowany w brick.h. Zobacz brick.cpp po przykład zastosowania. Odpowiednik krzywej eliptycznej GF(p) jest w ebrick.h i ebrick.cpp a odpowiednik krzywej eliptycznej GF(2^m) w ebrick2.h i ebrick2.cpp.

PRZYKŁAD

```
/*
•   Program do obliczania silni
*/

#include   <iostream>
#include   „big.h”                    /* włączamy system MIRACL */

using namespaced str;

Miracl precision (500, 10)            // Upewniamy się, że MIRACL jest zainicjalizowany przed wywołaniem
                                      // main()

void  main()
{ /* obliczanie silni liczby */
        Big  nf = 1;                  /* deklaracja zmiennej „Big” nf */
        int   n;
        cout <<”program silni\n”;
        cout   << „liczba wejściowa n = \n”;
        cin >> n;
        while (n>1)
                nf* = (n--);          /* nf =n! =n*(n-1)*(n-2)* ....3*2*1   */
cout << „n! = \n” << nf << „\n”;

}
```

Porównaj to z wersją w C z rozdziału 3. Zauważ, że staranne użycie fikcyjnej klasy Miracl używa zbioru dokładnych zmiennych big. Ich deklaracja w globalnym zakresie zapewnia ,że MIRACL jest inicjalizowana przed wywołaniem main() (Zauważ też, że nie będzie to poprawne w środowisku wielowątkowym) .Kiedy kompilujesz i linkujesz taki program, nie zapomnij zlinkować klasy Big zaimplementowanej w pliku big.cpp.

Konwersja to / z wewnętrznego formatu Big jest ważne:

Konwersja ciągu znaków hex do Big

```
Big     x;
char  c[100];

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

Konwersja Big to ciągu znaków hex

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

Do konwersji do / z czystej postaci binarnej stosujemy funkcje zaprzyjaźnione from_binary () i to_binary ()

```
        int         len;
        char        c[100];

        Big      x = from_binary (len, c-;
                 //tworzy Big x z len bajtów binarnych w c
len   = to_binary(x, 100, c, FALSE);
//konwertuje Big x do len bajtów binarnych w c[100]
```

```
len  = to_binary(x, 100, c ,TRUE);
    //konwertuje Big  x do len bajtów binarnych w c[100]
    //(wyrównanymi do prawej czołowymi zerami)
```

W wielu przykładowych programach, szczególnie programach ułamkowych, cała arytmetyka jest robiona w mod n. Dla uniknięcia nużących redukcji mod n wymaganych po każdym działaniu, zostanie zastosowana nowa klasa C++ ZZn i zdefiniowana w pliku monty.h. Ta klasa ZZn ma swoje arytmetyczne operatory zdefiniowane do automatycznego wykonania redukcji. Funkcja modulo(n) ustawia moduły. W analogiczny sposób klasa C++ GF2m działa z elementami pola zdefiniowanego ponad GF(2^m). W tym przypadku „moduł" jest ustawiony poprzez modulo (m., a, b, c), który również określa albo podstawę trinomialną $t^m + t^a + 1$ (i ustawia b=c=0) albo podstawę pentanomialną $t^m + t^a + t^b + t^c +1$. Po szczegóły zajrzyj do dokumentacji IEEE P1363.

Wewnętrznie klasa ZZn używa reprezentacji Montgomery'ego. Zobacz monty.h. Zauważ, że wewnętrzna implementacja ZZn jest ukryta przed programistą, klasyczna cecha C++. A zatem programista C++ nie musi się martwić o wewnętrzną niewygodną reprezentację Montgomery'ego.

Klasa ECn zdefiniowana w elliptic.h wykonuje manipulowanie wskazaną krzywą eliptyczną GF(p) w prosty sposób, ukrywając wszystkie zbędne szczegóły. Klasa EC2 zdefiniowana w ec2.h robi to samo dla krzywej eliptycznej GF(2^m)

Prawie cała funkcjonalność MIRACL'a jest osiągalna w C++. Programowanie może często być robione intuicyjnie, bez odnoszenia się do tego materiału, używając dobrze znanej składni C jak zilustrowano powyżej. Inne funkcje są dostępne przy użyciu 'oczywistej' składni - jak w przykładzie x= gcd (x, y); ,lub y = sin(x);. Po więcej szczegółów zajrzyj do plików nagłówkowych i przykładowych programów.

Wersje C++ większości przykładowych programów są zawarte na dostarczonym nośniku, z rozszerzeniem pliku .cpp

Jednym z problemów z manipulowaniem dużymi obiektami jest skłonność kompilatora do generowania kodu tworzącego/ niszczącego wielokrotne obiekty czasowe. Domyślnie MIRACL uzyskuje pamięć ze sterty dla zmiennych Big i Flash. Może to być trochę czasochłonne, a wszystkie takie obiekty muszą być ostatecznie usunięte. Byłoby szybciej przydzielić pamięć zamiast brać ze stosu. Może być to osiągnięte przez zdefiniowanie BIGS = n w czasie kompilacji. Na przykład jeśli używasz kompilatora Microsoft C++ w lini poleceń:-

```
C:miracl >c1 /02 /DBIGS=50 brent.cpp big.cpp monty.cpp miracl.lib
```

Zauważ, że wartość n powinna być taka sama jak określona w wywołaniu mirsys (n, 0); lub w Miracl precision = n; w programie głównym.

Nie jest to zalecane dla projektowanych programów lub jeśli obiekty są bardzo duże. Działą to tylko z programami w C++.

W końcu masz tu bardziej złożony program w C++ implementujący względnie złożony protokół kryptograficzny.

```
/*
•  ID Gunther'a oparty o wymianę klucza - skończona wersja pola
•  Zobacz RFC 1824
•  wariant r^r (z Perfect Forward Security)
*/

#include <iostream>
#include <fstream>
#include „monty.h"

using namespoaced std;
Miracl  precision = 100
char  *IDa  = „Identity 1';
char  *IDb  =  „Identity 2";

//Funkcja kodująca
Big H (char *ID)
{ //kodowanie ciągu znaków do 160 bitowej liczby big
        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");          //tworzenie pliku  strumieniowego
        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);

//pobranie  danych. W hex. G^q mod p =1
    common >> bits;
    mip -> IOBASE=16;
    common >> p >> q >> g;
    mip -> IOBASE=10;
    modulo(p);              // ustawienie modułu
    G = (ZZn) g;

