Residue Number System
General Purpose, Arbitrary Precision Arithmetic Library

Tutorial, Guide and Introduction to RNS-APAL
(RNS-APAL V0.10.2)

Document Version 1.02
June 7, 2016

(Originally Released June 5, 2016)

By:
Eric B. Olsen
of
Digital System Research, Inc

Contents

Document Version History

It is the goal of the RNS-APAL project to keep this document in track with the RNS-APAL code release. For example, if RNS-APAL is at Version V0.10.1, the matching guide document should be V1.01. We'll do our best to keep these version numbers matching.

V1.0:	June 5, 2016: Initial Release.
V1.01:	June 6, 2016: Added Header, minor corrections to content.
V1.02:	June 7, 2016: Corrected document to correctly describe corrected Prints() sign functionality; Lot's of format tune-ups and minor document tweaks. Document currently matches RNS-APAL 0.10.2

Introduction:

Arbitrary precision arithmetic libraries, such as MPIR and the GNU multiple precision arithmetic library (MPFR), are examples of programs that perform arithmetic in software, and perform this arithmetic well in excess of the native instruction set of the computer that the software runs on.

Multiple precision libraries are not trivial to write, and take time to debug and test. So imagine the task of creating an arbitrary precision library which performs arithmetic entirely in residue number format! The task is enormous to say the least. This brief manual describes the product of such an under-taking. To be clear, the library described performs both integer and fixed point arithmetic entirely in residue number format. This means there is no carry between digits, and this also means that calculations are totally modular, where general purpose calculations can be immersed in residue without conversion to a fixed radix system.

While integer based residue number arithmetic is not new, there are difficult problems in the field, such as performing arbitrary integer division in RNS. In addition, arbitrary precision residue arithmetic becomes difficult to analyze simply because an arbitrary length residue number is difficult to convert, and because the residue number may far exceed the normal arithmetic precision of the programming language that implements it. There are ways to solve this problem, such as using an arbitrary binary library to assist, but in this implementation, we adopt a stringent policy to perform all needed calculations in residue format entirely.

This arbitrary precision library described herein is the first I know which implements true fixed point and sliding point arithmetic in residue format. In fact, the library can demonstrate general purpose processing completely in residue format. That is, iterative residue calculations can operate entirely in the residue format until an answer in complete; only then is the answer required to be converted to decimal or binary format. These are new developments. For many RNS researchers, this may be difficult to understand at first. That is, that residue numbers can indeed perform calculations on fractional values, with very high precision, and yet do so in an iterative fashion, without performing a conversion to a weighted number system. The library also includes arbitrary conversion from residue fractional format to decimal fractional format that is also performed without carry. I hope you find these developments quite amazing as I do.

One thing worth mentioning is the enormous amount of effort and research required to get something like RNS-APAL up and running. From my experience it takes many different arithmetic operations to work before a full set of operations becomes viable. In residue format, even debugging these algorithms is a challenge. However, as one builds up enough resources, the work becomes more efficient and the effort proceeds more quickly. At the same time, the clarity of the structure of RNS arithmetic begins to un-fold, and that takes time to understand and assimilate. At some point, a completely working set of arithmetic code emerges. RNS-APAL is such a result, the result of years of effort.

The library as released may have some problems. It likely has some bugs, and is not formally tested. However, it has been shown to be stable enough, and therefore, it is hoped this initial release can spur the development of new work in this brand new field. Many of the methods were experimental and were deleted prior to this release, however, some are still included as work in progress. The "stable" methods are noted and demonstrated, and therefore, these are the preferred algorithms to use when experimenting with the library. As time goes, I hope to improve this work, and ultimately, create a completely tested and debugged piece of code, eliminating all un-necessary test code. Being that this finalizing task may take years, I have decided to release what I have now. To be truthful, I have many more library routines, but I've only released the most basic arithmetic functions for now, partly because of the complexity and size of the code, but also because DSR continues to use the code for development of residue based hardware processors.