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

//CA generuje swój tajny i publiczny klucz
        x = rand(q)                  // tajny klucz CA, 0 < x < q
        Y = pow(G,x);                //publiczny klucz CA, 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) + (k8ra)%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 calcualtion - 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
// Klucz = 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);                     // zwiększone potęgowanie
      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);                     //zwiększone potęgowanie
      cout <<” Key = \n”<< Key <<endl;
      Return 0;
}
```

8 PRZYKŁADY PROGRAMÓW

Notka: Programy opisane tutaj są natury eksperymentalnej, i w wielu przypadkach nie są kompletnie ‘dokończone’. Po dalsze informacje czytaj komentarze powiązane z właściwym plikiem źródłowym.

8.1 Proste programy

8.1.1 hail.c

Program ten pozwoli ci zbadać tak zwane liczby hailstone, opisane przez Grunenbergera [Gruen]. Prodedura jest prosta. Zaczynasz z jakąś liczbą stosując następujące zasady:

(a) Jeśli jest nieparzysta, mnożymy ją przez 3 i dodajemy 1
(b) Jeśli jest parzysta, dzielimy ją przez 2
(c) Powtarzamy proces dopóki liczba nie stanie się równa 1, wtedy zatrzymujemy

Wydawałoby się, że dla każdej liczby początkowej to przetwarzanie ostatecznie się skończy, chociaż nie udowodniono ,że tak musi się zdarzyć, lub, że ten proces nie może utknąć w pętli nieskończonej. Co idzie w górę, wydaje się , musi zejść w dół. Wypróbuj program dla wartości początkowej 27. Potem wypróbuj go używając dużo większej liczby, takiej jak 10709980568908647 (która ma ciekawe zachowanie)

8.1.2 palin.c

Ten program pozwala zbadać odwracanie palindromiczne [Gruen]. Liczba palindromiczna jest liczbą, którą czyta się tak samo w obu kierunkach. Zacznij z jakąś liczbą i zastosuj poniższe zasady.

(a) Dodaj liczbę do liczby uzyskanej przez odwrócenie porządku cyfr. Zrób nowa liczbę
(b) Zatrzymaj proces, kiedy nowa liczba jest palindromiczna

Wydaje się, że dla większości liczb początkowych ten proces zakończy się szybko. Spróbuj z 89. POtem zpróbuj z e 196

8.1.3 mersenne.c

Program ten próbuje wygenerować wszystkie liczby pierwsze z postaci 2^n-1. Największa znana liczba pierwsza zawsze jest w tej postaci z powodu wydajności testu Lucas - Lehmera. Podprogram fft_mult jest używany ponieważ jest szybszy dla bardzo dużych liczb

8.2 Programy rozkładania na czynniki pierwsze

Jest zawartych sześć różnych programów Całkowitego Rozkładania na Czynniki Pierwsze, pokrywając wszystkie nowoczesne podejścia do tego klasycznego problemu. Więcej informacji na temat tego algorytmu można znaleźć w {Scott89c]

8.2.1 brute.c

Program ten próbuje rozłożyć na czynniki pierwsze liczbę poprzez dzielenie „na siłę", używając tablicy małych liczb pierwszych. Kiedy próbujemy trudnego rozkładu, ma sens wypróbowanie najpierw tego podejścia. Rozłóż na czynniki pierwsze 12345678901234567890 używając tego programu. Potem wypróbuj go na dużej liczbie losowej.

8.2.2 brent.c

Ten program próbuje rozkładu na czynniki pierwsze używając metody Brent'a - Pollarda. Ta metoda jest szybsza przy znajdowaniu dużych współczynników niż nastawiona na proste podejście „na siłę". Jednakże, nie zawsze kończy się pomyślnie, nawet dla prostego rozkładania. Użyj go do rozłożenia na czynniki pierwsze R17, to jest 11111111111111111 (siedemnastu jedynek). Potem wypróbuj go na dużej liczbie, która nie uległa podejściu „na siłę"

8.2.3 pollard.c

Inny program, który implementuje metodę Pollarda (p-1), wyspecjalizowany w szybkim znajdowaniu współczynnika p liczby N dal każdego (p-1) mającego swój tylko mały współczynnik. Faza 1 tej metody działa jeśli wszystkie te małe współczynniki są mniejsze niż LIMIT1. Jeśli Faza1 nie powiedzie się wtedy Faza2 szuka jednego, dużego, końcowego współczynnika mniejszego niż LIMIT2. Stałe LIMIT! I LIMIT2 są ustawiane wewnątrz programu.

8.2.4 williams.c

Ten program jest podobny do metody Pollarda, ale może znajdować współczynnik p z N dla każdego (p+1) mającego tylko małe współczynniki. Ponownie są używane dwie fazy. Faktycznie ta metoda jest czasami metodą (p+1) a czasami (p-1), więc kilka prób jest czynionych aby trafić w warunek (p+1). Algorytm ten jest raczej bardziej złożony niż używany w metodzie Pollarda i czasami wolniejszy.

8.2.5 lenstra.c

Lenstra [Monty87] odkrył nową metodę rozkładania na czynniki pierwsze, generalnie podobną do metody Pollarda i Wiliamsa, ale potencjalnie dużo bardziej mocniejsza. Działa poprzez losowe generowanie Krzywej Eliptycznej, który może być potem używany do znajdowania współczynnika p z N, dla każdego p+1-δ mającego tylko małe współczynniki, gdzie δ zależy od wyboru szczególnej krzywej. Jeśli jedna krzywa jest niepoprawna, wtedy może być wypróbowana inna, opcja nie możliwa w metodach Pollarda/ Williamsa . Ponownie są dwie fazy metody i chociaż ma bardzo dobre zachowania asymptotyczne, jest dużo wolniejsza niż metody Pollarda/ Williamsa dla każdej iteracji.

8.2.6 qsieve.c

Jest to wymyślny program rozkładu na czynniki pierwsze Pomerance'a - Silvermana'a - Montgomery'ego ze współczynnikiem F7 = 2^{128}=1

340282366920938463463374607431768211457

w mniej niż 30 sekund, na komputerze opartym o Pentium 60 MHz. Kiedy ta liczba był pierwszy raz rozkładana na czynniki pierwsze, zabrało to 90 minut na mainframe'ie IBM 360 (Morrison & Brillhant[Morrison]), aczkolwiek używa nieco gorszego algorytmu.

Jego specjalnością jest rozkład na czynniki pierwsze wszystkich liczb (do długości około sześćdziesięciu cyfr), bez względu na rozmiar współczynników. Jeśli liczba rozkładana to N, wtedy program w rzeczywistości dzuiałą z liczbą k.N, gdzie k jest małym mnożnikiem Knuth-Schroepel'a. Sam program ustala najlepszą wartość k do użycia. Wewnętrznie program używa ' bazy współczynnika' małych liczb pierwszych. Duża liczba, większa będzie tym bazowym współczynnikiem. Program działa przez gromadzenie informacji z kilku prostych rozkładów. W miarę postępów w drukuje working...n. Kiedy sądzi , ze ma dosyć informacji drukuje trying, ale te próby mogą być przedwczesne i nie zakończy się sukcesem. Program zawsze kończy się przed liczbą n w obszarze working...n rozmiaru współczynnika bazowego.

Program ten używa dużo więcej pamięci niż każdy inny program przykładowy, szczególnie kiedy rozkładamy duże liczby. Ilość pamięci, którą program może pobrać jest ograniczona poprzez wartość zdefiniowaną dla MEM, MLF i SSIZE na początku tego programu. Ogranicza to, odpowiednio, ilość liczb pierwszych w bazie współczynników, liczbę 'dużych' liczb pierwszych używanych przez tak zwane wariacje dużych - pierwszych liczb algorytmu i sito rozmiaru. Powinno to, jeśli możliwe, być zwiększane lub redukowane jeśli twój komputer ma niewystarczającą pamięć. Zobacz [Silverman] po szczegóły.

Użyj qsieve do współczynnika 10000000000000000000000000000000009 (trzydzieści pięć cyfr)

8.2.7 factor.c

Program ten łączy powyższy algorytm w pojedynczym programie ogólnego przeznaczenia dla rozkładu całkowitego. Każda metoda jest używana po kolei w próbach wyodrębnienia współczynników. Liczba do rozkładu jest podana w linii poleceń, jako współczynnik 11111111111. Liczba może być ewentualnie okreslona jako formuła, używając przełącznika '-f', jako factor -f (10#11-1)/ 9. Symbol # ma tu znaczenie „do potęgi „ (# jest używane zamiast ^ ponieważ ostatni symbol ma specjalne znaczenie dla DOS na IBM PC). Typ factor służy dla pełnego opisu tego i innych przełączników , które mogą stosowane do sterowania wejścia/ wyjścia tego programu

8.3 Programy logarytmu dyskretnego

Dwa programy implementują algorytm Pollarda dla uzyskania logarytmu dyskretnego. Problemem logarytmu dyskretnego jest znalezienie x danego y , r i n w

$y = r^x \bmod n$

Powyższe jest dobrym przykładem funkcji jednokierunkowej. Łatwo jest obliczyć y danego z , ale najwyraźniej niezmiernie trudno znaleźć x danego y. Algorytm Pollarda jednak wykonuje to całkiem dobrze pod pewnymi warunkami, jeśli x jest znane jako małe lub jeśli n jest liczbą pierwszą p dla każdego p - 1, mającego tylko mały współczynnik.

8.3.1 kangaroo.c

Program ten znajduje x w powyższym równaniu, zakładając, że x jest całkiem małe. Wartość r jest stała (16) a moduł n jest również stałe wewnątrz programu. Początkowo 'pułapka' jest ustawiona. Później logarytm dyskretny może być znaleziony (prawie na pewno) dla każdej liczby, zakładając, że logarytm dyskretny jest mniejszy niż pewny górny limit. Liczba kroków wymagana będzie w przybliżeniu pierwiastkiem kwadratowym tego limitu.

8.3.2 genprime.c

Liczba pierwsza p ze znanym współczynnikiem p-1 jest generowana przez ten program, dla zastosowania przez programy index.c i identity.c opisane poniżej. Współczynnik p-1 jest daną wyjściowa dla pliku prime.dat

8.3.3 index.c

Program ten implementuje algorytm Polarda dla ekstrahowania logarytmów dyskretnych, kiedy moduł n w powyższym równaniu jest liczbą pierwsza, i kiedy p-1 ma tylko stosunkowo mały współczynnik. Liczba kroków wymaganych jest funkcją pierwiastka kwadratowego większego z tych współczynników.

8.4 Kryptografia klucza publicznego

Kryptografia Klucza Publicznego jest systemem kryptograficznym dwóch kluczy, z bardzo pożądaną cechą, jaką jest to, że kodowany klucz może być dostępny publicznie, bez osłabiania siły szyfru. Pierwszym przykładowy program demonstruje wiele popularnych technik klucza publicznego. Potem dwa funkcjonalne systemy

kryptograficzne Klucza Publicznego, których siła objawia się w zależności od trudności rozłożenia na czynniki pierwsze, jak przedstawiono. Pierwszy jest klasycznym system RSA (Rivest, Shamir i Adlemann). Jest szybki do kodowania wiadomości, ale dotkliwie wolny przy dekodowaniu. Dużo szybsza technika została wymyślona przez Bluma i Goldwassera. Ten probabilistyczny system Klucza Publicznego jest również silniejszy niż RSA w jakimś sensie. Po więcej szczegółów zajrzyj do [Brassard], który opisuje to jako „najlepsze co akademia miała do zaoferowania". W obu metodach klucze są skonstruowane z 'silnych' liczb pierwszych poprawiających bezpieczeństwo. Blisko powiązana z Kryptografią KP jest koncepcja Podpisu Cyfrowego. Grupa przykładowych programów implementuje Standard Podpisu Cyfrowego, używając klasycznych skończonych pól i krzywych eliptycznych dla pól GF9p) i GF(2^m).

8.4.1 pk-demo.c

Program ten przeprowadza wymianę 1024- bitowego klucza Diffie-Hellmana, a potem innego typu wymianę klucza, ale tym razem opartego o 160 bitową liczbę pierwszą i krzywa eliptyczną. Następnie testowany ciąg jest szyfrowany i deszyfrowany metodą El Gamala. Program kończy się 1024 bitowym szyfrowaniem / deszyfrowaniem RSA tego samego ciągu. Po dobry opis wszystkich tych technik zobacz {Stinton]. Przy każdej próbie implementacji systemu PK przy uzyciu MIRACL jest mocno zalecana analiza tego pliku i jego odpowiednika w C++, pk-demo.cpp

8.4.2 bmark.c / imratio.c

Program testowy bmark.c pozwala użytkownikowi szybko określić czas, który będzie wymagany do implementacji każdej z popularnych metod klucza publicznego. Może być skompilowany i zlinkowany z każdym wariantem biblioteki MIRACL, jaki określono w mirdef.h, określając który daje najlepszą wydajność na poszczególnej platformie dla określonej metody KP. Program imratio.c, skompilowany i zlinkowany oblicza znaczące współczynniki S/M., I/M. i J?M. gdzie S jest to okres modularnego podnoszenia do kwadratu, M. okresem modularnego mnożenia, I okresem dla modularnej inwersji a J okresem obliczania symbolu Jacobi'ego.

8.4.3 genkey.c

Program ten generuje 'publiczny' klucz kodujący i 'prywatny' klucz dekodujący, które są konieczne dla obu, oryginalnego systemu KP Rivera - Shamira - Adlemanna i ważniejszej metody Bluma - Goldwasera. Klucze te mogą zabrać duż0 czasu przy generowaniu, ponieważ są one formowane z bardzo dużej liczby liczb pierwszych, które muszą być wygenerowane ostrożnie dal maksymalnego bezpieczeństwa.

Rozmiar każdej liczby pierwszej w bitach jest ustawiany wewnątrz programu przez #define. Bezpieczeństwo systemu zależy od trudności rozkładu na czynniki pierwsze 'publicznego' klucza kodującego, który jest formowany z dwóch tak dużych liczb pierwszych. Największa liczby, które mogą być rutynowo rozłożone używające setek potężnych komputerów, są długie na 430 bitów (1996). Więc minimalny rozmiar 512 bitów dla każdej liczby pierwszej daje pełne bezpieczeństwo (na chwilę obecną!)

Po tym jak program ruszy, klucze są tworzone w plikach PUBLIC.KEY i PRIVATE.KEY

8.4.4 encode.c

Wiadomości lub pliki mogą być zakodowane tym programem, który używa 'publicznego' klucza kodującego z pliku PUBLIC.KEY, wygenerowanego przez program genkey, który musi być uruchomiony wcześniej zanim użyjemy tego programu. Kiedy działa, użytkownik jest proszony o plik do zaszyfrowania. Albo dostarcza nazwy pliku tekstowego, albo wciska return wprowadzając bezpośrednio z klawiatury, W pierwszym przypadku, zakodowane dane wyjściowe są wysyłane do pliku o takiej samej nazwie, ale z rozszerzeniem .RSA. W drugim przypadku wprowadza nazwę pliku wyjściowego, który musi być podany. Tekst wprowadzany z klawiatury musi być zakończony przez CONTROL-Z (znak końca pliku)

8.4.5 decode.c

Wiadomość lub plik zakodowane w systemie RSA mogą być zdekodowane przy użyciu tego programu, który używa 'prywatnego' klucza dekodującego, z pliku PRIVATE.KEY wygenerowanego przez program genkey, który musi być uruchomiony na wcześniejszym etapie używania tego programu.Kiedy działa, użytkownik jest zachęcany do podania nazwy pliku do odkodowania. Wpisuje nazwę pliku (bez rozszerzenia - program zakłada rozszerzenie .RSA) i wciskamy return. Potem użytkownik jest proszony o podanie nazwy pliku wyjściowego. Albo dostarcza nazwę pliku albo wciska return, w każdym przypadku plik zdekodowany będzie wysłany bezpośrednio na ekran. Problem z systemem RSA staje się natychmiast oczywisty - dekodowanie pobiera relatywnie całkiem dużo czasu! Jest toi zwłaszcza prawdziwe dla kluczy o dużych rozmiarach i długich wiadomości.

8.4.6 enciph.c

Program ten działa w identyczny sposób jak program ‘encode’, z wyjątkiem tego, że powoduje wywołanie losowej metody ziarna przed zaszyfrowaniem danej. Ta losowa metoda ziarna jest potem użyta wewnętrznie do wygenerowania dużej liczby losowej. Proces kodowania zależy od tej liczby losowej, co oznacza, że ta sama dana nie koniecznie będzie tworzyć taki sam tekst zaszyfrowany, co jest jedna z silniejszych stron tego podejścia. Tak jak jest tworzony plik z rozszerzeniem .BLG zawierającym dane zakodowane, również jest tworzony drugi mały plik (z rozszerzeniem .KEY).

8.4.7 deciph.c

Program ten działa identycznie jak program ‘decode’. Jednakże jego zaletą jest to, że dużo bardziej szybciej. To będzie znaczące początkowe opóźnienie, podczas gdy jest wykonywane raczej złożone obliczenie. Tu jest używany prywatny klucz a dane w pliku .KEY odzyskuje dużą liczbę losową używaną w procesie kodowania. Od tego czasu deszyfrowanie jest tak szybkie jak szyfrowanie.

8.4.8 dssetup.c

Standardowa metoda dla podpisu cyfrowego zaproponowana przez Narodowy Instytut Standardów i Technologii (NIST), i w pełni opisany w Standardzie Podpisu Cyfrowego [DSS]. Program generuje liczbę pierwszą q, inną, większą liczbę pierwszą p = 2nq + 1 (gdzie n jest losowe) i generator g. Te informacje są dostępne wszystkim. Program generuje powszechne informacje {p,q,g} w pliku common.dss.

8.4.9 limlee.c

Został pokazany przez Lee i Lima [LimLee], który na pewno opiera się na protokole logarytmu Dyskretnego (ale nie Dla Standardu Podpisu Cyfrowego), który jest słabo powiązany z rodzajem generowanych liczb pierwszych przez program dssetup.c omówiony powyżej. .Unikając tych problemów, rekomendują oni postać p jako p = $2.p_1.p_2.p_3....q+1$, gdzie p_i są liczbami pierwszymi większymi niż q. Program ten generuje wartości (p, q,g) do pliku common.dss i może być użyty w miejsce dssetup.c. Jest jednak wolniejszy.

8.4.10 dssgen.c

Każdy pojedynczy użytkownik, który życzy sobie cyfrowego podpisu w pliku komputerowym, losowo generuje swój własny prywatny klucz x < q i czyni dostępnym klucz publiczny $y = g^x$ mod p. Bezpieczeństwo systemu zależy od rozmiaru p i q (przynajmniej, odpowiednio, 512 bitów i 160 bitów) program ten generuje pojedynczą parę klucza publicznego / prywatnego w plikach public.dss i private.dss.

8.4.11 dssign.c

Program używa klucza prywatnego z private.dss do ‘oznakowania’ dokumentu przechowywanego w pliku. Najpierw plik z danymi jest ‘haszowany’ do 160 bitowej liczby używając SHA. Jest to również określone przez NIST i zaimplementowane w dostarczonym module mrsha.c. 160 bitowy ‘hash’ jest należycie ‘oznaczony’ jak opisano w [DSS], a podpis, w postaci dwóch 160 bitowych liczb, zapisany do pliku. Ten plik ma taką samą nazwę jak plik dokumentu, ale z rozszerzeniem .dss.

8.4.12 dssver.c

Program ten używa klucza publicznego z public.dss do zweryfikowania podpisu powiązanego z plikiem, jak opisano w [DSS]

8.4.13 ecsgen.c, ecsign.c, ecsver.c

Technika Podpisu Cyfrowego może być również zaimplementowana przy użyciu Krzywych Eliptycznych na polu GF(p) [Jurisic].Typowym zastosowaniem informacji w porządku {p,A,B,q,X,Y} jest wyciągnięcie z pliku common.ecs stworzonego przy użyciu jednego z algorytmów opisanych poniżej. Wartości te są określone w punkcie początkowym (X, Y) na krzywej eliptycznej $y^2 = x^3 + Ax + B$ mod p, która ma na niej q punktów. Zaletami są dużo mniejsze klucze publiczne dal takiego samego poziomu bezpieczeństwa. Mniejsze liczby mogą być używane ponieważ problem logarytmu dyskretnego jest najwyraźniej dużo bardziej trudniejszy w kontekście krzywej eliptycznej. To z kolei sugeruje, że arytmetyka krzywej eliptycznej jest również potencjalnie szybsza. Jednakże użycie małych liczb jest nieco zrównoważone przez bardziej złożone i zawiłe obliczenia.

Ten zbiór programów ma taką samą funkcjonalność jak te opisane powyżej dla standardu DSS. Zauważ, jednak, że rozszerzenie .ecs jest używane dla wszystkich wygenerowanych plików. Odczytaj komentarze w plikach źródłowych po więcej szczegółów.

8.4.14 ecsgen2.c, ecsign2.c, ecsver2.cpp

Programy te dostarczają takiej samej funkcjonalności jak te opisane powyżej, ale używają krzywej eliptycznej zdefiniowanej w polu GF(2^m). Zakresem informacji w tym przypadku jest wyciągnięcie z pliku common2.ecs w porządku {m.,A,B,q,X,Y,a,b,c}, gdzie (X, Y) określają punkt początkowy na krzywej eliptycznej $y^2 = x^3 + Ax^2 + B$ zdefiniowanej w GF(2^m). Parametry trinomialne i pentanomialne są odpowiednio również określone, $t^m + t^a + 1$ lub $t^m + t^a + t^b + t^c + 1$. W pierwszym przypadku b i c są zerami. W końcu, cf.q określa liczbę punktów na krzywej, tworząc duży współczynnik q liczby pierwszej i mały ko-współczynnik cf. Ten ostatni zazwyczaj to 2 lub 4

Plik common2.ecs może być stworzony przez program schoof2 opisany poniżej.

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

Problemem z kryptografią krzywej eliptycznej jest zbudowanie odpowiedniej krzywej. Jest to rzeczywiście dużo bardziej trudniejsze niż odpowiedni problem w całkowitym skończonym polu jaki zaimplementowano w programach dssetup.c / dssetup.cpp. Jednym z podejść jest metoda Złożonego Mnożenia, jaką opisano w aneksie do Standardowej Specyfikacji Kryptografii Klucza Publicznego IEEE P1363 (dostępnej w Sieci). Tu jest ona zaimplementowana przez program C++ cm.cpp i wspierającego go moduły comflash.cpp, fpoly.cpp, polly.cpp i powiązane pliki nagłówkowe.

Kiedy program działa używa argumentów linii poleceń. Na przykład cm

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

wygeneruje wspólną informację potrzebną do implementacji kryptografii krzywej eliptycznej w pliku common.ecs.

Jako alternatywa dla metody CM, krzywa losowa może być wygenerowana i wypunktowana na bezpośrednio wyliczonej krzywej. Jest to bardziej czasochłonne niż złożone mnożenie, ale może prowadzić do bardziej zabezpieczonej, mniej strukturalnej krzywej. Podstawowy algorytm jest stworzony przez Schoffa [Sch], [Blake] i jest praktyczny tylko z powodu użycia metody Szybkiej Transformacji Fouriera dla mnożenia / dzielenia w dużym stopniu polynomialnego . Zobacz mrfast.c. Jest bardzo wolny, ale wolniejszy niż cm. Na przykład:

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

oblicza punkty na krzywej $y^2 = x^3 - 3x + 35317045537 \bmod 2^{192} - 2^{64} - 1$

Krzywa ta jest wybrana losowo (w rzeczywistości 35317045537 to mój międzynarodowy numer telefonu). Odpowiedzią jest liczba pierwsza

6277101735386680763835789423127240467907482257771524603027

Bądź przygotowany na oczekiwanie, lub...

Użyj kompletu programów mueller, process i sea, które razem implementują lepszą, ale bardziej złożoną metodę Schoof - Elkies - Atkin dla obliczania punktów.

Przede wszystkim powinien być uruchomiony program mueller, do wygenerowania wymaganego Modularnego Wielomianu. To musi być zrobione - zawsze.Wielki zbiór Modularnego Wielomian, wielki rozmiar modułu liczb pierwszych, który może być użyty przez zainteresowaną krzywą eliptyczną. Zauważ, że program ten jest szczególnie trudny dla zasobów pamięci, tak jak i mówimy o długim czasie wukonania. Jednakże po godzinie najmniej będziesz miał dość eksperymentowania z Modularnym Wielomianem. Podobnie ja kwszystkie te programy,po prosty wypisujemy nazwę programu bez parametrów generując instrukcje do zastosowania. Również bądź pewny odczytu komentarzy na początku pliku źródłowego, w tym przypadku mueller.cpp

Następnie uruchom process, który przetwarza plik wyjściowy .raw modularnego wielomianu z mueller, do zastosowania z określonym modułem liczb pierwszych.

W końcu uruchom sea do obliczenia punktów na krzywej i opcjonalnie tworzy plik .ecs jak opisano powyżej

Na przykład:

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

generujemy wszystkie modularne wielomiany dla liczb pierwszych od 0 do 120, a dane na ich wyjściu do pliku mueller.raw. Wtedy te wielomiany są przetwarzane pod względem liczby pierwszej $p = 65112.2^{144}-1$, tworząc plik test160.pol. W końcu główna aplikacja sea oblicza punkty na krzywej $y^2 = x^3-3x=49 \mod p$

Może jest to bardziej skomplikowane do zastosowania, ale jest dużo szybsze niż schoof

Przeczytaj komentarze na początku sea.cpp po więcej onforamcji.

Dla krzywych eliptycznych nad $GF(2^m)$ może być użyty program schoof2 , który jest trochę podobny do schoof. Jest jednak wolniejszy, ale prawie użyteczny na współczesnym sprzęcie. Na przykład

Schoof2 1 52 191 9 -o common.ecs

oblicza punkty na krzywej $y^2 + xy = x^3 +x^2 +52$, nad polem $GF(2^{191})$,. Muszą być również określone odpowiednie nieredukowalne podstawy, w tym przypadku $t^{191} + t^9 + 1$. Tablica odpowiednich podstaw może być znaleziona w wielu dokumentach, na przykład w dodatku A standardu IEEE P1363.

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

Schematy klucza publicznego powinny być idealnie odporne na ataki adaptacyjne wybranym tekstem zaszyfrowanym, dzięki którym atakujący może uzyskać rozszyfrowany, dostępny tekst zaszyfrowany inny niż ten jakim jest zainteresowany. Ostatnio Cramer i Shoup[CS} wymyślili metodę szyfrowania Klucza Publicznego, która udowadnia odporność na taki silny atak. Program crsetup tworzy różne globalne parametry a crgen generuje jeden zbiór kluczy publicznego i prywatnego odpowiednio w plikach public.crs i private.crs. Kodując plik ASCII, na przykład fred.txt, uruchamiamy program crencode, który generuje losowy klucz sesyjny. I używa go do szyfrowania pliku. Ten klucz sesyjny jest po kolei szyfrowany przez klucz publiczny i przechowywany w pliku fred.key. Sam binarny , szyfrowany plik jest przechowywany jako fred.crs. Dla odkodowania pliku uruchamiamy program crdecode, który używa klucza prywatnego do odzyskania klucza sesyjnego, i od tej chwili dekoduje tekst na ekranie.

Parę punktów jest wartych zainteresowania. Przede wszystkim duża ilość kodowania jest przeprowadzana przy użyciu metody szyfru blokowego. Taki system hybrydowy jest praktyką standardową, ponieważ szyfr blokowy jest dużo szybszy niż metoda klucza publicznego. Schemat szyfru blokowego jest używany w nowym szyfrze blokowym Standardowego Zaawansowanego Szyfrowania, który jest implementowany w mraes.c

Przeanalizowanie kodu źródłowego crdecode.cpp , odsłoni ,ze deszyfrowanie jest procesem dwu przebiegowym. W pierwszym przebiegu program określa poprawność tekstu zaszyfrowanego, i tylko po to aby po sprawdzeniu ważności iść dalej w deszyfrowaniu pliku. Więc procedura deszyfrująca nie odpowiada wcale przypadkowemu ciągowi bitów spreparowanemu przez atakującego.

8.4.17 brick.cpp, ebrick.cpp, ebrick2.cpp

Protokoły Ustalonej Kryptografii wymagają podnoszenia do potęgi stałej liczby g, to jest obliczenia $g^x \mod n$, gdzie d\g i n są znane z wyprzedzeniem. W tym przypadku obliczenie może być znacznie przyspieszone przez wcześniejsze obliczenie, które generuje małą tablicę liczb big. Metoda at była najpierw opisana przez Brickell'a [Brick]. Przykładowy program brick.c ilustruje tą metodę. Odpowiednia krzyw eliptyczna GF(p) jest dostarczona w ebrick.c a $GF(2^m)$ w ebrick2.c

8.4.18 identity.c

Jest to program, który pozwala na pojedyncze wyjście z pewnymi tajnymi informacjami, zakładając wspólne klucze przez wykonanie obliczeń wymagających jedynie tylko korespondujących publicznie znanych identyfikatorów. Nie jest wymagana żadna wymiana danych [Maurer], i znane jest jako Nie Interakcyjna Wymiana Klucza. Zauważ, że 'publicznie znany identyfikator' może, na przykład ,być po prostu adresem email. Po pełny opis , sięgnij do [Scott92]. Ten przykładowy program generuje tajne dane z proponowanym Identyfikatorem. Jednakże przed uruchomieniem tego programu, dwukrotnie musi być uruchomiony program genprime.c, dla wygenerowania odpowiednich tylnych drzwi liczb pierwszych. Skopiuj dane wyjściowe programu prime.dat, najpierw do trap1.dat a potem do trap2.dat. Iloczyn tych liczb pierwszych będzie użyty jako moduł złożony używany w późniejszych obliczeniach.

8.5 Programy 'flash'

Kilka programów demonstruje użycie zmiennych flash. Jedne pokazują implementację eliminacji Gaussa do rozwiązania zbioru równań liniowych, wymagających znanych nie uwarunkowanych matryc Hilberta. Inne

pokazują jak arytmetyka wymierna może być użyta do przybliżeń arytmetyki rzeczywistej, w, na przykład, obliczeniu pierwiastków i π . Dalsze programy wykrywają błędy wwartościachdla pierwua\iasta kwadratowego z 5 danych w dodatku A Knutha[Knuth81]. Poprawną wartością jest

2.23606 79774 99789 69640 91736 68731 27623 54406
Błąd wystąpił tam gdzie mam 2 a nie 1.
Program roots działa szczególnie szybko kiedy oblicza pierwiastek kwadratowy liczby całkowitej o pjedynczej precyzji, ponieważ może być użyta prosta postać generatora ułamka. W jednym teście wzoercowy współczynnik (1+ √5) / 2 został obliczony do 100,000 miejsc dziesiętnych w 3 godziny na CPU na VAX11/780.

'Próbny' program został użyty do obliczenia π poprawnie do 1000 miejsc dziesiętnych, zabierając mniej niż minutę na IBM PC opartym na 80386 25MHz.

8.5.1roots.c

Program ten oblicza pierwiastek kwadratowy liczby wejściowej ,używając metody Newtona. Spróbuj użyć go do obliczenia pierwiastka kwadratowego z dwóch. Uzyskana dokładność zależy od rozmiaru zmiennych flash, określone w początkowym wywołaniu mirsys. Arytmetyka flash ma tendencję do preferowania prostych liczb, co może być zilustrowane przez prośbę, powiedzmy, o pierwiastek kwadratowy z 7. Program oblicza tą wartość a potem pierwiastkuje, dając ponownie dokładnie.

8.5.2 hilbert.c

Tradycyjnie inwersja matrycami 'Hilberta' jest stosowany do twardego testu dla każdego systemu arytmetycznego. Ten program rozwiązuje zbiór równań liniowych H.x = b, gdzie H jest matrycą Hilberta a b jest wektorem [1,1,1,1....1], używając klasycznej metody Eliminacji Gaussa

8.5.3 sample.c

Program ten jest taki sam jak używany przez Brenta [Brent78] do zademonstrowania potencjału jego pakietu arytmetyki Dużej precyzji Fortran. Oblicza π, exp (π.√163/9) i exp (π.√163).

8.5.4 ratcalc.c

Jako kompletną i użyteczną demonstracją arytmetyki flash jest ten program symulujący standardowy w pełni funkcjonalny kalkulator naukowy. Jego unikalną cechą (poza 36 cyfrową dokładnością) jest zdolność do bezpośredniego działania z ułamkami i działanie na mieszanych obliczeniach wymagających ułamków i liczb dziesiętnych. Poprzez zastosowanie tego programu użytkownik szybko uzyska wyczucie atyrmetyki flash i jej zdolności. Zauważ, że program ten zawiera nie przenośny kod (podprogramy obsługi ekranu), który musi być dopasowany do każdego pojedynczej kombinacji komputer / terminal. Ta wersja działa tylko na standardowych PC używających DOS'a, lub okna wiersza poleceń w Windows 'NT/'98.

9. PODPROGRAMY MIRACL

Notka: W tych podprogramach parametr big może być również użytym, gdziekolwiek jest określony flash, ale nie vice versa. Dalsze informacje mogą być zgromadzone ze skomentowanego kodu źródłowego. Gwiazdka * po nazwie wskazuje, że funkcja nie pobiera parametru mip jeśli MR_GENERIC_MT jest zdefiniowany w mirdef.h

9.1 PODPROGRAMY NISKIEGO POZIOMU

9.1.1 absol*

Funkcja:	void absol (x, y) flash x, y
Moduł:	mrcore.c
Opis:	Podaje wartość bezwzględną liczby big lub flash
Parametry:	Żadnych
Ograniczenia:	Żadnych

9.1.2 add

Funkcja:	void add (x, y ,z) big x ,y, z
Moduł:	mrarth0.c
Opis:	Dodaje dwie liczby big
Parametry:	Trzy liczby big x, y i z. Na wyjściu z = x+ y.
Wartość zwracana:	Żadnej
Ograniczenia:	Żadnych
Przykład:	add(x, x ,x); /* Podwaja liczbę x */

9.1.3 brand

Funkcja:	int brand
Moduł:	mrcore.c
Opis:	Generuje losową liczbę całkowitą
Parametry:	Żadnych
Wartość zwracana:	Losowa liczba całkowita
Ograniczenia:	Pierwsze użycie musi być poprzedzone początkowym wywołaniem irand. NOTKA: Ten generator nie jest mocny kryptograficznie. Dla aplikacji kryptograficznych zastosuj podprogram strong_rnd.

9.1.4 big_to_bytes

Funkcja:	int big_tobytes (max , x , ptr, justify) int max; big x; char *ptr; BOOL justify
Moduł	mrarth1.c
Opis:	Konwertuje dodatnią liczbę big x na binarny ciąg bajtowy
Parametry:	Liczba big x i tablica bajtowa ptr o długości max. Kontrola błędów jest realizowana przy założeniu, że funkcja nie zapisuje poza ograniczenie ptr jeśli max > 0. Jeśli max = 0 , żadne sprawdzanie nie jest przeprowadzane. Jeśli max > 0 a justify = TRUE, dane wyjściowe są wyrównane do prawej strony, w przeciwnym razie początkowe zera są usuwane.
Wartość zwracana:	Liczba bajtów generowana w ptr. Jeśli justify = TRUE, wtedy zwracaną wartością jest max.
Ograniczenia:	max musi być większe niż 0 jeśli justify = TRUE

9.1.5 bytes_to_big

Funkcja:	void bytes_to_big (len, ptr, x) int len; char *ptr; big x;
Moduł:	mrarth1.c
Opis:	Konwertuje binarny ciąg bajtowy na liczbę big. Konwersja binarna do big
Parametry:	Wskaźnik do tablicy bajtów ptr o długości len, a wynik big w x.
Zwracana wartość:	Żadnych
Ograniczenia:	Żadne
Przykład:	

```
/*
•     program  testowy  dla  ćwiczenia  big_to_bytes()  i  bytes_tobig()
*/
```

```
#include   <stdio.h>
#include   „miracl.h”

int  main()
{
     int  i,  len;
     miracl   *mip = mirsys (100,0);
     big  x, y;
             char    b[200];       /* b potrzebuje przestrzeni dla jej zaalokowania */
             x = mirvar(0);        /* wszystkie zmienne big muszą być „mirvar” */
y = mirvar (0);
     expint (2, 100,x);
     incr( x, 3, x);               /* x = 2^100 + 3 */

             len = big_to_bytes( 200, x, b, FALSE);
/* teraz b zawiera liczbę big x  jako surową liczbę binarną */ /*
ma len bajtów długości */
/* teraz wydrukujemy liczbę binarną b w hex */
     for (i=0; i< len; i++) printf („%02x”, b[i]);
     printf („\n”);

bytes_to_big (len,b, y);


/* teraz konwertujemy ją z powrotem do formatu big, i drukujemy ponownie */
     mip -> IOBASE=16;
     cotnum(y, stdout);

return  0;
}
```

9.1.6 cinnum

Funkcja:	int cinnum (x, f) flash x; FILE *f;
Moduł:	mrio2.c
Opis:	Wprowadza liczbę flash z klawiatury lub pliku, używając podstawy liczby bieżącej wartości instancji zmiennej IOBASE. Liczby flash mogą być wprowadzane przy użyciu albo ukośnika ‘/’ do wskazania licznika i mianownika lub kropki pozycyjnej.
Parametry:	Liczba big / flash x i deskryptor pliku f. Na wejściu z klawiatury określamy f jako stdin, w przeciwnym razie jako deskryptor jakiegoś innego otwartego pliku. Dla wymuszenia na wejściu stałej liczby bajtów, ustawiamy instancję zmiennej INPLEN na wymaganą liczbę, przed wywołaniem cinnum.
Wartość zwracana:	Liczba znaków wejściowych
Ograniczenia:	Żadnych
Przykład:	mip -> IOBASE=256; mip -> INPLEN=14; /* Ustawia to 14 bajtów z fp i */ cinnum (x, fp); /* konwertuje je do liczby big x */

9.1.7 cinstr

Funkcja:	int cinstr (x, s) flash x;

char *s;
Moduł: mrio2.c
Opis: Wprowadza liczbę flash z ciągu znaków używając jako podstawy liczbowej bieżącej wartości instancji zmiennej IOBASE. Liczby flash może być wprowadzona używając ukośnika '/' do wskazania licznika i mianownika lub kropki pozycyjnej.
Parametry: Liczba big / flash x i ciąg s
Wartość zwracana: Liczba wprowadzonych znaków
Ograniczenia: Żadnych
Przykład:

```
/* wprowadzenie dużej liczby hex do big x */
mip->IOBASE=16;
cinstr( x, „AF12398065BFE4c96DB723A");
```

9.1.8 compare*

Funkcja:

```
int compare (x, y)
big x, y;
```

Moduł: mrcore.c
Opis: Porównuje dwie liczby big
Parametry: Dwie liczby big, x i y
Wartość zwracana: Zwraca +1 jeśli x > y, zwraca 0 jeśli x =y, zwraca -1 jeśli x < y
Ograniczenia: Żadne

9.1.9 convert

Funkcja:

```
void convert (n,x)
int n;
big x;
```

Moduł: mrcore.c
Opis: Konwertuje liczbę całkowitą na liczbę formatu big
Parametry: Liczba całkowita n i liczba big n
Zwracana wartość: Żadna
Ograniczenia: Żadne

9.1.10 copy*

Funkcja:

```
void copy (x, y)
flash x, y
```

Moduł: mrcore.c
Opis: Kopiuje liczbę big lub flash do innej
Parametry: Dwie liczby big lub flash x i y. Na wyjściu y = x. Zauważ, że jeśli x i y są takimi samymi zmiennymi żadna operacja nie jest wykonana.
Zwracana wartość: Żadna
Ograniczenia: Żadne

9.1.11cotnum

Funkcja:

```
int cotnum (x, f)
flash x;
FILE *f;
```

Moduł mrio2.c
Opis: Wyprowadza liczbę big lub flash na ekran lub do pliku, używając jako podstawy liczby bieżącej wartości powiązanej z instancją zmiennej IOBASE. Liczba flash będzie skonwertowana do reprezentacji kropki pozycyjnej jeśli instancja zmiennej RPOINT=ON. W przeciwnym razie będzie wyprowadzona jako ułamek.

Parametry: Liczba big/flash x i deskryptor pliku f. Jeśli f jest stdout wtedy wyprowadzona będzie na ekran, w przeciwnym razie do pliku otwartego deskryptorem f.
Wartość zwracana: Liczba znaków wyprowadzonych.
Ograniczenia: Żadne
Przykład:
```
mip->IOBASE=16;
cotnum (x, fp);
```
Wyprowadza to x w hex do pliku powiązanego z fp

9.1.12 cotstr

Funkcja:
```
int cotstr (x, s)
flash x;
char   *s;
```
Moduł: mrio2.c
Opis: Wyprowadza liczbę big lub flash do określonego ciągu, używając jako podstawy liczbowej wartości obecnie przydzielonej do instancji zmiennej IOBASE. Liczba flash będzie skonwertowana do reprezentacji kropki pozycyjnej jeśli instancja zmiennej RPOINT=ON. W przeciwnym razie będzie wyprowadzona jako ułamek.
Parametry: Liczba big / flash x i ciąg s. Na wyjściu s będzie zawierało reprezentację liczby x
Zwracana wartość: Liczba wyprowadzonych znaków
Ograniczenia: Zauważ, że nic nie uniemożliwi temu podprogramowi przekroczenia ograniczeń użytkownika dostarczającemu tablicy znaków s, powodując niejasne problemy uruchomieniowe. Obowiązkiem programisty jest zapewnić, że s jest dosyć duże aby zawrzeć w sobie liczbę wyjściową. Alternatywnie użyjemy wewnętrznej deklaracji instancji ciągu IOBUFF, który jest rozmiaru IOBSIZ. Jeśli ta tablica się przepełni, będzie zasygnalizowany błąd MIRACL.

9.1.13 decr

Funkcja:
```
void decr (x, n, z)
big x, z;
int n;
```
Moduł: mrarth0.c
Opis: Zmniejszenie liczby big poprzez wielkość całkowitą
Parametry: Liczby big x i z, i liczba całkowita n.
Na wyjściu z = x - n
Zwracana wartość: Żadna
Ograniczenia : Żadnych

9.1.14 divide

Funkcja:
```
void  divide   (x, y, z)
big x, y, z;
```
Moduł: mrarth2.c
Opis: Dzieli jedną liczbę big przez inną
Parametry: Trzy liczby big x, y i z. Na wyjściu z=x/y; x=x mod y. Iloraz jest tylko zwracany jeśli x i z są takie same, reszta tylko jeśli y i z są takie same.
Zwracana wartość: Żadna
Ograniczenia: Parametry x i y muszą być a y nie może być zerem
Przykład:
```
divide (x,y,z);
```
Ustawia to x jako równe reszcie kiedy x jest dzielone przez y. Iloraz nie jest zwracany.

9.1.15 divisible

Funkcja:
```
BOOL  divisible (x, y)
```

	big x, y;
Moduł:	mrarth2.c
Opis:	testuje liczbę big na podzielność przez inna
Parametry:	Dwie liczby x i y
Zwracana wartość:	TRUE jeśli y dzieli się przez x dokładnie, w przeciwnym razie FALSE
Ograniczenia:	Parametr y musi być nie zerowy

9.1.16 exsign*

Funkcja:	int exsign (x) flash x;
Moduł:	mrcore.c
Opis:	Wyodrębnia znak z liczby big / flash
Parametry:	Liczba big/ flash x
Wartość zwracana;	znak z x, tj. -1 jeśli ujemna, +1 jeśli x jest zerem lub dodatnia
Ograniczenia:	Żadnych

9.1.17 getdig

Funkcja:	int getdig (x, i) big x; int i;
Moduł:	mrcore.c
Opis:	Wyodrębnia cyfrę z dużej liczby
Parametry:	Liczba big x i wymagana cyfra i
Zwracana wartość:	Wartość żądanej cyfry
Ograniczenia:	Zwraca bzdury jeśli żądana cyfra nie istnieje

9.1.18 get_mip

Funkcja:	miracl *get_mip (void)
Moduł:	mrcore.c
Opis:	Pobiera bieżący Miracl Instace Pointer
Parametry:	Żadnego
Zwracana wartość:	mip - Miracl Instance Pointer - dla bieżącego wątku
Ograniczenia:	Funkcja ta nie istnieje jeśli jest zdefiniowany MR_GENERIC_MT.

9.1.19 igcd*

Funkcja:	int igcd (x, y) Int x, y
Moduł:	mrcore.c
Opis:	Oblicza Największy Wspólny Dzielnik dwóch liczb całkowitych używając Metody Euklidesa
Parametry:	Dwie liczby całkowite x i y
Zwracana wartość:	NWD x i y

9.1.20 incr

Funkcja:	void incr (x, n, z) big x, z int n;
Moduł:	mrarth0.c
Opis:	Zwiększa zmienną big

Parametry:	Liczby big xi z i liczba całkowita n. Na wyjściu z = x+ n
Zawracana wartość:	Żadna
Ograniczenia:	Żadnych
Przykład:	incr (x, 2, x); /* To jest zwiększenie x przez 2 */

9.1.21 innum

Funkcja:	int innum (x, f) flash x; FILE *f;
Moduł:	mrio1.c
Opis:	Wprowadza liczby big lub flash z pliku lub klawiatury używając jako podstawy liczbowej wartości określonej w początkowym wywołaniu mirsys. Liczby flash mogą być wprowadzone używając albo ukośnika '/' do wskazania licznika i mianownika lub z kropką pozycyjną.
Parametry:	Liczba big / flash x i deskryptor pliku f. Na wejściu z klawiatury określamy f jako stdin, w przeciwnym razie jako deskryptor jakiegoś innego otartego pliku.
Zwracana wartość:	Liczba znaków wprowadzonych
Ograniczenia:	Podstawa liczbowa określona w mirsys musi być mniejsza niż lub równa 256. Jeśli nie użyjemy w zamian cinnum.
Wskazówka:	Dla szybszego wprowadzania tekstu ASCII do liczby big, i jeśli jest możliwa podstawa o pełnej szerokości, używamy mirsys(...256); pierwotnie. Ma to taki sam efekt jak określenia mirsys(..., 0); , z wyjątkiem tego, że teraz bajty ASCII mogą być wprowadzone bezpośrednio poprzez innum(x, fp); bez czasochłonnej zmiany bazy niejawnie przy użyciu cinnum.

9.1.22 insign*

Funkcja:	void insign (s, n) int s; flash x;
Moduł	mrcore.c
Opis:	Wymuszenie określonego znaku dla liczby big / flash
Parametry:	Liczba big / flash x i znak s, który jest do pobrania. Na wyjściu x = s./x|.
Zwracana wartość:	Żadna
Ograniczenia:	Żadne
Przykład:	insign (PLUS, x); /* wymuszenie x jako liczby dodatniej */

9.1.23 instr

Funkcja:	int instr (x, s) flash x; char *s;
Moduł:	mrio1.c
Opis:	Wprowadza liczbę big lub flash z ciągu znaków, używając jako podstawy liczbowej wartości określonej w początkowym wywołaniu mirsys. Liczby flash mogą być wprowadzone przy użyciu ukośnika '/' do wskazania licznika i mianownika, lub kropki pozycyjnej.
Parametry:	Liczba big / flash x i ciąg znaków s
Zwracana wartość:	Liczba znaków wejściowych
Ograniczenia:	Podstawa liczbowa okręslona w mirsys musi być mniejsza niż lub róna 256. Jeśli nie, używamy w zamian cinstr.

9.1.24 irand

Funkcja: void irand (seed)
long seed;
Moduł: mrcore.c
Opis: Inicjalizuje wewnętrzny system liczb losowych. Typ całkowity long jest używany wewnętrznie do dostarczania generatora z maksymalnym cyklem.
Parametry: Liczba całkowita long jest używana do działania generatora liczb losowych
Zwracana wartość: Żadna
Ograniczenia: Żadne

9.1.25 lgconv

Funkcja: void lgconv (ln, x)
long ln;
big x;
Moduł: mrcore.c
Opis: Konwertuje liczby całkowite long do formatu liczb big
Parametry: Liczba całkowita ln i liczba big x
Zwracana wartość: Żadna
Ograniczenia: Żadne

9.1.26 mad

Funkcja: void mad (x,y,z,w,q,r)
big x,y,z,w,q,r;
Moduł: mrarth2.c
Opis: Wielokrotne dodawanie i dzielenie liczb big. Początkowy iloczyn jest przechowywany w podwójnej długości wewnętrznej zmiennej unikając możliwości przepełnienia w tej fazie
Parametry: Sześć liczb big x,y,z,w,q i r. Na wyjściu q=(x.y + z)/ w a r zawiera resztę. Jeśli w i q nie są różnymi zmiennymi wtedy tylko reszta jest zwracana; jeśli q i r nie są różne wtedy tylko iloraz jest zwracany. Dodatkowo z nie jest dane jeśli x i z (lub y i z) są takie same
Zwracana wartość: Żadna
Ograniczenia: Parametry w i r muszą być różne. Wartość w musi nie by zerem
Przykład: mad (x,x,x,w,x,x); /* x=x^2/w */

9.1.27 memalloc

Funkcja: void *memalloc (n)
int n;
Moduł: mrcore.c
Opis: Rezerwuje przestrzeń dla n big zmiennych w jednym dostępie do sterty. Pojedyncze zmienne big / flash mogą być później zainicjalizowane z tej pamięci poprzez wywołanie mirvar_mem
Parametry: Liczba n zmiennych big / flash dla zarezerwowania przestrzeni
Zwracana wartość: Wskaźnik do zaalokowanej pamięci
Ograniczenia: Żadnych

9.128 memkill

Funkcja: void memkill (mem, n)
char *mem;
int n;
Moduł: mrcore.c
Opis: Usuwa i ustawia na zero pamięć uprzednio alokowaną przez memalloc
Parametry: Wskaźnik do pamięci kasowanej i usuwanej, i rozmiar tej pamięci w bigach
Zwracana wartość: Żadna

Ograniczenia: Musi być poprzedzona przez wywołanie memalloc

9.1.29 mirexit

Funkcja: void mirexit ()
Moduł: mrcore.c
Opis: Czyści po aktualnej instancji IRACL, i zwalnia wszystkie wewnętrzne zmienne. Późniejsze wywołanie mirsys będzie re -inicjalizował system MIRACL
Parametry: Żadnych
Zwracana wartość: Żadna
Ograniczenia: Musi być wywołany po mirsys

9.1.30 mirkill*

Funkcja: void mirkill (x)
big h;
Moduł: mrcore.c
Opis: Pewnie niszczy bezpowrotnie liczbę big / flash poprzez zerowanie i zwalnianie ich pamięci
Parametry: Liczba big / flash x
Zwracana wartość: Żadna

9.1.31 mirsys

Funkcja: miracl *mirsys (nd, nb)
int nd, nb;
Moduł: mrcore.c
Opis: Inicjalizuje system MIRACL dla bieżącego wątku programu, jak opisano poniżej. Musi być wywołany przed próbami użycia innych podprogramów MIRACL
(1) Jest inicjalizowany mechanizm śledzenia błędów
(2) Liczba słów komputerowych do zastosowania dla każdej liczby big i flash jest obliczana z nd i nb
(3) Szesnaście zmiennych big (cztery z nich o podwójnej długości jest inicjalizowanych
(4) Pewne instancje zmiennych są ustawiane wartościami domyślnymi
(5) Generator liczb losowych jest startowany poprzez wywołanie irand(0)
Parametry: Liczba cyfr nd jest używana dla każdej zmiennej big / flash a podstawa liczby nb. Jeśli nd jest ujemna, jest brana jako wskaźnik rozmiaru liczb big / flash w 8 bitowych bajtach
Zwracana wartość: Miracl Instance Pointer, poprzez wszystkie instancje zmiennych może być dostępny lub NULL jeśli nie było dość pamięci do stworzenia instancji
Ograniczenia: Podstawa liczby nb powinna zazwyczaj być większa niż 1 i mniejsza niż lub równa MAXBASE. Baza 0 sugeruje, że powinna być używana „maksymalna szerokość" podstawy liczby. Liczba cyfr nd musi być mniejsza niż pewne maksimum, w zależności od odpowiedniego typu mr_utype i czy lub mnie jest zdefiniowany MR_FLASH.
Przykład: miracl *mip = mirsys (500, 10);
To inicjalizuje system MIRACL przy użyciu 500 cyfr dziesiętnych dla każdej liczby big i flash.

9.1.32 mirvar

Funkcja: flash mirvar (iv)
int iv;
Moduł: mrcore.c

Opis: Inicjalizuje zmienne big / flash przez zarezerwowanie stosownej ilości komórek pamięci dla niej. Ta pamięć może być udostępniona przez późniejsze wywołanie funkcji mirkill
Parametry: Początkowa wartość całkowita dla liczby big / flash
Zwracana wartość: Wskaźnik do zarezerwowanej pamięci
Ograniczenia: Żadne
Przykład: flash x;
x = mirvar (8);
Tworzy zmienną flash x = 8

9.1.33 mirvar_mem

Funkcja: flash mirvar_mem (mem, index)
char *mem;
int index;
Moduł: mrcore.c
Opis: Alokuje pamięć dla zmiennej big / flash z pre - alokacją tablicy bajtowej mem. Tablica ta jest tworzona przez wywołanie memalloc. Używana wewnętrznie. Jest to szybsze niż wielokrotne wywołanie mirvar
Parametry: Wskaźnik do pre-alokacji tablicy mem, i indeks do tej tablicy. Każdy indeks powinien być unikalny
Zwracana wartość: Wskaźnik do zarezerwowanej pamięci
Ograniczenia: Musi być poprzedzony przez wywołanie memalloc
Przykład: Zobacz brent.c dla przykładowego zastosowania

9.1.34 multiply

Funkcja: void multiply (x, y, z)
big x, y, z;
Moduł: mrarth2.c
Opis: Mnoży dwie liczby big
Parametry: Trzy liczby big x,y i z. Na wyjściu z = x.y
Zwracana wartość: Żadna
Ograniczenia: Żadne

9.1.35 negify*

Funkcja: void negify (x, y)
flash x, y;
Moduł: mrcore.c
Opis: Neguje liczbę big / flash
Parametry: Dwie liczby big / flash x i y. Na wyjściu y = -x
Zwracana wartość: Żadna
Ograniczenia: Żadne. Zauważ, że negify (x, x) jest poprawne i ustawia x = -x

9.1.36 normalise

Funkcja: int normalise (x, y)
big x, y;
Moduł: mrarth2.c
Opis: Mnoży liczbę big, taką, której Najbardziej Znaczące Słowo jest większe niż połowa podstawy liczby. Jeśli jest używana taka liczba jako dzielnik przez divide, dzielenie wykonuje się szybciej. Jeśli jest wymagane wiele dzieleń przez ten sam dzielnik, ma sens normalizacja dzielnika z wyprzedzeniem.
Parametry: Dwie liczby big x i y. Na wyjściu y = n.x.

Zwracana wartość:	Zwraca n, znormalizowany mnożnik
Ograniczenia:	Używać ostrożnie. Używany wewnętrznie

9.1.37 numdig

Funkcja:	int numdig (x) big x;
Moduł:	mrcore.c
Opis:	Określa liczbę cyfr w liczbie big
Parametry:	Liczba big x
Zwracana wartość:	Liczba cyfr w x
Ograniczenia:	Żadne

9.1.38 otnum

Funkcja:	int otnum (x, f) flash x; FILE *f;
Moduł:	mrio1.c
Opis:	Wyprowadza liczbę big lub flash na ekran lub do pliku używając jako podstawy liczby wartości określonej w początkowym wywołaniu mirsys. Liczba flash będzie skonwertowana do reprezentacji kropki pozycyjnej jeśli instancja zmiennej RPOINT=ON. W przeciwnym razie będzie wyprowadzona jako ułamek.
Parametry:	Liczba big / flash x i deskryptor pliku f. Jeśli f jest stdout wtedy wyprowadzenie będzie na ekran, w przeciwnym razie do pliku otwartego deskryptorem f
Zwracana wartość:	Liczba wyprowadzonych znaków
Ograniczenia:	Podstawa liczbowa określona w mirsys musi być mniejsza niż lub równa 256. Jeśli nie, używamy cotnum

9.1.39 otstr

Funkcja:	int otstr (x,s) flash x; char *s;
Moduł:	mrio1.c
Opis:	Wyprowadza liczbę big lub flash doi określonego ciągu, używając jako podstawy liczbowej wartości określonej w początkowym wywołaniu mirsys. Liczba flash będzie skonwertowana do reprezentacji kropki pozycyjnej jeśli instancja zmiennej RPOINT=ON.W przeciwnym razie będzie wyprowadzona jako ułamek.
Parametry:	Liczba big / flash x i ciąg znaków s .Na wyjściu s będzie zawierać reprezentację x
Zwracana wartość:	Liczba wyprowadzonych znaków
Ograniczenia:	Podstawa liczbowa określona w mirsys musi być mniejsza niż lub równa 256. Jeśli nie używamy cotstr. Zauważ, że nic nie uniemożliwi temu podprogramowi przekroczenia ograniczeń użytkownika dostarczającemu tablicy znaków s, powodując niejasne problemy uruchomieniowe. Obowiązkiem programisty jest zapewnić, że s jest dosyć duże aby zawrzeć w sobie liczbę wyjściową. Alternatywnie użyjemy wewnętrznej deklaracji instancji ciągu IOBUFF, który jest rozmiaru IOSIZE. Jeśli ta tablica przepełni się, zostanie zasygnalizowany błąd MIRACL

9.1.40 premult

Funkcja:	void premult (x, n, z) int n; big x, z;

Moduł: mrarth1.c
Opis: Mnoży liczbę big przez liczbę całkowitą
Parametry: Dwie liczby big x iż i całkowita n
Na wyjściu z =n.x
Wartość zwracana: Żadna
Ograniczenia: Żadne

9.1.41 putdig

Funkcja: void putdig (n,x i)
big x;
int i, n;
Moduł: mrcore.c
Opis: Ustawia z liczby big na daną wartość
Parametry: Liczba big x, cyfra liczby i i jej nowa wartość n
Zwracana wartość: Żadna
Ograniczenia: Wskazywana cyfra musi istnieć

9.1.42 remain

Funkcja: int remain (x, n)
big x;
int n;
Moduł: mrarth1.c
Opis: Znajduje całkowitą resztę, kiedy liczba big jest dzielona przez liczbę całkowitą
Parametry: Liczba big x i całkowita n
Zwracana wartość: Całkowita reszta

9.1.43 set_io_buffer_size

Funkcja: void set_io_buffer_size (len)
int len;
Moduł: mrcore.c
Opis: Ustawia rozmiar bufora wejściowego / wyjściowego. Domyślnie jest ustawiony na 1024, ale programy, które muszą działać na bardzo dużych liczbach mogą wymagać większego bufora I/O
Parametry: Wymagany rozmiar bufora I/O
Zwracana wartość: Żadna
Ograniczenia: Niszczy aktualną zawartość bufora I/O

9.1.44 set_user_funkction

Funkcja: void set_user_function (func)
BOOL (*user) (void);
Moduł: mrcore.c
Opis: Dostarcza funkcji definiowanej przez użytkownika, która jest okresowo wywoływana podczas jakieś bardzo czasochłonnej funkcji MIRACL, szczególnie tych wywoływanych w modularnym potęgowaniu i znajdowaniu dużych liczb pierwszych, Dostarczona funkcja nie może pobierać parametrów i zwraca wartość BOOL. Zazwyczaj powinno to być TRUE. Jeśli FALSE wtedy MIRACL próbuje przerwać swoją bieżącą operację. W tym przypadku funkcja powinna kontynuować zwracać FALSE, dopóki sterownie nie jest zwrócone do programu wywołującego. Funkcja zdefiniowana przez użytkownika powinna normalnie zawierać tylko kilka instrukcji i żadnych pętli, w przeciwnym razie może niekorzystnie wpłynąć na szybkość funkcji MIRACL.

Ponieważ MIRACL jest inicjalizowana, funkcja ta może być wywołana wiele razy z nowymi dostarczonymi funkcjami. Jeśli nie jest dłużej potrzebna, wywołujemy ją z parametrem NULL.

Parametry: Wskaźnik do funkcji zdefiniowanej przez użytkownika lub NULL jeśli nie jest wymagana

Zwracana wartość: Żadna

Przykład:

```
/* Windows Message Pump */
static BOOL idle ()
{
        MSG msg;
        if  (PeekMessage  (&msg,  NULL,  0,  0,PM_NOREMOVE))
        {
                if  (msg.message  !=  WM_QUIT)
                {
                        if (PeekMessage (&msg, NULL,0,0,PM_REMOVE)) {/*
                        zrób Message Pump */
                                TranslateMessage (&msg);
                                DispatchMessage(&msg);
                        }
                }
                else  return  FALSE;
        }
        return TRUE;
}


set_user_function (idle);
```

9.1.45 size*

Funkcja:

```
int size (x)
big x;
```

Moduł: mrcore.c

Opis: Próbuje skonwertować liczbę big do prostej liczby całkowitej. Również użyteczna do testowania znaku zmiennej big / flash w : if (size (x) <0)...

Parametry: Liczba big x

Zwracana wartość: Wartość x jako liczba całkowita. Jeśli jest to ie możliwe (ponieważ x jest zbyt duże) zwraca wartość plus lub minus MR_TOOBIG

Ograniczenia: Żadne

9.1.46 subdiv

Funkcja:

```
int subdiv (x, n,z)
int n;
big x, z;
```

Moduł: mrarth1.c

Opis: Dzieli liczbę big przez liczbę całkowitą

Parametry: Dwie liczby big x i z i liczba całkowita n. Na wyjściu z = x/n

Zwracana wartość: Reszta całkowita

Ograniczenia: Wartość n nie może być zerem

9.1.47 subdivisible

Funkcja:

```
BOOL  subdivisible (x, n)
```

big x;
int n;

Moduł: mrarth1.c

Opis: Testuje liczbę big na podzielność przez liczbę całkowitą

Parametry: Liczba big x i całkowita n

Zwracana wartość: TRUE jeśli n dzieli się przez x, w przeciwnym razie FALSE

Ograniczenia: Wartość n nie może być zerem

9.1.48 subtract

Funkcja: void subtract (x,y,z)
big x,y,z;

Moduł: mrarth0.c

Opis: Odejmuje dwie liczby big

Parametry: Trzy liczby big x, y i z. Na wyjściu z = x - y

Zwracana wartość: Żadna

Ograniczenia: Żadne

9.1.49 zero*

Funkcja: void zero (x)
flash x;

Moduł: mrcore.c

Opis: Ustawia liczbę big lub flash na zero

Parametry: Liczba big lub flash x

Zwracana wartość: Żadna

Ograniczenia: Żadne

9.2 ZAAWANSOWANE PODPROGRAMY ARYTMETYCZNE

9.2.1 bigdig

Funkcja: void bigdig (n, b, x)
int n, b;
big x;

Moduł: mrrand.c

Opis: Generuje liczbę losową big o danej długości. Używa wbudowanego prostego generatora liczb losowych inicjalizowanego przez irand.

Parametry: Liczba big x i dwie liczby całkowite n i b. Na wyjściu x zawiera losową liczbę big n cyfrową o podstawie b

Zwracana wartość: Żadna

Ograniczenia: Baza b musi być drukowalna, to znaczy $2 \leq b \leq 256$

Przykład: bigdig (100, 10,x);
Generuje 100 cyfrową dziesiętną liczbę losową

9.2.2 bigrand

Funkcja: void bigrand (w, x)
big w, x;

Moduł: mrrand.c

Opis: Generuje losową liczbę big. Używa wbudowanego prostego generatora liczb losowych inicjalizowanego przez irand

Parametry: Dwie liczby ig w i x. Na wyjściu x jest liczbą losową big w zakresie $0 \leq x \leq w$

Zwracana wartość: Żadna

Ograniczenia: Żadne

9.2.3 brick_init

Funkcja:	BOOL brick_init (binst, g, n, nb) brick *binst; big g, n; int nb;
Moduł:	mrbrick.c
Opis:	inicjalizuje instancję Brickell'a wszystkich metod modularnego potęgowania z preobliczeniami. Jest alokowana pewna wewnętrzna przestrzeń robocza.
Parametry:	Wskaźnik do bieżącej instancji binst, stały generator g, moduł n i maksymalna liczba bitów używanych w wykładniku nb.
Zwracana wartość:	TRUE jeśli wszystko poszło dobrze, FALSE jeśli był problem
Ograniczenia:	Żadnych

9.2.4 brick_end*

Funkcja:	void brick_end (binst) brick *binst
Moduł:	mrbrick.c
Opis:	Porządkowanie po zastosowaniu metody Brickell'a
Parametry:	Wskaźnik do bieżącej instancji
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.2.5 crt

Funkcja:	void crt 9pbc, rem, x) big_chinese *pbc; big *rem; big x;
Moduł:	mrcrt.c
Opis:	Stosowanie Chińskiego Twierdzenia o Resztach
Parametry:	Wskaźnik pbc do bieżącej instancji. Na wyjściu x zawiera liczbę big , która przynosi daną resztę big rem[,] kiedy jest dzielona przez moduł big określony we wcześniejszym wywołaniu crt_init.
Zwracana wartość:	Żadna
Ograniczenia:	Najpierw musi być wywołany podprogram crt_init

9.2.6 crt_end*

Funkcja:	void crt_end (pbc) big _chinese *pbc;
Moduł:	mrcrt.c
Opis:	Porządkuje po zastosowaniu Chińskiego Twierdzenia o Resztach
Parametry:	Wskaźnik do bieżącej instancji Chińskeigo Twierdzenia o Resztach
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.2.7 crt_init

Funkcja:	BOOL crt_init (pbc, np.,m.) big_chinese *pbc; int np.; big *m.;
Moduł:	mrcrt.c
Opis:	Inicjalizuje instancję Chińskiej Teorii o Resztach. Jest alokowana jakaś wewnętrzna przestrzeń robocza.
Parametry:	Wskaźnik do bieżącej instancji pbc, liczba modułów ko-pierwszych np. i tablica przynajmniej dwóch modułów big m.[.]
Zwracana wartość;	TRUE jeśli wszystko idzie dobrze, FALSE jeśli był jakiś problem.

Ograniczenia: Żadne

9.2.8 egcd

Funkcja: int egcd (x,y,z)
big x, y ,z ;
Moduł: mrgcd..c
Opis: Oblicza Największy Wspólny Dzielnik dwóch liczb big
Parametry: Trzy liczby big x, y i z. Na wyjściu z = gcd(x,y)
Zwracana wartość: NWD jako liczba całkowita ,jeśli możliwe, w przeciwnym razie MR_TOOBIG
Ograniczenia: Żadne

9.2.9 expint

Funkcja: void expint (b, n,x)
int b,n;
big x;
Moduł: mrarth3.c
Opis: Oblicza całkowitą potęgę liczby całkowitej jako big
Parametry: Liczba całkowita b, n i wynik big x. Na wyjściu $x = b^n$.
Zwracana wartość: Żadna
Ograniczenia: Żadne
Przykład: expint (2, 1398269,x);
decr (x,1,x);
mip -> IOBASE=10;
cotnum(x, stdout);
Oblicza i drukuje największą znaną liczbę pierwszą (na prawdziwym 32 bitowym komputerze z dużą ilością pamięci!)

9.2.10 fft_mult

Funkcja: void fft_mult (x, y ,z)
big x,y,z;
Moduł: mrfast.c
Opis: Mnoży dwie liczby big używając szybkiej metody Fouriera
Parametry: Trzy liczby big x,y i z. Na wyjściu z = x.y
Zwracana wartość: Żadna
Ograniczenia: Powinien być używany tylko na 32 bitowych komputerach kiedy x i y są bardzo duże, przynajmniej 1000 cyfr dziesiętnych
Przykład: Zobacz mersenne.c

9.2.11 gprime

Funkcja: void gprime (n)
int n;
Moduł: mrprime.c
Opis: Generuje wszystkie liczby pierwsze do pewnego limitu w instancji tablicy PRIMES, zakończonej zerem. Tablica ta jest używana wewnętrznie przez podprogramy isprime i nxprime.
Parametry: Całkowita liczba dodatnia n wskazująca maksymalną liczbę pierwszą mającą być wygenerowaną. Jeśli n =0 tablica PRIMES jest usuwana
Zwracana wartość: Żadna
Ograniczenia: Żadne

9.2.12 invers*

Funkcja: unsigned int invers (x,y);
unsigned int x, y;
Moduł: mrsmall.c
Opis: Oblicza odwrotność modułu całkowitego całkowitej liczby ko-pierwszej
Parametry: Liczba całkowita x i ko -pierwsza liczba całkowita y

Zwracana wartość: x^{-1} mod y
Ograniczenia: Wynik nieprzewidywalny jeśli x i y są nie ko - pierwsze

9.2.13 isprime

Funkcja: BOOL isprime (x)
big x;
Moduł: mrprime.c
Opis: Testuje czy lub nie liczba big jest pierwsza przy użyciu probabilistycznego testu pierwszości. Liczba jest z założenia pierwsza jeśli przekazuje ten test NTRY razy, gdzie NTRY jest instancją zmiennej z domyślną inicjalizacją w podprogramie mirsys

NOTKA: Podprogram ten najpierw testuje dzielenie x przez listę małych liczb pierwszych przechowywanych w instancji tablicy PRIMES. Testowanie dużych liczb pierwszych będzie znacznie szybsze w wielu przypadkach jeśli ta lista jest zwiększana. Zobacz gprime. Domyślnie tylko małe liczby pierwsze mniejsze niż 1000 są używane.
Parametry: Liczba big x
Zwracana wartość: Zwraca wartość boolowską TRUE jeśli x jest (prawie na pewno) liczbą pierwszą, w przeciwnym razie FALSE
Ograniczenia: Żadne

9.2.14 jac

Funkcja: int jac (x, n)
unsigned int x, n;
Moduł: mrsmall.c
Opis: Oblicza wartość symbolu Jacobi'ego.
Parametry: Dwie liczby unsigned x i n
Zwracana wartość: Wartość (x/n) jako +1 lub -1, lub 0 jeśli symbol niezdefiniowany
Ograniczenia: Żadne

9.2.15 jack

Funkcja: int jack (x, n)
big x, n;
Moduł: mrjack.c
Opis: Oblicza wartość symbolu Jacobi'ego
Parametry: Dwie liczby big x i n
Zwracana wartość: Wartość (x/n) jako +1 lub -1, lub 0 jeśli symbol niezdefiniowoany
Ograniczenia: Żadne

9.2.16 logb2

Funkcja: int logb2 (x)
big x;
Moduł: mrarth3.c
Opis: Oblicza przybliżony całkowity log o podstawie 2 liczby big (faktycznie liczba bitów w nim.)
Parametry: Liczba big x
Zwracana wartość: Liczba bitów w x
Ograniczenia: Żadne

9.2.17 lucas

Funkcja: void lucas (x,e,n,vp,v)
big x,e,n,vp,v
Moduł: mrlucas.c
Opis: Wykonuje modularne potęgowanie Lucas'a. Używa wewnętrznie arytmetyki Mongomery'ego. Funkcja ta może być dalej przyspieszona dla poszczególnych

	modułów, poprzez wywołanie specjalnego podprogramu języka asemblera do implementacji arytmetyki Montgomery'ego. Zobacz powmod
Parametry:	Pięć liczb big, x,e,n, vp i v. Na wyjściu $v = V_e(x)$ mod n i $vp = V_{e-1}(x)$ mod n gdzie n jest bieżącym modułem Mongomery'ego. Tylko v jest zwracane jeśli jeśli v i vp nie są różne
Zwracana wartość:	Żadna
Notka:	„Siostrzana" funkcja Lucasa Ue(x) może, jeśli jest to wymagane, być obliczona jako

$$U_e(x)=[x.V_e(x)-2.V_{e-1}(x)]/(x^2-4) \bmod n$$

9.2.18 multi_inverse

Funkcja:	BOOL multi_inverse (m., x n, w) int m.; big n; big *x, *w;
Moduł:	mrxgdc
Opis:	Znajduje modularną inwersję wielu liczb równocześnie, wyzyskując obserwację Mongomery'ego, że $x^{-1} = y.(xy)^{-1}$, $y^{-1} =x.(xy)^{-1}$. Będzie to szybsze, ponieważ modularne inwersje są wolne do obliczenia, a ten sposób jest tylko wymagany.
Parametry:	Liczba wymaganych inwersji m., tablica x[.] m. liczb których inwersja jest oczekiwana, moduł n i tablica wynikowa inwersji w[.].
Zwracana wartość:	TRUE jeśli pomyślnie, w przeciwnym razie FALSE
Ograniczenia:	Parametry x i w muszą być różne

9.2.19 nres

Funkcja:	void nres (x, y) big x, y;
Moduł:	mrmonty.c
Opis:	konwertuje liczbę big do postaci n-resztkowej
Parametry:	Dwie liczby big xi y. Na wyjściu y jest n-resztkową postacią x
Wartość zwracana:	Żadna
Ograniczenia:	Musi być poprzedzona wywołaniem prepare_monty

9.2.20 nres_dotprod

Funkcja:	void nres_dotprod (m.,x,y,w) int m.; big x[]. Y[], w;
Moduł:	mrmonty.c
Opis:	Znajduje iloczyn skalarny dwóch tablic n-resztkowych. Jest używana tak zwana „leniwa" redukcja, w tym suma iloczynu jest tylko redukowana w związku z modułem Montgomery'ego. Jest to szybsze - prawie dwukrotnie tak szybkie.
Parametry:	Dwie tablice x i y każda m. n-resztkowa. Na wyjściu $w = \Sigma x_i y_i$ mod n , gdzie n jest aktualnym modułem Montgomery'ego.
Zwracana wartość:	Żadna
Ograniczenia:	Musi być poprzedzony wywołaniem prepare_monty

9.2.21 nres_lucas

Funkcja:	void nres_lucas (x,e,vp,v) big x,e,vp,v;
Moduł:	mrlucas.c
Opis:	Modularne potęgowanie Lucasa n-resztkowe
Parametry:	n-resztkowe x, wykłądnik big e i dwa n-resztkowe wyjściwe vp i v. Na wyjściu v $=V_e(x)$ mod n i $vp = V_{e-1}(x)$ mod n gdzie n jest bieżącym modułem Montgomery'ego. Tylko v jest zwracane jeśli v i vp są takimi samymi zmiennymi big.
Zwracana wartość:	Żadna

Ograniczenia:	Musi być poprzedzony wywołaniem prepare_monty i konwersją pierwszego parametru do postaci n-resztkowej. Zauważ, że wykłądnik nie jest konwertowany do postaci n-resztkowej.

9.2.22 nres_moodadd

Funkcja:	void nres_modadd (x,y,z) big x,y,z;
Moduł:	mrmonty.c
Opis:	Modularne dodawanie dwóch liczb n-resztkowych
Parametry:	Trzy n-resztkowe liczby x, y i z. Na wyjściu z = x+y mod n, gdzie n jest bieżącym modułem Mongomery'ego
Zwracana wartość:	Żadna
Ograniczenia:	Musi być poprzedzona przez wywołanie prepare_monty

9.2.23 nres_moddiv

Funkcja:	int nres_moddiv (x,y,z) big x,y,z;
Moduł:	mrmonty.c
Opis:	Modularne dzielenie dwóch liczb n-resztkowych
Parametry:	Trzy liczby n-resztkowe x,y i z. Na wyjściu z =x/y mod n, gdzie n jest aktualnym modułem Montgomery'ego
Zwracana wartość:	NWD y i n jako liczba całkowita, jeśli to możliwe, lub MR_TOOBIG. Powinno być 1 dla poprawnego wyniku.
Ograniczenia:	Musi być poprzedzony przez wywołanie prepare_monty i skonwertować parametrów do postaci n-resztkowej. Parametry x i y muszą być różne

9.2.24 nres_modmult

Funkcja:	void nres_modmult (x,y,z) big x,y,z;
Moduł:	mrmonty.c
Opis:	Modularne mnożenie dwóch liczb n-resztkowych. Zauważ, ze ten podprogram będzie wywoływał Modularne Mnożenie KCM jeśli MR_KCM została zdefiniowana w mirdef.h i ustawia właściwy rozmiar dla bieżącego modułu, lub stałego rozmiaru modularne mnożenie Comba jeśli MR_COMBA jest zdefiniowane jako dokładny rozmiar modułu
Parametry:	Trzy liczby n-resztkowe x,y i z. Na wyjściu z= xy mod n, gdzie n jest bieżącym modułem Montgomery'ego
Zwracana wartość:	Żadna
Ograniczenia:	Musi być poprzedzony przez wywołanie prepare_monty i skonwertować parametry do postaci n-resztkowej

9.2.25 nres_modsub

Funkcja:	void nres_modsub (x,y,z) big x, y, z;
Moduł:	mrmonty.c
Opis:	Modularne odejmowanie dwóch liczb n-resztkowych
Parametry:	Trzy liczby n-resztkowe x,y i z. Na wyjściu z = x-y mod n, gdzie n jest bieżącym modułem Mongomery'ego
Zwracana wartość:	Żadna
Ograniczenie:	Musi być poprzedzony przrez prepare_monty

9.2.26 nres_multi_inverse

Funkcja:	BOOL nres_multi_inverse (m.,x,w) int m.; big *x *w;

Moduł:	mrmonty.c
Opis:	Znajduje modularną inwersję wielu liczb równocześnie, wykorzystując obserwację Mongomery'ego, że $x^{-1} = y.(xy)^{-1}$, $y^{-1} = x.(xy)^{-1}$. Będzie to szybsze, ponieważ modularne inwersje są wolne do obliczeń, a to jedyny sposób jaki jest wymagany.
Parametry:	Wymagana liczba do inwersji m., tablica x[.] m. liczb n-resztkowych których inwersja jest oczekiwana i tablica ich inewersji w[.].
Zwracana wartość:	TRUE jeśli pomyślnie, w przeciwnym razie FALSE
Ograniczenia:	Parametry x i w muszą się różnić

9.2.27 nres_negate

Funkcja:	void nres_negate (x,w) big x,w;
Moduł:	mrmonty.c
Opis:	Modularna negacja
Parametry:	Dwie liczby n-resztkowe x i w. Na wyjściu w = -x mod n, gdzie n jest bieżącym modułem Montgomery'ego
Zwracana wartość:	Żadna
Ograniczenie:	Musi być poprzedzona przez wywołanie prepare_monty

9.2.28 nres_powltr

Funkcja:	void powltr (x,e,w) int x; big e,w;
Moduł:	mrpower.c
Opis:	Modularne potęgowanie liczb n-resztkowych
Parametry:	Zwykłą mała liczba całkowita x, liczba big x e i n resztkowy wynik w. Na wyjściu w $=x^e$ mod n, gdzie n jest bieżącym modułem Mongomery'ego
Zwracana wartość:	Żadna
Ograniczenia:	Musi być poprzedzona przez wywołanie prepare_monty. Zauważ, że mała liczba całkowita x i wykładnik nie są konwertowane do postaci n-resztkowe.

9.2.29 nres_powmod

Funkcja:	void nres_powmod (x,y,z) big x,y,z;
Moduł:	mrpower.c
Opis:	Modularne potęgowanie liczb n-resztkowych
Parametry:	N-resztkowa liczba x, liczba big y i n-resztkowy wynik z. Na wyjściu $z = x^y$ mod n, gdzie n jest bieżącym modułem Mongomery'ego
Zwracana wartość:	Żadna
Ograniczenie:	Musi być poprzedzony wywołaniem prepare_monty i konwersją pierwszego parametru do postaci n-resztkowej. Zauważ, że wykładnik nie jest konwertowany do postaci n-resztkowej.
Przykład:	prepare_monty (n); nres (x,y); /* konwersja do postaci n-resztkowej */ nres_powmod (y,e,z); redc (z,w); /* konwersja powrotna do zwykłej postaci */

9.2.30 nres_powmod2

Funkcja:	void nres_powmod2 (x,y,a,b,w) big x,y,a,b,w;
Moduł:	mrpower.c
Opis:	Oblicza iloczyn dwóch modularnych potęgowań wymagających liczb n-resztkowych
Parametry:	Trzy n-resztkowe liczby x,a i w, i dwie liczby całkowite big y i b. Na wyjściu w $=x^y.a^b$ mod n, gdzie n jest bieżącym modułem Montgomery'ego.
Zwracana wartość:	Żadna

Ograniczenia:	Musi być poprzedzony przez wywołanie prepare_monty i konwersję włąściwych parametrów do postaci n-resztkowej. Zauważ, że wykładniki nie są konwertowane do postaci n-resztkowej.

9.2.31 nres_powmodn

Funkcja:	void mres_powmodn (m.,x,y,w) int m.; big *x, *y ,w;
Moduł:	mrpower.c
Opis:	Oblicza iloczyn m. modularnych potęgowań wymagających liczb n-resztkowych. Dodatkowa pamięć jest alokowana wewnętrznie przez tą funkcję.
Parametry:	Liczba całkowita m., tablica m. n-resztkowych liczb x, tablica m. liczb całkowitych big y i n-resztkowe w. Na wyjściu w = $x[0]^{y[0]}$. $x[1]^{y[1]}$ ....$x[m.-1]_{y[m.-1]}$ mod n, gdzie n jest bieżącym modułem Mongomery'ego.
Zwracana wartość;	Żadna
Ograniczenia:	Musi być poprzedzony przez wywołanie prepare_monty i konwersję właściwych parametrów do postaci n-resztkowej. Zauważ, ze wykładniki nie są konwertowane Do postaci n-resztkowej.

9.2.32 nres_premult

Funkcja:	void nres_premult (x,k,w) int k; big x, w;
Moduł:	mrmonty.c
Opis:	Mnożenie liczby n-resztkowej przez małą liczbę całkowitą
Parametry:	Dwie n-resztkwoe liczby x i w, i mała liczba całkowita k. Na wyjściu w= kx mod n, gdzie n jest bieżącym modułem Montgomery'ego.
Zwracana wartość;	Żadna
Ograniczenia:	Musi być poprzedzona przez wywołanie prepare_monty i konwersją pierwsego parametru do postaci n-resztkowej. Zauważ, ze małą liczba całkowita nie jest konwertowana do postaci n-resztkowej

9.2.33 nres_sqroot

Funkcja:	BOOL nres_sqroot (x,w) big x,w;
Moduł:	mrpower.c
Opis:	Oblicza pierwiastek kwadratowy z liczby n-resztkowej mod moduł liczb pierwszych
Parametry:	Dwie liczby n-resztkwoe x i w. Na wyjściu w =$\sqrt{x}$ mod n, gdzie n jest bieżącym modułem Mongomery'ego
Zwracana wartość:	TRUE jeśli pierwiastek kwadratowy istnieje , w przeciwnym razie FALSE
Ograniczenia:	Musi być poprzedzony przez wywołanie prepare_monty i konwersję pierwszego parametru na postać n-resztkową

9.2.34 nroot

Funkcja:	BOOL nroot (x,n,z) big x,z; int n;
Moduł:	mrarth3.c
Opis:	Wyodrębnia największe przybliżenie pierwiastka z liczby big
Parametry:	Dwie liczby big x i z, i liczba całkowita n. Na wyjściu z =$[x^{1/n}]$
Zwracana wartość:	Zwraca boolowską wartość TRUE jeśli pierwiastek znaleziony jest dokładny, w przeciwnym razie FALSE.
Ograniczenia:	Wartość n musi być dodatnia. Jeśli x jest ujemne, wtedy n musi być nieparzyste.

9.2.35 nxprime

Funkcja: BOOL nxprime (w,x)
big w,x;
Moduł: mrprime.c
Opis: Znajduje następną liczbę pierwszą
Parametry: Dwie liczby big w ix. Na wyjściu x zawiera następną liczbę pierwszą większą niż w.
Zwracana wartość: TRUE jeśli pomyślnie, FALSE w przeciwnym razie
Ograniczenia: Żadnych

9.2.36 nxsafeprime

Funkcja: BOOL nxsafepriome (type, subset,w, p)
int type, subset;
big w, p;
Moduł: mrprime.c
Opis: Znajduje następną bezpieczną liczbę pierwszą większą niż w. Bezpieczna liczba pierwsza p jest zdefiniowana tak, że, q=(p-1)/ t (type = 0) lub q=(p+1)/2 (type =1) jest również liczbą pierwszą.
Parametry: Całkowity parametr type określa typ bezpiecznej liczby pierwszej jak powyżej. Jeśli parametr subset = 2, wtedy poszukiwanie jest ograniczane aby wartość liczby pierwszej q była zgodna do 1 mod 4. Jeśli subset = 3 wtedy poszukiwanie jest ograniczone aby wartość q byłą zgodna z 3 mod 4. Jeśli subset = 0 wtedy nie ma warunku dla q: może być albo 1 albo 3 mod 4
Zwracana wartość: TRUE jeśli pomyślnie, FALSE w przeciwnym razie

9.2.37 pow_brick

Funkcja: void pow_brick (binst, e, w)
brick *binst;
big e,w;
Moduł: mrbrick.c
Opis: Przeprowadza modularne potęgowanie, używając pre -obliczonych wartości przechowanych w strukturze brick.
Parametry: Wskaźnik do bieżącej instancji, wykładnik big e i liczba big w. Na wyjściu w $=g^e$ mod n, gdzie n jest określone w początkowym wywołaniu brick_init.
Zwracana wartość; Żadna
Ograniczenia: Musi być poprzedzony wywołany przez wywołanie brick_init

9.2.38 power

Funkcja: void power (x,n,z,w)
long n;
big x,z,w;
Moduł: mrarth3.c
Opis: Podnosi liczbę big do potęgi całkowitej
Parametry: Dwie liczby big x i z i liczba całkowita n. Na wyjściu w = x^n. Jeśli w i z są różne, wtedy w = x^n mod z