Another issue with the library is the fact there is no overflow checking. Overflow checking in residue format is challenging, but not impossible. Therefore, in the future, revisions may aid in overflow and range checking. For this initial release, the user is cautioned regarding the range limitations of the particular residue system they choose to operate with. If the range of the arithmetic operation exceeds the range of the number system, the answer is erroneous. This is similar to most forms of integer arithmetic such as in the C/C++ language, and this is also true with most fixed fractional arithmetic. In comparison, floating point format supports advanced overflow detection, as such features usually require specialized hardware and the use of exponent formats.

We hope that you will find this brand new form of arithmetic fascinating, as you are free to use it for educational activities as well as for personal use. If you are interested in making commercial use of this software, or use hardware described in the referenced patents, please contact Eric Olsen, President, Digital System Research, Inc.

Speed and Optimization Notes:

This software was not developed for speed. Please don't expect blinding calculations. The slowest methods are actually the conversion methods, such as Print() functions. One reason Print() methods are slow is they are based on residue integer division, which is the slowest arithmetic operation in the library. If LUT support is enabled, the speed is faster, but at the expense of memory for look-up tables to store and perform inverse modular multiplication.

Other reasons for the slow speed of the library come from the style of the code design. For example, whenever a residue constant is needed, such as the value of a fractional unit, it is calculated, not simply read from a location. It is believed that translation of RNS-APAL methods into hardware will perform more reasonably. For example, the Rez-9 ALU, which is prototyped in an FPGA, is running at Megahertz clock speed, and therefore, calculating speed is more reasonable. In other words, the hardware equivalent of these methods scales better than expected. Moreover, more research and optimization to increase hardware speed is being made. Speed and memory optimization were not objectives when RNS-APAL was written; instead, residue arithmetic functionality has been the main consideration.

Downloading and Compiling the RNS-APAL software:

We call the work a library, but it has yet to be compiled as a library at the time of this writing. The zip file for RNS-APAL can be downloaded from the downloads page at www.digitalsystemresearch.com. Included in the zip file you will find a Microsoft project file which can be used to compiled the files. The files have been compiled using Microsoft Visual Studio Express 2010 or later. The files should also be easily compiled with other compilers with a few minor changes. The example demo software executes from a windows console at this time.

C++ code style:

The coding style is easy to understand, and I have not attempted any tricky functions, such as operator overloading, etc. I've worked to place updated comments throughout, and explain the functionality of each method as best possible. Some methods are prototypical, and may be improved in the future; some methods are only provided to show some different algorithm strategies during RNS-APAL development. Future releases of RNS-APAL will help improve speed, readability, and features.

Major Files of the RNS-APAL:

Main.cpp - contains the demo code of the RNS library.

config.h - this file helps the user configure the precision of the RNS number system by selecting the number of residue digits. Other advanced features are provided, such as allowing the user to select their own modulus. More information on this important file is given later.

ppm.cpp - contains the base integer class objects that process basic residue number integer arithmetic. PPM stands for "partial power based modulus" since the library allows for RNS numbers that use power based modulus, and also partial powers. A long time ago, it stood for "power prime modulus". Remember it anyway you wish!

mrn.cpp - contains a base class for mixed radix number system. This module is not fully developed, since the RNS library really only needs a few methods in this base class, and mainly uses this class to contain mixed radix numbers during intermediate calculations.

sppm2 - contains the sppm2 class which is derived from the ppm class and is used to perform signed integer arithmetic. This class stands for "signed partial power modulus". Obviously, it is used for signed integer arithmetic in residue format. Singed numbers use the method of complements along with two flags, a sign flag, and a flag to indicate whether the sign flag is valid, called a sign valid flag. Somewhere along the line, I decided to re-write this module, so its named with a "2" on the end.

spmf2 - this module contains the spmf2 class which is derived from the spmm2 class and is used to perform signed fractional arithmetic in residue format. This class stands for signed power modulus fractional. This class object is responsible for performing fractional arithmetic in residue format, and may be used to actually calculate complex problems indefinitely in residue format without conversion to the decimal or binary number system.

utilities.cpp - this module contains a number of helper functions.

Getting Started with RNS-APAL

The easiest way to demonstrate residue arithmetic is to run the "**demo.exe**" application included in the zip file. The demo program allows the user to perform some basic arithmetic operations, which are printed in residue format as well as decimal format.