Zwracana wartość: Żadna
Ograniczenia: Wartość n musi być dodatnia

9.2.39 powltr

Funkcja: int powltr (x,y,z,w)
int x;
big y,z,w;
Moduł: mrpower.c
Opis: Podnosi int do potęgi modułu liczby big innej liczby big. Używa metody binarnej Lewy - do - Prawego i będzie noieco szybsza niż powmod dla małego x. Używamy arytmetyki Montgomery'ego wewnętrznie jeśli moduł z jest nieparzysty.
Parametry: Liczba całkowita x i trzy liczby big y,z i w. Na wyjściu w $=x^y$ mod z

Zwracana wartość:	Wynik wyrażony jako liczba całkowita, jeśli możliwe. W przeciwnym razie wartość MR_TOOBIG.
Ograniczenia:	Wartość y musi być dodatnia. Parametry w iż muszą się różnić.

9.2.40 powmod

Funkcja:	void powmod (x,y,z,w) big x,y,w,z;
Moduł:	mrpower.c
Opis:	Podnosi liczbę big do potęgi modułu big innej liczby big. Używa wyrafinowanej 5 bitowej techniki okna rozsuwanego, która jest bliska optimum dla popularnego rozmiaru modułu(taki jak 512 lun 1024 bitów). Używa wewnętrznie arytmetyki Montgomery'ego jeśli moduł z jest nieparzysty. Funkcja ta może być przyspieszona dalej dla poszczególnych modułów, przez wywołanie specjalnego podprogramu asemblerowego (jeśli Twój kompilator na to pozwala). Modularny Mnożnik KCM będzie automatycznie wywołany jeśli MR_KCM będzie zdefiniowany w mirdef.h i będzie ustawiony na właściwy rozmiar. Alternatywnie modularny mnożnik Comba będzie użyty jeśli MR_COMBA jest również zdefiniowany a moduł jest określonego rozmiaru. Eksperymentalnie kod koprocesora będzie wywołany jeśli MR_PENTIUM jest zdefiniowany. Tylko jeden z tych warunków powinien być zdefiniowany.
Parametry:	Cztery liczby big x,y,z i w. Na wyjściu $w = x^y$ mod z
Zwracana wartość:	Żadna
Ograniczenia:	Wartość y musi być dodatnia. Parametry w i z muszą być różne

9.2.41 powmod2

Funkcja:	void powmod2 (a,b,c,d,z,w) big a,b,c,d,z,w;
Moduł:	mrpower.c
Opis:	Oblicza iloczyn dwóch modularnych potęgowań. Jest to szybsze niż robienie dwóch oddzielnych potęgowań i jest użyteczne dla pewnych protokołów Kryptograficznych. Używa 2 bitowego przesuwanego okna.
Parametry:	Sześć liczb big a,b,c,d,z i w. Na wyjściu $w = a^b.c^d$ mod z
Zwracana wartość:	Żadna
Ograniczenia:	Wartości b i d muszą być dodatnie. Parametry w i z muszą być różne. Moduł z musi być nieparzysty
Przykład:	Zobacz dssver.c

9.2.42 powmodn

Funkcja:	void powmod (m.,a,b,z,w) int m.; big a,b,z,w;
Moduł:	mrpower.c
Opis:	Oblicza iloczyn m. modularnych potęgowań. Jest to szybsze niż robienie m. oddzielnych potęgowań, i jest to użyteczne dla pewnych protokołów Kryptograficznych. Dodatkowa pamięć jest alokowana wewnętrznie dla tej funkcji.
Parametry:	Liczba całkowita m., dwie tablice liczb big a[] i b[], i dwie liczby big z i w. Na wyjściu $w = a[0]^{b[0]}.a[1]^{b[1]}.... .a[m.-1]_{b[m.-1]}$ mod z
Zwracana wartość:	Żadna
Ograniczenia:	Wartość b[] musi być dodatnia. Parametry w i z muszą być różne. Moduł z musi być nieparzysty. Odpowiednia podstawa liczby musi być potęgą 2.

9.2.43 prepare_monty

Funkcja:	void prepare_monty (n) big n;

Moduł:	mrmonty.c
Opis:	Przygotowuje Moduł Montgomery'ego do użycia. Każde wywołanie tej funkcji zamienia miejscami poprzednie moduły (jeśli trzeba)
Parametry:	Liczba big n, która jest modułem Montgomery'ego
Ograniczenia:	Parametr n musi być dodatni i nieparzysty. Alokowana pamięć jest zwalniana, kiedy bieżąca instancja MIRACL jest zakończona przez wywołanie mirexit

9.2.44 redc

Funkcja:	void redc (x,y) big x,y;
Moduł:	mrmonty.c
Opis:	Konwertuje liczbę n-resztkową z powrotem do normalnej postaci
Parametry:	Dwie liczby big x i y. Na wyjściu y jest normalną postacią n-resztkowekj liczby x
Zwracana wartość:	Żadna
Ograniczenia:	Użycie musi być poprzedzone wywołaniem prepare_monty

9.2.45 scrt

Funkcja:	void scrt (psp, rem, x) small_chinese *psc; int *rem; big x;
Moduł:	mrscrt.
Opis:	Stosuje Chińskie Twierdzenie o Resztach (dla małych modułów liczb pierwszych)
Parametry:	Wskaźnik psc do bieżącej instancji Chińskiego Twierdzenia o Resztach. Na wyjściu x zawiera liczbę big, która daje daną resztę rem[.] kiedy jest dzielona przez całkowity moduł określony we wcześniejszym wywołaniu scrt_init.
Zwracana wartość:	Żadna
Ograniczenia:	Podprogram scrt_init musi być wywołany najpierw

9.2.46 scrt_end*

Funkcja:	void scrt_end (psc) small_chinese *psc;
Moduł:	mrscrt.c
Opis:	Porządkuje po zastosowaniu ChińskiegoTwierdzenia o Resztach
Parametry:	Wskaźnik do bieżącej instancji Chińskeigo Twierdzenia o Resztach
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.2.47 scrt_init

Funkcja:	BOOL scrt_init (psc, np., m.) small_chinese *psc; int np.; int *m.;
Moduł:	mrscrt.c
Opis:	Inicjalizuje instancję Chińskiego Twierdzenia o Resztach. Jest alokowana wewnętrznie pewna przestrzeń robocza
Parametry:	Wskaźnik do bieżącej instancji psc. Liczba ko -pierwszych modułów np. i tablica przynajmniej dwóch całkowitych modułów m.[.]
Zwracana wartość:	TRUE jeśli wszystko poszło dobrze, FALSE jeśli wystąpił problem.
Ograniczenia:	Żadne

9.2.47 sftbit

Funkcja:	void sftbit (x,n,z) big x,z; int n;

Moduł: nrarth3.c
Opis: Przesuwa liczbę całkowitą big w lewo lub prawo o liczbę bitów
Parametry: Parametr big x jest przesuwany o n bitów danych w z, Dodatnie n przesuwa w lewo, ujemne w prawo
Zwracana wartość: Żadna
Ograniczenia: Żadne

9.2.49 smul*

Funkcja: unsigned int smul (x,y,z)
unsigned int x,y,z;
Moduł: mrsmall.c
Opis: Mnoży dwie liczby całkowite mod z trzecią
Parametry: Liczby całkowite x,y i z
Zwracana wartość: x,y mod z

9.2.50 spmd*

Funkcja: unsigned int spmd (x,y,z)
unsigned int x,y,z;
Moduł: mrsmall.c
Opis: podnosi liczbę całkowitą do całkowitej potęgi trzeciego modułu
Parametry: Liczby całkowite x,y i z
Zwracana wartość: x^y mod z
Ograniczenia: Żadne

9.2.51 sqrmp*

Funkcja: unsigned int sqrmp (x,p)
unsigned int x, p;
Moduł: mrsmall.c
Opis: Oblicza pierwiastek kwadratowy liczby całkowitej mod całkowita liczba pierwsza
Parametry: Liczba całkowita x i liczba pierwsza p
Zwracana wartość: $\sqrt{x}$ mod p, lub 0 jeśli pierwiastek nie istnieje
Ograniczenia: Zwracana wartość jest nieprzewidywalna, jeśli p nie jest liczbą pierwszą

9.2.52 sqroot

Funkcja: BOOL sqroot (x,p,w)
big x,p;
Moduł: mrpower.c
Opis: Oblicza pierwiastek kwadratowy liczby całkowitej big mod liczba całkowita pierwsza big
Parametry: Dwie liczby całkowite x i w, i liczba pierwsza big p. Na wyjściu $w =\sqrt{x}$ mod p jeśli pierwiastek kwadratowy istnieje, w przeciwnym razie w =0. Zauważ, ze „inny” pierwiastek kwadratowy może być znaleziony poprzez odjęcie w od p.
Zwracana wartość: TRUE jeśli istnieje pierwiastek kwadratowy, w przeciwnym razie FALSE
Ograniczenia: Liczba p musi być pierwszą. Zauważ, że ten podprogram jest szczególnie wydajne jeśli p = 3 mod 4

9.2.53 trial_division

Funkcja: int trial_division (x,y)
big x,y;
Moduł: mrprime.c
Opis: Dwufunkcyjny podprogram próbnego dzielenia.. Jeśli x i y są takimi samymi zmiennymi big, wtedy próbne dzielenie przez małe liczby pierwsze w instancji tablicy PRIMES jest próbą określenia statusu pierwszości liczby big. Jeśli x i y są różne, wtedy, po próbnym dzieleniu, część x jest zwracana w y
Parametry: Dwie liczby big x i y

Zwracana wartość: Jeśli x i y są takie same, wtedy zwracana wartość 0 oznacza, że liczba big nie jest zdecydowanie liczbą pierwszą, wartość zwracana 1 oznacza, że jest zdecydowanie liczbą pierwszą, podczas gdy wartość 2 oznacza, że możliwe, że to liczba pierwsza (i być może należy przeprowadzić dalsze testowanie). Jeśli x i y są różne, wtedy wartość zwracana 1 oznacza, że x jest wygładzona, (y=1). Wartość zwracana 2 oznacza, że część y jest prawdopodobnie liczbą pierwszą.

9.2.54 xgcd

Funkcja: int xgcd (x,y,xd,yd,z)
big x,y,xd,yd,z);

Moduł: mrxgcd.c

Opis: Oblicza rozszerzony Największy Wspólny Dzielnik dwóch liczb big. Może być również użyty do obliczenia modularnej inwersji. Zauważ, że ten podprogram jest nieco wolniejszy niż operacja mod na liczbach o podobnych rozmiarach.

Parametry: Pięć liczb big x,y,xd,yd i z. Na wyjściu z =gcd(x,y) =x.xd+y.yd

Zwracana wartość: NWD jako liczba całkowita, jeśli możliwe, w przeciwnym razie MR_TOOBIG

Ograniczenia: Jeśli xd i yd nie są różne, tylko xd jest zwracana. NWD jest zwracany tylko jeśli z różni się od obu xd i yd.

Przykład: xgcd (x,p,x,x,x); /* x=1/x mod p (p jest liczbą pierwszą) */

9.3 PODPROGRAMY KRZYWEJ ELIPTYCZNEJ

9.3.1 ebrick_init

Funkcja: BOOL ebrick_init (binst ,x,y,a,b,n,nb)
ebrick *binst;
big x,y;
big a,b,n;
int nb;

Moduł: mrebrick.c

Opis: Inicjalizuje instancję metody Brickell'a dla mnożenia z preobliczeniami krzywej eliptycznej GF(p). Wewnętrznie jest alokowana przestrzeń robocza.

Parametry: Wskaźnik do bieżącej instancji binst, stały punkt G=(x,y) na krzywej $y^2 = x^3+ax+b$, moduł n i maksymalna liczba bitów użytych w wykładniku nb.

Zwracana wartość: TRUE jeśli wszystko poszło dobrze, FALSE jeśli był problem

Ograniczenia: Żadnych

9.3.2 ebrick2_init

Funkcja: BOOL ebrick2_init (binst,x,y,A,B,m.,a,b,c,nb)
ebrick2 *binst
big x,y;
big A,B;
int m.,a,b,c,nb;

Moduł: mrecgf2m.c

Opis: Inicjalizuje instancję metody Brickell'a dla mnożenia z preobliczeniami dla krzywej eliptycznej GF(2^m). pole jest zdefiniowane pod względem podstawy trinomialnej t_m+t_a+1 lub podstawy pentanomialnej $t_m+t_a+t_b+t_c+1$. Wewnętrzne jest alokowana przestrzeń robocza.

Parametry: Wskaźnik do bieżącej instancji binst, stały punkt G=(x,y) na krzywej $y^2+xy = x^3+Ax^2+B$, pole parametrów m.,a,b,c i maksymalna liczba bitów używana w wykładniku nb. b= 0 dla podstawy trinomialnej.

Zwracana wartość: TRUE jeśli wszystko idzie dobrze, FALSE jeśli był probrlem

Ograniczenia: Żadne

9.3.3 ebrick_end*

Funkcja: void ebrick_end (binst)

	ebrick *binst;
Moduł:	mrebrick.c
Opis:	Porządkuje po zastosowaniu metody Brickell'a dla krzywej eliptycznej GF(p)
Parametry:	Wskaźnik do beżącej instancji
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.3.4 ebrick2_end*

Funkcja:	void ebrick2_end (binst) ebrick2 *binst;
Moduł:	mrecgf2m.c
Opis:	Porządkuje po zastosowaniu metody Brickell'a dla krzywej eliptycznej GF(2^m)
Parametry:	Wskaźnik do bieżącej instancji
Zwracana wartość:	Żadna
Ograniczenia:	Żadna

9.3.5 ecurve_add

Funkcja:	void ecurve_add (p,pa) epoint *p, *pa;
Moduł:	mrcurve.c
Opis:	Dodaje dwa punkty na krzywej eliptycznej GF(p) używając specjalnych zasad dla dodawania. Zauważ, ze jeśli pa = p, wtedy jest używana inna zasada duplikowania. Dodawanie jest szybsze jeśli p jest znormalizowne.
Parametry:	Dwa punkty na bieżącej, aktywnej krzywej, pa i p. Na wyjściu pa=pa+p
Zwracana wartość:	Żadna
Ograniczenia:	Punkty wejściowe muszą być w rzeczywistości na bieżącej aktywnej krzywej

9.3.6 ecurve2_add

Funkcja:	void ecurve2_add 9p,pa) epoint *p, *pa;
Moduł:	mrecgf2m.c
Opis:	Dodaje dwa punkty na krzywej eliptycznej GF(2^m) uzywając specjalnych zasad dla dodawania. Zauważ, że jeśli pa = p, wtedy jest używana inna zasada duplikacji. Dodawanie jest szybsze jeśli p jest znormalizowane.
Parametry:	Dwa punkty na bieżącej, aktywnej krzywej pa i p. Na wyjściu pa = pa+p
Zwracana wartość:	Żadna
Ograniczenia:	Punkty wejściowe muszą w rzeczywistości być na bieżącej, aktywnej krzywej

9.3.7 ecurve_init

Funkcja:	void ecurve_init (A,B,p, type) big A,B,p; int type;
Moduł:	mrcurve.c
Opis:	Inicjalizuje wewnętrzne parametry bieżącej aktywnej krzywej eliptycznej GF(p). Krzywa jest z założenia w postaci $y^2 = x^3+Ax+B$ mod p, tzw. model Weierstrass'a. Podprogram ten może być wywołany później z parametrami innych krzywych.
Parametry:	Trzy liczby big A,B i p. Parametr type musi być albo MR_PROJECTIVE albo MR_AFFINE, i określać czy projektowe czy afiniczne współrzędne powinny być użyte wewnętrznie. Zwykle te pierwsze są szybsze
Zwracana wartość:	Żadna
Ograniczenia:	Żadne. Pamięć zaalokowana będzie zwolniona kiedy bieżąca instancja MIRACL jest kończona przez wywołanie mirexit. Tylko jedna krzywa eliptyczna GF(p) lub GF(2^m) może być aktywna wewnątrz pojedynczej instancji MIRACL

9.3.8 ecurve2_init

Funkcja:	BOOL ecurve2_init (m.,a,b,c,A,B,check, type) big A,B; int m.,a,b,c,type; BOOL check;
Moduł:	mrecgf2m.c
Opis:	Inicjalizuje wewnętrzne parametry bieżącej, aktywnej krzywej eliptycznej. Krzywa jest założona w postaci $y^2+xy = x^3+Ax^2+B$. Pole jest zdefiniowane pod względem podstawy trinomialnej t^m+t^a+1 lub podstawy pentanomialnej $t^m+t^a+t^b+t^c+1$. Podprogram ten może być wywołany później z parametrami innej krzywej
Parametry:	stały punkt G=(x,y) na krzywej $y^2+xy = x^3+Ax^2+B$, pole parametrów m.,a,b,c. Ustawione b = 0 dla podstawy trinomialnej. Parametr type musi być albo MR_PROJECTIVE lub MR_AFFINE, i określa czy współrzędne projektowane lub afiniczne powinny zostać użyte wewnętrznie. Zazwyczaj te pierwsze są szybsze. Jeśli check jest TRUE, sprawdzane jest czy określona podstawa jest nieredukowalna.. Jeśli FALSE, sprawdzana jest poprawność tej podstawy., co jest czasochłonne .
Zwracana wartość:	TRUE jeśli parametry mają sens, w przeciwnym razie FALSE
Ograniczenia:	Żadnych. Alokowana pamięć będzie zwolniona kiedy bieżąca instancja MIRACL jest kończona przez wywołanie mirexit. Tylko jedna krzywa eliptyczna GF(p) lub $GF(2^m)$ może być aktywna wewnątrz pojedynczej instancji MIRACL.

9.3.9 ecurve_mult

Funkcja:	void ecurve_mult (k,p,pa) big k; epoint *p. *pa;
Moduł:	mrcurve.c
Opis:	Mnożenie punktu na krzywej eliptycznej GP(p) przez liczbę całkowitą. Używam metod y dodawania / odejmowania
Parametry:	Liczba big k i dwa punkty p i pa. Na wyjściu pa = k*p
Zwracana wartość:	Żadna
Ograniczenia:	Punkt p musi być na aktywnej krzywej

9.3.10 ecurve2_mult

Funkcja:	void ecurve2_mult k,p,pa) big k; epoint *p, *pa;
Moduł:	mrecgf2m.c
Opis:	Mnoży punkt na krzywej eliptycznej $GF(2^m)$ przez liczbę całkowitą. Używa metody dodawania / odejmowania
Parametry:	Liczba big k, i dwa punkty p i pa. Na wyjściu pa = k*pa
Zwracana wartość:	Żadna
Ograniczenia:	Punkt p musi być na aktywnej krzywej

9.3.11 ecurve_mult2

Funkcja:	void ecurve_mult2 (k1,p1,k2,p2,pa) big k1, k2; epoint *p1, *p2, *pa;
Moduł:	mrcurve.c
Opis:	Oblicza punkt k1.p1+k2.p2 na krzywej eliptycznej GF(p). jest to szybsze niż robienie dwóch oddzielnych mnożeń i dodawań. Użyteczne dal pewnych kryptosystemów.
Parametry:	Dwie liczby big k1 i k2, i trzy punkty p1,p2 i pa. Na wyjściu pa =k1.p1 +k2.p2
Zwracana wartość:	Żadna
Ograniczenia:	Punkty p1 i p2 muszą być na aktywnej krzywej

9.3.12 ecurve_mult2

Funkcja:	void ecurve_mult (k1,p1,k2,p2,pa)

	big k1,k2; epoint *p1, *p2, *pa;
Moduł:	mrecgf2m.c
Opis:	Oblicza punkt k1.p1+k2.p2 na krzywej eliptycznej GF(2^m). jest to szybsze niż robienie dwóch oddzielnych mnożeń i dodawań. Użyteczne dla pewnych kryptosystemów.
Parametry:	Dwie liczby big k1 i k2 i trzy punkty p1, p2 i pa. Na wyjściu pa = k1.p1+k2.p2
Zwracana wartość:	Żadna
Ograniczenia:	Punkty p1 i p2 muszą być na aktywnej krzywej

9.3.13 ecurve_mulyi-add

Funkcja:	void ecurve_multi_add (m.,x,w) int m.; epoint **x, **w;
Moduł:	mrcurve.c
Opis:	Równocześnie dodaje pary punktów na aktywnej krzywej GF(p). Jest to dużo szybsze niż dodawanie ich pojedynczo, ale tylko kiedy używamy współrzędnych Afinicznych
Parametry:	Liczba całkowita m. i dwie tablice punktów w i x. Na wyjściu w[i] =w[i]+x[i] dla i = 0 do m.-1
Zwracana wartość:	Żadna
Ograniczenia:	Tylko użyteczna, kiedy używamy współrzędnych Afinicznych

9.3.14 ecurve2_multi_add

Funkcja:	void ecurve2_multi_add (m.,x,w) int m.; epoint **x, **w;
Moduł:	mrecgf2m.c
Opis:	Równocześnie dodaje pary punktów na aktywnej krzywej GF(2^m). Jest to dużo szybsze niż dodawanie ich pojedynczo, ale tylko kiedy używamy współrzędnych afinicznych.
Parametry:	Liczba całkowita m. i dwie tablice punktów w i x. Na wyjściu w[i] = w[i]+x[i] dla i = 0 do m.-1
Zwracana wartość:	Żadna
Ograniczenia:	Tylko użyteczna jeśli używamy współrzędnych afinicznych

9.3.15 ecurve_multn

Funkcja:	void ecurve_multn (n,k,p,pa) int n; big *k; epoint **p;
Moduł:	mrcurve.c
Opis:	Oblicza punkt k[0].p[0]+k[1].p[1]+...+k[n-1].p[n-1] na krzywej eliptycznej GF(p), dla n>2
Parametry:	Liczba całkowita n, tablica n liczb big k[], i tablica n punktów. Wynik jest zwracany w pa
Zwracana wartość:	Żadna
Ograniczenia:	Punkty muszą być na krzywej aktywnej. Wartości k[] wszystkie muszą być dodatnie. Odpowiednia podstawa liczbowa musi być potęgą 2

9.3.16 ecurve2_multn

Funkcja:	void ecurve2_multn (n,k,p,pa) int n; big *k; epoint **p
Moduł:	mrecgf2m.c
Opis:	Oblicza punkt k[0].p[0]+k[1].p[1]+ ...+k[n-1].p[n-1] na krzywej eliptycznej GF(2^m.) , dla n > 2

Parametry:	Liczba całkowita n, tablica n liczb big k[] i tablica n punktów. Wynik jest zwracany w pa
Zwracana wartość:	Żadna
Ograniczenia:	Punkty muszą być na aktywnej krzywej. Wartości k[] muszą wszystkie być dodatnie. Odpowiednia podstawa liczbowa musi być potęgą 2

9.3.17 ecurve_sub

Funkcja:	void ecurve_sub (p,pa) epoint p,pa;
Moduł:	mrcurve.c
Opis;	Odejmuje dwa punkty na krzywej eliptycznej GF(p). W rzeczywistości neguje p i dodaje go do pa. Odejmowanie jest szybsze jeśli p jest znormalizowane.
Parametry:	Dwa punkty na bieżącej, aktywnej krzywej, pa i p. Na wyjściu pa = pa-p.
Zwracana wartość:	Żadna
Ograniczenia:	Wejściowe punkty muszą w rzeczywistości być na bieżącej krzywej aktywnej.

9.3.18 ecurve2_sub

Funkcja:	void ecurve2_sub (p,pa) epoint *p, *pa;
Moduł:	mrecgf2m.c
Opis:	Odejmuje dwa punkty na krzywej eliptycznej GF(2^m). W rzecywistosci neguje p i dodaje go do pa. Odejmowanie jest szybsze jeśli p jest znormalizowane.
Parametry:	Dwa punkty na bieżącej aktywnej krzywej, pa i p. Na wyjściu pa = pa -p
Zwracana wartość:	Żadna
Ograniczenia:	Punkty wejściowe muszą w rzeczywistości być na bieżącej, aktywnej krzywej

9.3.19 epoint_comp

Funkcja:	BOOL epoint_comp (p1,p2) epoint *p1, *p2;
Moduł:	mrcurve.c
Opis:	Porównuje dwa punkty na bieżącej aktywnej krzywej eliptycznej GF(p)
Parametry:	Dwa punkty p1 i p2
Zwracana wartość:	TRUE jeśli punkty są takie same, FALSE w przeciwnym razie.
Ograniczenia:	Żadne

9.3.20 epoint2_comp

Funkcja:	BOOL epoint2_comp (p1,p2) epoint *p1, *p2;
Moduł:	mrcurve.c
Opis:	Porównuje dwa punkty na bieżącej aktywnej krzywej eliptycznej GF(2^m)
Parametry:	Dwa punkty p1 i p2
Zwracana wartość:	TRUE jeśli punkty są takie same, FALSE w przeciwnym razie.
Ograniczenia:	Żadne

9.3.21 epoint_copy*

Funkcja:	void epoint_copy (p1, p2) epoint *p1, *p2
Moduł:	mrcurve.c
Opis:	Kopiuje jeden punkt do innego na krzywej eliptycznej GF(p)
Parametry:	Dwa punkty p1 i p2. Na wyjściu p2 = p1
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.3.22 epoint2_copy*

Funkcja:	void epoint2_copy (p1, p2) epoint *p1, *p2
Moduł:	mrcurve.c
Opis:	Kopiuje jeden punkt do innego na krzywej eliptycznej $GF(2^m)$
Parametry:	Dwa punkty p1 i p2. Na wyjściu p2 = p1
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.3.23 epoint_free*

Funkcja:	void epoint_free (p) epoint *p;
Moduł:	mrcurve.c
Opis:	Zwalnia pamięć powiązaną z punktem na krzywej eliptycznej GF(p)
Parametry:	A punkt p
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.3.24 epoint2_free*

Funkcja:	void epoint2_free (p) epoint *p;
Moduł:	mrcurve.c
Opis:	Zwalnia pamięć powiązaną z punktem na krzywej eliptycznej $GF(2^m)$
Parametry:	A punkt p
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.3.25 epoint_get

Funkcja:	int epoint_get (p,x,y) epoint *p; big x, y;
Moduł:	mrcurve.c
Opis:	Normalizuje punkt i wyodrębnia jego współrzędne (x,y) na aktywnej krzywej eliptycznej GF(p)
Parametry:	A punkt p, i dwie liczby całkowite big x i y. Jeśli x i y nie są różnymi zmiennymi na wejściu, wtedy tylko wartość x jest zwracana
Zwracana wartość:	Najmniej znaczący bit y. Zauważ, że jest możliwe jest zrekonstruowanie punktu z jego współrzędnej x i najmniej znaczącego bitu y. Często taki „skondensowany' opis punktu jest użyteczny.
Ograniczenia;	Punkt p musi być na aktywnej krzywej
Przykład:	i =epoint_get (p,x,x); /* wyodrębnienie współrzędnej x i lsb z y */

9.3.26 epoint_getxyz

Funkcja:	void epoint_getxyz (p,x,yz) epoint *p; big x,y,z;
Moduł:	mrcurve.c
Opis:	Wyodrębnia nieprzetworzone współrzędne (x,y,z) punktu na krzywej eliptycznej GF(p).
Parametry:	A punkt p i trzy liczby całkowite x,y i z. Jeśli jakaś z nich jest NULL wtedy współrzędna nie jest zwracana.
Zwracana wartość:	Żadna
Ograniczenia:	Punkt p musi być na aktywnej krzywej

9.3.27 epoint2_get

Funkcja:	int epoint_get (p,x,y) epoint *p; big x, y;
Moduł:	mrcurve.c
Opis:	Normalizuje punkt i wyodrębnia jego współrzędne (x,y) na aktywnej krzywej eliptycznej GF(2^m)
Parametry:	A punkt p, i dwie liczby całkowite big x i y. Jeśli x i y nie są różnymi zmiennymi na wejściu, wtedy tylko wartość x jest zwracana
Zwracana wartość:	Najmniej znaczący bit y. Zauważ, że jest możliwe jest zrekonstruowanie punktu z jego współrzędnej x i najmniej znaczącego bitu y. Często taki „skondensowany' opis punktu jest użyteczny.
Ograniczenia;	Punkt p musi być na aktywnej krzywej
Przykład:	i =epoint_get (p,x,x); /* wyodrębnienie współrzędnej x i lsb z y */

9.3.28 epoint2_getxyz

Funkcja:	void epoint2_getxyz (p,x,yz) epoint *p; big x,y,z;
Moduł:	mrcurve.c
Opis:	Wyodrębnia nieprzetworzone współrzędne (x,y,z) punktu na krzywej eliptycznej GF(2^m).
Parametry:	A punkt p i trzy liczby całkowite x,y i z. Jeśli jakaś z nich jest NULL wtedy współrzędna nie jest zwracana.
Zwracana wartość:	Żadna
Ograniczenia:	Punkt p musi być na aktywnej krzywej

9.3.29 epoint_init

Funkcja:	epoint * epoint_init()
Moduł:	mrcurve.c
Opis:	Przydziela pamięć do punktu na krzywej eliptycznej GF(p) i inicjalizuje go do „wskazywania nieskończoności"
Parametry:	Żadne
Zwracana wartość:	A punkt p (faktycznie wskaźnik do struktury alokowanej ze sterty)
Ograniczenia:	Obowiązkiem programisty C jest założyć, że wszystkie punkty na krzywej eliptycznej inicjalizowane przez wywołanie tej funkcji, są ostatecznie zwalniane przez epoint_free

9.2.30 epoint2_init

Funkcja:	epoint * epoint2_init()
Moduł:	mrcurve.c
Opis:	Przydziela pamięć do punktu na krzywej eliptycznej GF(2^m) i inicjalizuje go do „wskazywania nieskończoności"
Parametry:	Żadne
Zwracana wartość:	A punkt p (faktycznie wskaźnik do struktury alokowanej ze sterty)
Ograniczenia:	Obowiązkiem programisty C jest założyć, że wszystkie punkty na krzywej eliptycznej inicjalizowane przez wywołanie tej funkcji, są ostatecznie zwalniane przez epoint_free

9.3.31 epoint_norm

Funkcja:	BOOL epoint_norm (p) epoint *p;
Moduł:	mrcurve.c

Opis:	Normalizuje punkt na bieżącej, aktywnej krzywej eliptycznej GF(p). Ustawia współrzędną z na 1. Punkt dodawany jest szybciej jeśli dodajemy punkt znormalizowany. Funkcja ta nie robi nic, jeśli są używane współrzędne afiniczne (w przypadku których nie ma współrzędnej z)
Parametry:	A punkt na bieżącej aktywnej krzywej eliptycznej
Zwracana wartość:	TRUE jeśli pomyślnie

9.3.32 epoint2_norm

Funkcja:	BOOL epoint2_norm (p) epoint *p;
Moduł:	mrcurve.c
Opis:	Normalizuje punkt na bieżącej, aktywnej krzywej eliptycznej GF(2^m). Ustawia współrzędną z na 1. Punkt dodawany jest szybciej jeśli dodajemy punkt znormalizowany. Funkcja ta nie robi nic, jeśli są używane współrzędne afiniczne (w przypadku których nie ma współrzędnej z)
Parametry:	A punkt na bieżącej aktywnej krzywej eliptycznej
Zwracana wartość:	TRUE jeśli pomyślnie

9.3.33 epoint_set

Funkcja:	BOOL epoint_set (x,y,lsb,p) big x,y; int lsb; epoint *p
Moduł:	mrcurve.c
Opis:	Ustawia punkt na bieżącej aktywnej ,krzywej eliptycznej GF(p) (jeśli możliwe)
Parametry:	Całkowite współrzędne x i y punktu p. Jeśli x i y nie są różnymi zmiennymi wtedy tylko x k jest przekazywane do funkcji, a lsb jest brane jako najmniej znaczący bit y. W tym przypadku pełna wartość y jest rekonstruowana wewnętrznie. Jest to znane jako „punkt dekompresji” (i jest trochę czasochłonne, wymagające wykonania modularnego pierwiastka kwadratowego). Na wyjściu p =(x,y)
Zwracana wartość:	TRUE jeśli punkt istnieje na bieżącym aktywnym punkcie, w przeciwnym razie FALSE
Ograniczenia:	Żadnych
Przykład:	p = epoint_init (); epoint_set (x,x,1,p); /* dekompresja p */

9.3.34 eporint2_set

Funkcja:	BOOL epoint_set (x,y,lsb,p) big x,y; int lsb; epoint *p
Moduł:	mrecgf2m.c
Opis:	Ustawia punkt na bieżącej aktywnej ,krzywej eliptycznej GF(2^m) (jeśli możliwe)
Parametry:	Całkowite współrzędne x i y punktu p. Jeśli x i y nie są różnymi zmiennymi wtedy tylko x jest przekazywane do funkcji, a lsb jest brane jako najmniej znaczący bit z y/x. W tym przypadku pełna wartość y jest rekonstruowana wewnętrznie. Jest to znane jako „punkt dekompresji” (i jest trochę czasochłonne, wymagające wyodrębnienia z pola pierwiastka kwadratowego). Na wyjściu p =(x,y)
Zwracana wartość:	TRUE jeśli punkt istnieje na bieżącym aktywnym punkcie, w przeciwnym razie FALSE
Ograniczenia:	Żadnych
Przykład:	p = epoint_init (); epoint_set (x,x,1,p); /* dekompresja p */

9.3.35 mul_brick

Funkcja: int mul_brick (binst,e,x,y)
ebrick *binst;
big e,x,y;
Moduł: mrebrick.c
Opis: Wykonuje mnożenie GF(p) krzywej eliptycznej używając preobliczonych wartości przechowywanych w strukturze ebrick
Parametry: Wskaźnik do bieżącej instancji, wykładnik big e i liczba big w. Na wyjściu (x,y) = e.G mod n, gdzie G i n są określonymi w początkowym wywołaniu ebrick_init. Jeśli x i y nie są różnymi zmiennymi, zwracane jest tylko x.
Zwracana wartość: Najmniej znaczący bit z y
Ograniczenia: Musi być poprzedzony przez wywołanie ebrick_init

9.3.36 mul2_brick

Funkcja: int mul2_brick (binst,e,x,y)
ebrick *binst;
big e,x,y;
Moduł: mrecgf2m.c
Opis: Wykonuje mnożenie GF(2^m) krzywej eliptycznej używając preobliczonych wartości przechowywanych w strukturze ebrick2.
Parametry: Wskaźnik do bieżącej instancji, wykładnik big e i liczba big w. Na wyjściu (x,y) = e.G, gdzie G jest określonymi w początkowym wywołaniu ebrick2_init. Jeśli x i y nie są różnymi zmiennymi, zwracane jest tylko x.
Zwracana wartość: Najmniej znaczący bit z y / x
Ograniczenia: Musi być poprzedzony przez wywołanie ebrick2_init

9.3.37 point_at_infinty*

Funkcja: BOOL point_ta_infinity (p)
epoint *p;
Moduł: mrcore.c
Opis: Testuje czy punkt krzywej eliptycznej „wskazuje nieskończoność”
Parametry; Punkt na krzywej eliptycznej p
Zwracana wartość: TRUE jeśli p jest punktem - na - nieskończoność, w przeciwnym razie FALSE
Ograniczenia: Punkt musi być zainicjalizowany.

9.4 PODPROGRAMY SZYFRUJĄCE

9.4.1 aes_decrypt*

Funkcja: mr_unsign32 aes_decrypt (a, buff)
aes *a;
cgar *buff;
Moduł: mraes.c
Opis: Deszyfruje 16 lub n bajtów bufora wejściowego
Parametry: Wskaźnik na zainicjalizowaną instancję struktury aes zdefiniowaną w miracl.h, i bufora bajtów poddanego deszyfrowaniu. Jeśli trybem działania jest szyfr blokowy (MR_ECB lub MR_CBC) wtedy będzie odszyfrowane 16 bajtów. Jeśli trybem działania jest szyfr strumieniowy (MR_CFBn, MR_OFBn lub MR_PCFBn) wtedy n bajtów będzie odszyfrowane.
Zwracana wartość: W trybach MR_CFBn i MR_PCFBn n bajt(y), który był przesunięty na koniec wejściowego rejestru jako wynik deszyfrowania n wejściowych bajtach, w przeciwnym razie 0.

9.4.2 aes_encrypt*

Funkcja: mr_unsign32 aes_encrypt (a, buff)
aes *a;

	cgar *buff;
Moduł:	mraes.c
Opis:	Szyfruje 16 lub n bajtów bufora wejściowego
Parametry:	Wskaźnik na zainicjalizowaną instancję struktury aes zdefiniowaną w miracl.h, i bufora bajtów poddanego szyfrowaniu. Jeśli trybem działania jest szyfr blokowy (MR_ECB lub MR_CBC) wtedy będzie szyfrowane 16 bajtów. Jeśli trybem działania jest szyfr strumieniowy (MR_CFBn, MR_OFBn lub MR_PCFBn) wtedy n bajtów będzie szyfrowane.
Zwracana wartość:	W trybach MR_CFBn i MR_PCFBn n bajt(y), który był przesunięty na koniec wejściowego rejestru jako wynik szyfrowania n wejściowych bajtach, w przeciwnym razie 0.

9.4.3 aes_end*

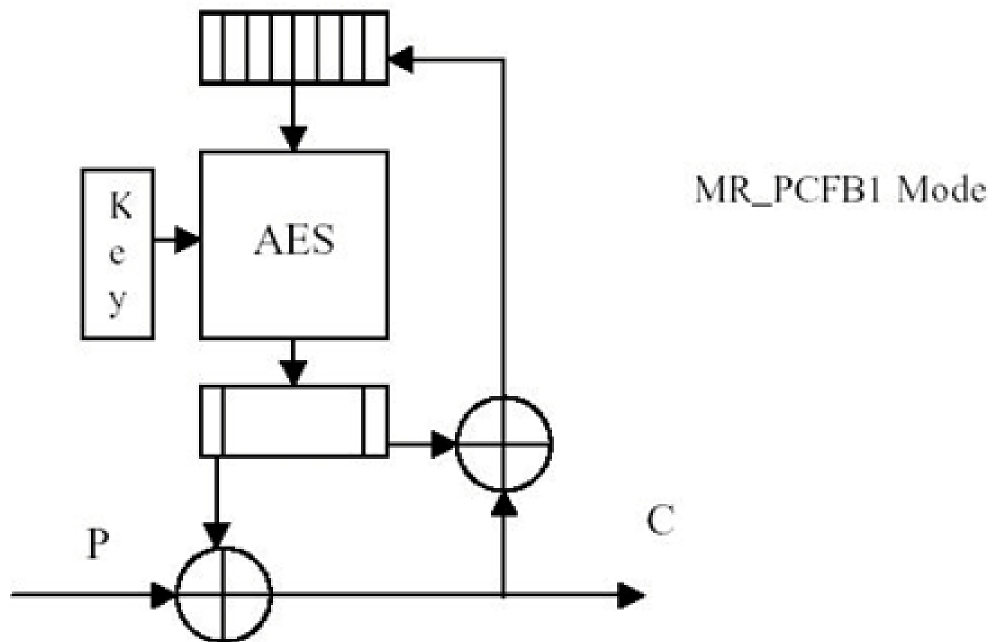
Funkcja:	void aes_end (a) aes *a;
Moduł:	mraes.c
Opis:	Kończy sesję szyfrowania AES i de alokuje pamięć z nią powiązaną. Niszczony jest wewnętrzny klucz sesyjny.
Parametry:	Wskaźnik do zainicjalizowanej instancji struktury aes zdefiniowanej w miracl.h
Zwracana wartość:	Żadna

9.4.4 aes_getreg*

Funkcja:	void aes_getreg (a, ir) aes *a; char *ir;
Moduł:	mraes.c
Opis:	Odczytuje bieżącą zawartość wejściowego rejestru łańcuchowego powiązanego z tą instancją AES. Jest to rejestr zainicjalizowany przez IV w wywołaniu aes_init i aes_reset.
Parametry;	Wskaźnik do instancji struktury aes, zdefiniowanej w miracl.h i tablica znaków przechowująca wyodrębnioną 16 bajtową daną
Zwracana wartość:	Żadna

9.4.5 aes_init

Funkcja:	BOOL aes_init (a, mode,nk, key,iv) aes *a; int mode , nk; char key, iv;
Moduł:	mraes.c
Opis:	Inicjalizuje sesję Szyfrowania / Deszyfrowania uzywając Zaawansowanego Systemu Szyfrowania (AES). Jest to system szyfru blokowego, który szyfruje dane w 128 bitowych blokach stosując klucz 128, 192 i 256 bitowy.
Parametry:	Wskaźnik do instancji struktury aes zdefiniowanej w miracl.h, tryb operacji będącej w użyciu, liczba całkowita nk, która określa rozmiar Klucza w bajtach i wskaźniki do samego klucza i opcjonalnie Wektor Inicjujący (IV). Tryb może być jednym z MR_ECB(Elektroniczna Książka Kodów), MR_CFB (Wiązanie Bloków Zaszyfrowanych), MR_CFBn (Sprzężenie zwrotne zaszyfrowanego tekstu gdzie n to 1,2 lub 4), MR_PCFBn (błąd Propagującego Sprzężenia zwrotnego zaszyfrowanego tekstu gdzie n to 1,2 lub 4) lub MR_OFBn (Sprzężenie zwrotne bloków wyjściowych gdzie n to 1,2,4,8 lub 16).Wartość n wskazuje liczbę bajtów będących przetwarzanymi w każdym zastosowaniu. Po więcej informacji o Trybach Działania zobacz [Stinton]. MR_PCFBn jest naszym własnym wymysłem [Scott93]. Wartość nk to 16,24 lub 32. 16 bajtowy wektor inicjujący iv powinien być określony dla wszystkich trybów innych niż MR_ECB, w którym to przypadku może być NULL.
Zwracana wartość:	TRUE jeśli zainicjalizowano pomyślnie, w przeciwnym razie FALSE



9.4.6 aes_reset*

Funkcja:	void aes_reset (a, mode, iv) aes *a; int mode; char *iv;
Moduł:	mraes.c
Opis:	Resetuje strukturę AES
Parametry:	Wskaźnik do instancji struktury aes zdefiniowanej w miracl.h, znak nowego trybu działania i wskaźnik do (być może nowego) wektora inicjującego iv.
Zwracana wartość:	Żadna

9.4.7 shs_init

Funkcja:	void shs_init (psh) sha *psh;
Moduł:	mrshs.c
Opis:	Inicjalizuje instancję Secure Hash Algorithm SHA-1. Musi być wywołany przed nowym użyciem
Parametry:	Wskaźnik do instancji struktury zdefiniowanej w miracl.h
Zwracana wartość:	Żadna

9.4.8 shs_hash*

Funkcja:	void shs_hash (psh, hash) sha *psh; char hash[20];
Moduł:	mrshs.c
Opis:	Generuje dwudziesto bajtową (160 bitów) wartość klucza haszującego w dostarczonej tablicy
Parametry:	Wskaźnik do bieżącej instancji i wskaźnik do tablicy mającej być wypełnioną
Zwracana wartość:	Żadna

9.4.9 shs_process*

Funkcja:	void shs_process (psh,ch) sha *psh; int ch;

Moduł:	mrshs.c
Opis:	Przetwarza pojedynczy bajt. Zazwyczaj wywoływany wiele razy dostarczając danych wejściowych do procesu haszowania. Wartość klucza haszującego wszystkich przetwarzanych bajtów może być odzyskany poprzez późniejsze wywołanie shs_hash
Parametry:	Wskaźnik do bieżącej instancji i znak do przetworzenia
Zwracana wartość:	Żadna

9.4.10 shs256_init*

Funkcja:	void shs256_init (psh) sha256 *psh;
Moduł:	mrshs256.c
Opis:	Inicjalizuje instancję Secure Hash Algorithm SHA-256. Musi być wywołany przed nowym użyciem
Parametry:	Wskaźnik do instancji struktury zdefiiowanej w miracl.h
Zwracana wartość:	Żadna

9.4.11 shs256_hash*

Funkcja:	void shs256_hash (psh, hash) sha256 *psh; char hash[32];
Moduł:	mrshs256.c
Opis:	Generuje 32 bajtową (256 bitów) wartość klucza haszujacego w dostarczonej tablicy
Parametry:	Wskaźnik do bieżącej instancji i wskaźnik do wypełnianej tablicy
Zwracana wartość:	Żadna

9.4.12 shs256_process*

Funkcja:	void shs256_process (psh, ch) sha256 *psh; int ch;
Moduł:	mrshs256.c
Opis:	Przetwarza pojedynczy bajt. Zazwyczaj wywoływany wiele razy dostarczając dane do procesu haszowania. Wartość klucza haszującego wszystkich przetwarzanych bajtów może być odzyskana przez późniejsze wywołanie shs256_hash.
Parametry:	Wskaźnik do bieżącej instancji i znak do przetwarzania
Zwracana wartość:	Żadna

9.4.13 shs384_init*

Funkcja:	void shs384_init (psh) sha384 *psh;
Moduł:	mrshs512.c
Opis:	Inicjalziuje instancję Secure Hash Algorithm SHA -384. Musi być wywołany przed nowym użyciem
Parametry:	Wskaźnik do instancji struktury zdefiniowanej w miracl.h
Zwracana wartość:	Żadna
Ograniczenia:	Algorytm SHA-384 jest dostępny tylko jeśli jest zdefiniowana 64 bitowa dana

9.4.14 shs384_hash*

Funkcja:	void shs384_hash (psh, hash) sha384 *psh; char hash[48];
Moduł:	mrshs512,c
Opis:	Generuje 48 bajtową (384 bitów) wartość klucza haszujacego w dostarczonej tablicy.
Parametry:	Wskaźnik do bieżącej instancji i wskaźnik do wypełnianej tablicy
Zwracana wartość:	Żadna

9.4.15 shs384_process*

Funkcja:	void shs512_process (psh,ch) sha384 *psh; int ch;
Moduł:	mrshs512.c
Opis:	Przetwarza pojedynczy bajt. Zazwyczaj wywoływany wiele razy dostarczając danych wejściowych do procesu haszowania. Wartość klucza haszującego wszystkich przetwarzanych bajtów może być odzyskana przez późniejsze wywołanie shs384_hash
Parametry:	Wskaźnik do bieżącej instancji i znak będący przetwarzanym
Zwracana wartość:	Żadna

9.4.16 shs512_init*

Funkcja:	void shs512_init (psh) sha512 *psh;
Moduł:	mrshs512.c
Opis:	Inicjalizuje instancję Secure Hash Algorithm SHA-512. Musi być wywołany przed nowym użyciem
Parametry:	Wskaźnik do instancji struktury zdefiniowanej w miracl.h
Zwracana wartość:	Żadna
Ograniczenia:	Algorytm SHA-512 jest dostępny tylko jeśli jest zdefiniowana 64 bitowy typ danej

9.4.17 shs512_hash*

Funkcja:	void shs512_hash (psh, hash) sha512 *psh; char hash[64];
Moduł:	mrshs512.c
Opis:	Generuje 64 bajtową (512 bitów) wartość klucza haszującego w dostarczonej tablicy
Parametry:	Wskaźnik do bieżącej instancji i wskaźnik do wypełnianej tablicy
Zwracana wartość:	Żadna

9.4.18 shs512_process*

Funkcja:	void shs512_process (psh,ch) sha512 *psh; int ch;
Moduł:	mrshs512.c
Opis:	Przetwarza pojedynczy bajt. Zazwyczaj wywoływany wiele razy dostarczając dane do procesu haszowania. Wartość klucza haszującego wszystkich przetwarzanych bajtów może być odzyskana przez późniejsze wywołanie shs512_hash
Parametry:	Wskaźnik bieżącej instancji i znak będący przetwarzanym
Zwracana wartość:	Żadna

9.4.19 strong_bigdig

Funkcja:	void strong_bigdig (rng,n,b,x) csprng *rng; int n,b; big x;
Moduł:	mrstrong.c
Opis:	generuje losową liczbę big o danej długości z kryptograficznego silnego generatora rng
Parametry:	Wskaźnik do losowego generatora liczb rng. Liczba big x i dwie liczby całkowite n i b. Na wyjściu x zawiera losową liczbę o n cyfrach o podstawie b
Zwracana wartość:	Żadna
Ograniczenia:	Podstawa b musi być drukowalna, to znaczy $2 \leq b \leq 256$

9.4.20 strong_bigrand

Funkcja:	void strong_bigrand (rng,w,x) csprng *rng; big w,x;
Moduł:	mrstrong.c
Opis:	Generuje silną kryptograficznie liczbę losową big x używając generatora liczb losowych rng, taką, że $0 \le x < w$
Parametry:	Dwie liczby big w i x i generator liczb losowych rng
Zwracana wartość:	Żadna

9.4.21 storng_init*

Funkcja:	void strong_init (rng,rawlen, raw, tod) csprng *rng; int rawlen; char *raw; long tod;
Moduł:	mrstrong.c
Opis:	Inicjalizuje silny kryptograficznie generator liczb losowych rng
Parametry:	Wskaźnik do generatora liczb losowych rng. Tablica raw o długości rawlen i 32 bitowa wartość tod. Te dwa źródła są używane razem do generowania ziarnistego. Pierwszy może być dostarczony przez losowe uderzenia w klawisze, drugi z wewnętrznego zegara. Późniejsze wywołania strong_rng dostarczą losowych bajtów.
Zwracana wartość:	Żadna
Przykład:	Zobacz test1363.c i p1363.c jako przykład użycia

9.4.22 strong_kill*

Funkcja:	void strong_kill (rng) csprng *rng;
Moduł:	mrstrong.c
Opis:	Kasuje wewnętrzny stan generatora liczb losowych rng
Parametry:	Wskaźnik do generatora liczb losowych
Zwracana wartość:	Żadna

9.4.23 strong_rng*

Funkcja:	int strong_rng (rng) csprng *rng;
Moduł:	mrstrong.c
Opis:	Generuje sekwencję silnych kryptograficznie bajtów losowych
Parametry:	Wskaźnik do liczby losowej
Zwracana wartość:	Losowy bajt

9.5 PODPROGRAMY FLOATING-SLASH