This manual provides a more in-depth treatment of the RNS-APAL library. The user can cut and paste the examples included in the manual to help better understand the features of the arithmetic library. The user is also invited to study the source code. It is through the study of the source code that one adopts a complete understanding of the inner workings of general purpose residue arithmetic. Lastly, the user is pointed to US patent application US 2013/0311532 A1 which provides in-depth explanation of many of the methods covered in this library package.

Adjusting RNS-APAL using the *config.h* file:

The user can adjust the RNS number system that is used by the library. At the time of this release, the library must be compiled to adjust specific user settings. Future versions may provide the ability to dynamically select the RNS number system attributes during run time. This section discusses some of the settings found in the config.h header file which control the way the RNS number system is constructed. RNS has many degrees of freedom, so it is up to the user to decide which features and modulus are required for their residue number system.

One important define located in config.h is the define:

```
#define NUM_PPM_DIGS 18.
```

By changing the value of this define, the number of residue digits used during calculation can be specified. Another important define is:

```
#define SPMF_FRACTION_DIGS 6
```

which controls the number of digits associated to the fractional range of the SPMF type. For the number of fractional digits, these are included in the total number of digits specified in `NUM_PPM_DIGS.`

The ModTable class is a static class which contains initialized variables and arrays for the operation of the PPM class. One of the most important initialized arrays is the *prime array, which contains the modulus values used by the RNS number system. The primes array may be initialized automatically by setting the following #define statement in the *config.h* file:

```
#define CUSTOM_MODULUS 0
```

If initialized automatically, the modulus values of the RNS number system will take on prime numbers starting with the base modulus 2, and moving upwards to include all primes until the number of RNS digits is met, which is set by NUM_PPM_DIGS. To create a custom RNS number system, the user may define the primes array by initializing the modulus[] array contained in *config.cpp*. To do this, set the modulus[] array appropriately with pair-wise prime values, and then set the define:

```
#define CUSTOM_MODULUS 1
```

A power based RNS system is defined as an RNS system having low valued primes being a power of that prime, such powers usually limited by the digit word size of the RNS system defined as "q bits". More about a power based RNS number system may be found in patent application US 2013/0311532 A1, in figure 11D and Table 3. To create a power based RNS system, set the following #define in config.h as:

```
#define CUSTOM_POWERS 0
```

When using both automatically generated RNS numbers, *and* power based modulus, all of the methods defined in the application should work. Some methods, specifically fractional scaling and Goldschmidt division, require that a power based RNS system be supported. In addition, both operations require that the two's modulus be the largest modulus of the system, which means the two's modulus support 'q' number of powers, where q is the bit width of the largest RNS digit. The ModTable class supports and initializes a static member **q**, which again, is the word size of the largest digit of the RNS system.

ModTable may also initialize the **arrayDivTable** if:

```
#define USE_MODDIV_LUT
#define USE_BRUTE_LUT
```

is defined. The arrayDivTable supports inverse multiplication by look-up table, using a brute force technique.

A more improved LUT approach is supported by defining:

```
#define USE_MODDIV_LUT
//#define USE_BRUTE_LUT.
```

This is the default setting. In this case, **ModDivTbl** is initialized; this LUT is a more efficient inverse multiplication look-up table. Otherwise, if both:

```
//#define USE_MODDIV_LUT
//#define USE_BRUTE_LUT
```

are commented out, inverse multiplication is supported with a brute force search. It is recommended to keep the default settings, which uses RAM more efficiently, and makes RNS-APAL run faster.

The Fundamental Base Residue Class: PPM

The ppm.cpp module contains the base residue class, and therefore, the config.h file can be seen to affect this base class directly. The PPM modulus contains three basic classes; they are:

1) the ModTable class,
2) the PPMDigit class, and
3) the PPM class.

The PPMDigit Class

The PPMDigit class implements each particular digit of the RNS number system. Therefore, this class also implements each digit operation. The digits themselves are stored in an array within the PPM class to implement the entire residue number.