9.5.1 build

Funkcja:	void build (x, gener) flash x; int (*gener);
Moduł:	mrbuild.c

Opis:	Używa dostarczonego zwykłego generatora rozszerzonego ułąmka do zbudowania liczby flash x, zaokrąglonej jeśli trzeba.
Parametry:	Tworzona liczba flash i generator funkcji
Zwracana wartość:	Żadna
Przykład:	int phi (w,n) flash w; int n; { /* generator rcf dla złotego współczynnika */ return 1; } build (x, phi); Oblicza złoty współczynnik (1+ √5)/2 w x - bardzo szybko

9.5.2 dconv

Funkcja:	void dconv (d,x) double d; flash x;
Moduł:	mrflash.c
Opis:	Konwertuje liczbę double do postaci flash
Parametry:	Podwójna liczba d i zmienna flash x. Na wyjściu x będzie zawierało odpowiednik flash d
Zwracane wartości:	Żadne

9.5.3 denom

Funkcja:	void denom (x,y) flash x; big y;
Moduł:	mrcore.c
Opis:	Wyodrębnia mianownik z liczby flash
Parametry:	Liczba flash x i liczba big y. Na wyjściu y będzie zawierało mianownik z x
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.4 facos

Funkcja:	void facos (x,y) flash x,y;
Moduł:	mrflsh3.c
Opis:	Oblicza arcus - kosinus liczby flash używając fasin
Parametry:	Dwie liczby flash xi y. Na wyjściu y= arcos(x)
Zwracana wartość:	Żadna
Ograniczenia:	/x/ musni być mniejsze niż lub równe 1

9.5.5 facosh

Funkcja:	void facosh (x,y) flash x,y;
Moduł:	mrflsh4.c
Opis:	Oblicza hiperboliczny arkus - kosinus liczby flash
Parametry:	Dwie liczby flash x i y. Na wyjściu y = arccosg(x)

Zwracana wartość: Żadna
Ograniczenie: /x| musi być większe niż lub równe 1

9.5.6 fadd

Funkcja: void fadd (x,y,z)
flash x,y,z;
Moduł: mrflash.c
Opis: Dodaje dwie liczby flash
Parametry: Trzy liczby flash x,y i z. Na wyjściu: z=x+y
Zwracana wartość: Żadna
Ograniczenia: Żadnych

9.5.7 fasin

Funkcja: void fasin (x,y)
flash x,y;
Moduł: mrflsh3.c
Opis: Oblicza arkus - sinus liczby flash używając fatan
Parametry: Dwie liczby flash xi y . Na wyjściu y =arcsin(x)
Zwracana wartość: Żadna
Ograniczenia: /x| musi być mniejsze niż lub równe 1

9.5.8 fasinh

Funkcja: void fasinh 9x,y)
flash x,y;
Moduł: mrflsh4.c
Opis: Oblicza hiperboliczny arkus -sinus liczby flash
Parametry: Dwie liczby flash xi y .Na wyjściu y= arcsinh(x)
Zwracana wartość: Żadna

9.5.9 fatan

Funkcja: void fatan (x,y)
flash x,y;
Moduł: mrflsh3.c
Opis: Oblicza arkus - tnagens liczby flash, używając metody $O(n^{2.5})$ opartej na iteracji Newtona
Parametry: Dwie liczby flash x i y. Na wyjściu y = arctan(x)
Zwracana wartość: Żadna

9.5.10 fatanh

Funkcja: void fatanh 9x,y)
flash x,y;
Moduł: mrflsh4.c
Opis: Oblicza hiperboliczny arkus - tangens liczby flash
Parametry; Dwie liczby flash x i y. Na wyjściu y = arctanh(x)
Zwracana wartość: Żadna
Ograniczenia: x^2 musi być mniejsze niż 1

9.5.11 fcomp

Funkcja: int fcomp (x,y)
flash x,y;
Moduł: mrflash.c
Opis: Porównuje dwie liczby flash.
Parametry: Dwie liczby flash xi y
Wartość zwracana: Zwraca -1 jeśli y>x, +1 jeśli x>y i 0 jeśli x = y

9.5.12 fconv

Funkcja: void fconv (n,d,x)
int n,d;
flash x;
Moduł: mrflash.c
Opis: Konwertuje prosty ułamek do postaci flash
Parametry: Liczby całkowite n i d i liczba flash x. Na wyjściu x = n/d
Zwracana wartość: Żadna

9.5.13 fcos

Funkcja: void fcos (x,y)
Flash x,y
Moduł: mrflsh3.c
Opis: Oblicza kosinus danego kąta flash używając ftan
Parametry: Dwie liczby flash x i y. Na wyjściu y = cos(x)
Zwracana wartość: Żadna
Ograniczenia: Żadne

9.5.14 fcosh

Funkcja: void fcosh (x,y)
flash x,y;
Moduł: mrflsh4.c
Opis: Oblicza hiperboliczny kosinus danego kąta flash
Parametry: Dwie liczby flash x i y. Na wyjściu y =cosh(x)
Zwracana wartość: Żadna

9.5.15 fdiv

Funkcja: void fdiv (x,y,z)
flash x,y,z;
Moduł: mrflash.c
Opis: Dzieli dwie liczby flash
Parametry: Trzy liczby big x,y, iż. Na wyjściu z= x/y
Zwracana wartość: Żadna

9.5.16 fdsize

Funkcja: double fdsize 9x)
flash x;
Moduł: mrdouble.c
Opis: konwertuje liczbę flash do formatu double
Parametry: Liczba flash x
Zwracana wartość: Wartość parametru x jako double

Ograniczenia:	Wartość x musi być przedstawiana jako double

9.5.17 fexp

Funkcja:	void fexp (x,y) flash x,y;
Moduł:	mrflsh2.c
Opis:	Oblicza wykładnik liczby flash używając metody $O(n^{2.5})$
Parametry:	Dwie liczby flash xi y .Na wyjściu $y=e^{x}$.
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.18 fincr

Funkcja:	void fincr (x,,n,d,y) big x,y; int n,d;
Moduł:	mrflash.c
Opis:	Dodaje prosty ułamek do liczby flash
Parametry:	Dwie liczby flash x iy, i dwie liczby całkowite n i d. Na wyjściu y=x +n/d
Zwracana wartość:	Żadna
Ograniczenia:	Żadne
Przykład:	fincr (x,-2,3,x); Odejmuje dwie trzecie od wartości x

9.5.19 flog

Funkcja:	void flog (x,y) flash x,y;
Moduł:	mrflsh2,c
Opis:	Oblicza naturalny logarytm liczby flash używając metody $O(n^{2.5})$
Parametry:	Dwie liczby flash x i y. Na wyjściu y =log(x)
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.20 flop

Funkcja:	void bflop (x,y,op,z) flash x,y,z; int *op;
Moduł:	mrflash.c
Opis:	Wykonuje podstawowe operacje flash. Używany wewnętrznie
Parametry:	Trzy liczby flash x,y iż. Na wyjściu z =Fn(x,y), gdzie wykonanie funkcji zależy od parametru op. Zobacz źródłowy listing z komentarzami
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.21 fmodulo

Funkcja:	void fmodulo (x,y,z) flash x,y,z;
Moduł:	mrflash.c
Opis:	Znajduje resztę, kiedy jedna liczba flash jest dzielona przez inną
Parametry:	Trzy liczby flash x,y iż. Na wyjściu z = x mod y

Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.22 fmul

Funkcja:	void fmul (x,y,z) flash x,y,z;
Moduł:	mrflash.c
Opis:	Mnoży dwie liczby flash
Parametry:	Trzy liczby flash x,y i z. Na wyjściu z = x.y
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.23 fpack

Funkcja:	void fpack (n,d,x) flash x; big n, d;
Moduł:	mrcore.c
Opis:	Formuje liczbę flash z licznika i mianownika big
Parametry:	Liczba flash x i dwie liczby big n i d. Na wyjściu x =n/d
Zwracana wartość:	Żadna
Ograniczenia:	Mianownik musi być nie zerowy. Zmienna flash x i zmienna big d muszą się różnić. Wynikowa zmienna flash nie może być zbyt duża do przedstawienia.

9.5.24 fpi

Funkcja:	void fpi (x) flash x;
Moduł:	mrpi.c
Opis:	Oblicza π stosując metodę Gaussa-Legendre'a $O(n^2.\log n)$. Zauważ, że w późniejszych wywołaniach tego podprogramu, π jest dostępne bezpośrednio, ponieważ jest przechowywana wewnętrznie. (Podprogram ten jest zaskakująco wolny .Pojawił się, chociaż jest prostszy sposób obliczenia szybko wymiernego przybliżenia π)
Parametry:	Liczba flash x. Na wyjściu x = π
Zwracana wartość:	Żadna
Ograniczenia:	Żadne. Wewnętrznie alokowana pamięć jest zwalniana kiedy bieżąca instancja MIRACL jest kończona przez wywołanie mirexit.

9.5.25 fpmul

Funkcja:	void fpmul 9x,n,d,y) flash x,y; int n,d;
Moduł:	mrflash.c
Opis:	Mnoży liczbę flash przez prosty ułamek
Parametry:	Dwie liczby flash x i y i dwie liczby całkowite n id. Na wyjściu y =x. n/d
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.26 fpower

Funkcja:	void fpower (x,n,y)

flash x,y;
int n;

Moduł: mrflsh1.c
Opis: Podnosi liczbę flash do potęgi całkowitej
Parametry: Zmienne flash x i y, i liczba całkowita n. Na wyjściu $y=x^n$.
Zwracana wartość: Żadna

9.5.27 fpowf

Funkcja: void fpowf (x,y,z)
flash x,y,z;

Moduł: mrflsh2,c
Opis: Podnosi do potęgi flash liczbę flash
Parametry: trzy liczby flash x,y i z.Na wyjściu $z =x^y$.
Zwracana wartość: Żadna

9.5.28 frand

Funkcja: void frand (x)
flash x;

Moduł: frnd.c
Opis: Generuje liczbę losową flash
Parametry: Liczba big x. Na wyjściu x zawiera losową liczbę flash w zakresie $0 < x <1$
Zwracana wartość: Żadna

9.5.29 frecip

Funkcja: void frecip (x,y)
flash x,y;

Moduł: mrflash.c
Opis: Oblicza odwrotność liczby flash
Parametry: Dwie liczby flash x i y .Na wyjściu $y = 1/x$
Zwracana wartość: Żadna

9.5.30 froot

Funkcja: BOOL froot (x,m.,y)
flash x,y;
int m.;

Moduł: mflsh1.c
Opis: Oblicza m-ty pierwiastek liczby flash używając metody Newtona $O(n^2)$
Parametry: Liczby flash x i y i liczba całkowita m. Na wyjściu y jest m.-tym pierwiastkiem z x
Zwracana wartość: TRUE jeśli jest pierwiastek, w przeciwnym razie FALSE

9.5.31 fsin

Funkcja: void fsin (x,y)
flash x,y;

Moduł: mrflsh3.c
Opis: Oblicza sinus danego kąta flash. Używa ftan
Parametry: Dwie liczby flash xi y. Na wyjściu y=sin(x)
Zwracana wartość: Żadna

9.5.32 fsinh

Funkcja:	void fsinh (x,y) flash x,y;
Moduł:	mrflsh4.c
Opis:	Oblicza hiperboliczny sinus danego kąta flash
Parametry:	Dwie liczby flash x i y. Na wyjściu y =sinh(x)
Wartość zwracana:	Żadna

9.5.33 fsub

Funkcja:	void fsub (x,y,z) flash x,y,z;
Moduł:	mrflash.c
Opis:	Odejmuje dwie liczby flash
Parametry:	Trzy liczby flash x,y iż. Na wyjściu z= x -y
Zwracana wartość:	Żadna

9.5.34 ftan

Funkcja:	void ftan (x,y) flash x,y;
Moduł:	mrflsh3.c
Opis:	Oblicza tangens danego kąta flash, używając metody $O(n^{2.5})$
Parametry:	Dwie liczby flash x i y. Na wyjściu y=tan(x)
Zwracana wartość:	Żadna

9.5.35 ftanh

Funkcja:	void ftanh (x,y) flash x,y;
Moduł:	mrflsh4.c
Opis:	Oblicza hiperboliczny tangens danego kąta flash
Parametry:	Dwie liczby flash x i y. Na wyjściu y =tanh(x)
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.36 ftrunc

Funkcja:	void ftrunc (x,y,z) flash x,y; big y;
Moduł:	mrflash,c
Opis:	Rozdziela liczbę flash na liczbę big i resztę flash.
Parametry:	Liczby flash x i z i liczba big y. Na wyjściu y =int(x) i z jako reszta ułamkowa. Jeśli y jest takie samo jak z, tylko int(x) jest zwracane
Zwracana wartość:	Żadna
Ograniczenia:	Żadne

9.5.37 numer

Funkcja:	void numer (x,y) flash x; big y;

Moduł: mrcore.c
Opis: Wyodrębnia licznik liczby flash
Parametry: Liczba flash x i liczba big y. Na wyjściu y będzie zawierało licznik z x
Zwracana wartość: Żadna
Ograniczenia: Żadne

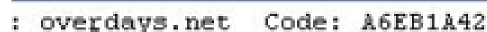
9.5.38 mround

Funkcja:
```
void mround (n,d,x)
flash x;
big n,d;
```
Moduł: mrround.c
Opis: Formuje zaokrągloną liczbę flash z licznika i mianownika big. Jeśli zaokrąglenie ma miejsce, instancja zmiennej EXACT jest ustawiona na FALSE. EXACT jest inicjalizowane na TRUE w podprogramie mirsys. Ten podprogram jest używany wewnętrznie.
Parametry: Liczba flash x i dwie liczby big n i d. Na wyjściu x=R{n/d}, to znaczy liczba flash n/d jest zaokrąglana jeśli to konieczne do odpoweidniej reprezentacji
Zwracana wartość: Żadna
Ograniczenia: Mianownik nie może być zerem

10 INSTANCJE ZMIENNYCH
Zmienne te, wszystkei są składnikami struktury miracl, zdefiniowanej w miracl.h. Są one dostepne poprzez

mip - Miracl Instance Pointer

BOOL EXACT; Inicjalizowana TRUE. . Ustawia FALSE jeśli jakieś zaokrąglenie miało miejsce podczas arytmetyki flash.
int INPLEN; „Drukowalna" podstawa liczbowa uzywana dla danych wejsciowych i wyjściowych. Może być zmieniana wewwnątrz programu. Musi być większa niż lub równa 2 i mniejsza niż lub równa 256
int IOBSIZ; Rozmiar bufora I/O
BOOL ERCON; Błędy generujące domyślne komunikaty o błędzie i bezpośrednie przerwanie programu.. Alternatywnie przez ustawienie mip ->ERCON=TRUE kontrola błędów jest pozostawiona użytkownikowi.
int ERNUM; Liczba ostatnio występujących błędów
char IOBUFF[ ]; Bufor Wejścia/Wyjscia
int NTRY; Liczba iteracji uzywana w probabilistycznym teście pierwszości. Inicjalizowany 6.
int *PRIMES; Wskaźnik do tablicy małych liczb pierwszych
BOOL RPOINT; Jeśli jest ustawiona na TRUE liczby są wyprowadzane z kropką pozycyją. W przeciwnym razie,są wyprowadzane jako ułamek (domyślnie)
BOOL TRACER; Jeśli jest ustawiony na ON powoduje wydrukowanie informacji o debuggowaniu, śledząc postęp wszystkich późniejszych wywołań podprogramów MIRACL. Zainicjalizowany na OFF



11 KOMUNIKATY BŁĘDÓW MIRACL

Komunikaty błędów, diagnoza i reakcja MIRACL

Komunikat: Number base too big for reprersentation
Diagnoza: Próba wprowadzenia lub wyprowadzenia liczby, której podstawa liczbowa jest zbyt duża. Na przykład wyprowadzenie przy użyciu podsatwy liczbowej 2^{32} jest najwyraźniej niemożliwe. Dla sprawności, jest używana największa z możliwych wewnętrznych podstaw liczbowych, ale liczby w tym formacie powinny być

wprowadzane/wyprowadzane w dużo mniejszej podstawie liczbowej ≤ 256. Ten błąd zazwyczaj bierze się, kiedy uzywamy innum(.) lub otnum(.) po mirsys(.,0)

Reakcja: Wykonujemy zmianę podstawy wcześniej przed wprowadzeniem / wyprowadzeniem. Na przykład ustawiamy instancję zmiennej IOBASE na 10 a potem używamy cinnum(.) lub cotnum(.). Unikając zmainy w podstawie liczbowej , alternatywnie inicjalizujemy MIRACL używając czegoś takiego jak mirsys(400,16),co używa wewnętrznej podstawy 16. Teraz I/O hex może być wykonanae przy zastosowaniu innum(.) i otnum(.). Zauważ, że nie będzie to wpływało na wydajność na 32 bitowym procesorze, ponieważ 8 cyfr hex będzie spakowane do każdego komputerowego słowa.

Komunikat: Division by zero attempted
Diagnoza: Zrozumiałe samo przez się
Reakcja: Nie rób tego!

Komunikat: Overflow - Number too big
Diagnoza: Liczba w obliczeniu jest zbyt duża do przechowania w jej zaalokowanej pamieci o stałej długości.
Reakcja: Określnie większej przestrzeni dla pamięci dla wszystkich zmiennych big i flash, przez zwiększenie wartości n w początkowym wywołaniu mirsys(n.b)

Komunikat: Internal Result is Negative
Diagnoza: Jest to wenętrzny błąd, który nie powinien wystąpić przy użyciu funkcji MIRACL wysokiego poziomu. Może być sposowdowany przez indukowanie pzrepełnienia pamięci przez użytkownika.
Reakcja: Raportuj do mike@compapp.dcu.ie

Komuikat: Input Format Error
Diagnoza: Liczba na wejściu zawiera jeden lub więcej niepoprawnych symboli w zwiazku z bieżącą podstawą liczbową I/O. Na przykład ten błąd może wystąpić jeśli IOBASE jest ustawione na 10 a wprowadzana jest liczba Hex.
Reakcja: Ponowne wporwadzenie liczby i bądź ostrożny używając tylko poprawnych symboli. Zauwąz, że dla wporwadzania liczb Hex dopuszcza wporwadzanie tylko duzych liter A-F

Komunikat: Illegal number base
Diagnoza: Podstawa liczbowa okreslona w wtywołaniu mirsys(.) jest niepoprawna. Na przykład podstawa liczbowa 1 nie jest uznawana
Reakcja: Używaj różnych podstaw liczbowych

Komunikat: Illegal parameter usage
Diagnoza: Parametry używane w funkcji wywołującej nie są uznawane. W pewnych przypadkach pewne parametry muszą się różnić - na przykład w divide(.) pierwsze dwa parametryu muszą odnosić się różnych zmiennych big
Reakcja: Przeczytaj dokumentację dla danej funkcji

Komunikat: Out of space
Diagnoza: Próba uczynienia przez funkcję MIRACL alokacji zbyt dużej pamięci na stercie
Reakcja: Zredukuj wymagania co do pamięci. Spróbuj użyć mniejszej wartości n w pocżatkowym wywołaniu mirsys(n,b)

Komunikat: Even root of a negative number
Diagnoza: Próba znalezienia , na przykład, pierwiastka kwadratowego z liczby ujemnej
Reakcja: Nie rób tego!

Komunikat: Raising integer to negative power
Diagnoza: Zrozumiałe samo przez się
Reakcja: Nie rób tego!

Komunikat: **Integer operation attemptef on flash number**
Diagnoza: Pewne funkcje powinny być używane tylko z liczbami big i nie mają sensu dla liczb flash. Zauważ, że ten komunikat błędu jest często wywoływany przez problemy z pamiecią,gdzie na przykład alokowana pamięć dla zmeinnej big jest przypadkiem nadpisywana
Reakcja: Nie rób tego!

Komunikat: **Flash overflow**
Diagnoza: Błąd ten jest wywoływany przez przepełnienie lub niedomiar flash, Wynik jest za zewnątrz reprezentowanego zakresu dynamicznego
Reakcja: Uzywaj większych liczb flash, Zanalizuj sój program ostrożnie pod kątem niestabilności liczbowej

Komuikat: **Numbers too big**
Diagnoza: Rozmiar liczb big lub flash żadane w w yołaniu mirsys(.) jest po prostu zbyt duży. Długość każdej big i flash jest kodowana do pojedynczego słowa komputerowego. Jeśli jest niewystarczająca ilosć miejsca dla tego kodowania, wystąpi ten komunikat błędu.
Reakcja: Budowa biblioteki MIRACL, która używa większego „odpowiedniego typu”. Jeśli nie jest uzywana arytmetyka flash, budujemy bibliotekę bez tego - pozwala to na użycie większych liczb big.

Komuikat: **Log of a non-positive number**
Diagnoza: Próba obliczenia logarytmu z nie dodatniej liczby flash
Reakcja: Nie rób tego!

Komunikat: **Flash to double conversion failure**
Diagnoza: Próba konwersji liczby flash do standardowego wbudowanego typu C double, zakończona niepowodzeniem, prawdopodobnie poniewważ liczba flasjh jest poza dynamicznym zakresem, który może być reprezentowany jako double
Reakcja: Nie rób tego!

Komuikat: **I/O buffer overflow**
Diagnoza: Niepowodzenie operacji wejścia wyjścia ponieważ bufor I/O ie jest dość duży.
Reakcja: Alokowanie większego bufora przez wywołanie set_io_buffer_size(.) po wywołaniu mirsys(.)

Komunikat: **MIRACL not initialise - no call to mirsys()**
Diagnoza: Zrozumiałe samo przez się
Reakcja: Nie rób tego!

Komuikat: **Illegal modulus**
Diagnoza: Moduł określony do wewnętrznego zastosowania dla redukcji Montgomery’ego jest nielegalny. Zauważ, że ten moduł nie może być parzysty
Reakcja: Używanie dodatnich nieparzystych modułów.

Komunikat: **No modulus defined**
Diagnoza: Żaden moduł nie został okreslony , a mimo to funkcja, któ®a go potzrebuje zostałą wywołana.
Reakcja: Ustawienie modułów do wewnętrznego zastosowania

Komunikat: **Exponent too big**
Diagnoza: Próba wykonania obliczenia przy użyciu preobliczonej tablicy, dal wykładnika (lub mnożnika w przypadku krzywych eliptycznych) większego niż ten, dla pre-obliczonej tablicy
Reakcja: Ponowne obliczenie tablicy, z większym wykłądnikiem lub użycie małego wykładnika

Komunikat: Number base must be power of 2
Diagnoza: Małą liczba funkcji wymaga aby podstawa liczbowa okreslona w poczatkowym wywołaniu mirsys(.) była potęga 2.
Reakcja: Użycie innej funkcji lub określenie potęgi 2 jako podstawy liczbowej w poćżtkowym wywołniu mirsys(.)

Komunikat: Specified double-length type isn't
Diagnoza: MIRACL okreslił, że długość typi double okreslona w mirdef.h nie jest faktycznie podwójną długością. Na przykład jeśli odpwoedni typ jest 32 bitowy, podwójna długość powinna wynosić 64 bity.
Reakcja: Nie rób tego!

Komuikat: Specified basis is not irreducible
Diagnoza: Posdstawa określona dla arytmetyki GF(2^m) jest nierdukowalna
Reakcja: Nie rób tego!