Several important variables are defined for PPMDigit object. In the future, such variables may be declared private, so for now, do not directly modify these variables unless you know what you're doing, or unless you want to perform unique tests. The key variables of the PPMDigit class are:

- **Index**: This is the position within the Rn[] array where the PPMDigit object will be referenced. The same index value is used to reference the associated values in the primes[] array, and also the arrayDivTable LUT table.
- **Digit**: stores the value of the RNS digit. The value can only be positive, even though it's of type integer.
- **DigitCopy**: Stores a copy of the digit. Used by the Restore method, but this method may be deprecated in later revision. Instead of Restore method, copy the value by using the Assign method.
- **Skip**: If non-zero, then the digit is skipped, or undefined. This occurs quite frequently, such as when an RNS number is divided by a modulus value (inverse multiplication by a modulus value). After such an operation, the digit associated with the modulus will be un-defined, or skipped. Base extending an RNS number will restore all skipped digits to their correct value. The correct value is the value of the digit required to maintain the original RNS number's value (before the base extend operation). By maintaining skip digit flags, dynamic RNS number systems are supported by the library. (i.e., a unique RNS number system is uniquely defined by its set of modulus, and changing the modulus set changes the format of the RNS number system.) To learn more about skipped digits, refer to patent application US 2013/0311532 A1.
- **Modulus**: This is the "base modulus" of the digit. The base modulus is the n^{th} root of a power based modulus of n powers. For the digit modulus=64, the **Modulus** variable will be equal to two (2).
- **Power:** This is the power of the digit modulus. The maximum (normal) modulus of the digit is equal to Modulus^Power.
- **PowerValid**: This variable defines how many powers of the modulus are valid at any given time. This variable provides the ability for a digit modulus to take on a variable modulus functionality, sometimes referred to as "partial powers" in this reference. A Modulus power can change when an RNS value is divided by a base modulus value. In this case, the modulus of the digit matching the base modulus dividing the RNS value may be decreased by a single power. The value of a power based digit modulus at any instant is given by **Modulus^PowerValid**. To restore the digit modulus to full power, a base extend is executed, after execution, the **PowerValid** value will equal **Power**. To learn more about variable power

modulus operation, refer to patent application US 2013/0311532 A1, particularly sections covering fractional scaling and integer division.

- **NormalPower:** This variable is essentially a constant, and is assigned the original Power of the modulus of the base class. Using this value, a modified partial power RNS number system may be derived, and then returned back to normal format.

The PPMDigit also includes important methods. These methods perform the necessary "digit operations". Digit operations perform an arithmetic operation on one or two digits where these operations are essentially modular, and do not support carry to other digits. All of the basic digit operations now recognize partial power based modulus. These most fundamental digit methods are:

- Add() - This method performs a modular addition by adding a value to the digit object.
- Sub() - This method performs a modular subtraction, that is, it subtracts an argument value from the digit object.
- Mult() - This method performs a modular multiply by multiplying a value to the digit object.
- Div() - This method performs an inverse modular multiplication of a value with the digit object with respect to the digit objects modulus. The term "Div" is loosely used, and strictly is not division, but actually inverse multiplication.
- DivPM() - This method is the same as the Div method, but uses only table look-up. During development, the DivPM method was designed to handle the new "partial power" based modulus feature; however as development proceeded, the Div procedure, and many other procedures, adopted the same logic to handle partial power modulus. Therefore, this method is no longer distinctive, and will likely be deprecated in the future.

Other methods in the PPMDigit class perform other important functions, such as returning state information:

- GetDigit() - returns the digit value of the digit object.
- Get Modulus() returns the base modulus of the digit class. It also returns the current power of the modulus.
- GetFullPowMod() - returns the full (normal) power of the digit modulus.
- GetPowMod2() - returns the present power of the digit modulus.
- GetPowOffset() - returns the digit value that when subtracted from a residue number makes the digit evenly divisible by the specified power.
- SkipDigit() - sets the digits skip digit flag, and therefore marks the digit as skipped. When skipped, the digit is invalid, and is ignored until base extended.
- ClearSkip() - clears the skip digit flag of the digit. It is called by base extension routines.
- ResetPowerValid() - sets the power valid count of the digit to its normal value. This resets the modulus of the digit to its full power. The full power is defined by the Power variable, and therefore, this routine sets the modulus back to its full "derived" power. This routine is also called by base extension routines.
- SetPowerNormal() - sets the power valid count of the digit to its normal (full) power. The normal power is the value of the original base class RNS number as originally defined using config.h.
- TruncMod() - truncates (reduces) the power of the digit modulus by the number of specified powers. The method also re-adjusts the digit value in accordance to this truncation.
- IsZero() methods - These methods are used in integer division and return true if the digit is "zero". If a residue number is non-zero, but a digit of the residue number has a zero in a particular digit position, then the number is evenly divisible by the modulus of that digit position. This fact forms the basis of the integer division used.

The IsZero() method is more complex when partial power modulus are used, since this method returns a "zero" if the digit is evenly divisible by any power of the modulus. The name "is zero" might be better remembered as "evenly divisible by".

The Base PPM Class Object

The PPM class object constitutes the main RNS number supported by RNS-APAL. The PPM object is essentially an array of PPMDigit objects contained within a vector array, called Rn[]. The variable name "Rn[]" stands for "Residue number". The class name PPM stands for "partial power modulus".

The PPM class acts as the base residue number for all other derived RNS number types in RNS-APAL. It can be thought of as the underlying *hardware register* holding any residue number type. In other words, the base PPM number type is a raw residue number, it is an unsigned integer type, and it serves as a basic type which may be used to support primitive operations of other residue number types. Other RNS number types, such as signed integers, are derived from the PPM base class. The number of digits supported by the PPM class is defined using `#define NUM_PPM_DIGS` located in config.h.

A couple of things can be said about the PPM object implementation. The PPM object allows the user to define all modulus of the PPM object using the *config.h* and *config.cpp* files. Alternatively, the user can let RNS-APAL auto-generate the digit modulus, which is preferred for basic study. The RNS-APAL library, and in fact, the entire mathematics exposed by the library, rely on several key ideas. For one, the existence of a two's modulus is fundamental for both division and fractional scaling. Secondly, the PPM class is designed after the notion of a "natural residue number system". The natural residue number system was defined in US patent application US 2013/0311532 A1, and essentially consists of a residue number system having prime digit modulus which include every prime number starting with two. With this definition, one can define any specific natural residue number system by simply specifying the number of digits needed (using `NUM_PPM_DIGS)`. This definition also greatly aids mathematical analysis.

A natural residue number system can also be extended by increasing the powers of the prime modulus. One way to do this is to determine the number of bits required to represent the largest prime digit modulus in a given natural residue number sequence. This bit width is referred to as "Q". By increasing the power of every prime modulus to the largest power that fits within Q bits, a "full power based residue number" system is formed. The full power based residue number system has strong properties for performing integer division; such a system can also represent very accurate fractional quantities. Refer to patent application US 2013/0311532 A1 for lengthy discussion of these topics.

The PPM constructor:

Instantiation of a PPM object is essentially the same as declaring an unsigned integer type in RNS-APAL. The main constructor requires an __int64 argument, which allows the user to initialize the integer variable to a value within the __int64 range. Note however, that only the unsigned range of the __int64 argument makes sense for the base PPM class, since the base PPM class only stores positive integers. We declare a variable of type PPM using:

```
        PPM *x = new PPM(100LL);
```

The statement above creates a PPM variable called "x", and assigns the value of 100 to it. We recommend that all instantiation of PPM variables be allocated using the pointer method above, since all PPM arguments in the library are pointers to PPM objects.

Another more advanced PPM constructor takes the form of:

```
        PPM *derived = new PPM(PPM *ppm, __int64 x);
```

In the example above, the constructor call creates a PPM variable called "derived", and assigns it the modulus properties of the PPM argument, and the value of the __int64 x variable. If x equals -1, the value of the ppm argument is assigned to the derived PPM variable. The derived variable assumes any modified power properties of the argument. This is an advanced procedure, and is not needed for general RNS-APAL use.

Assigning values to PPM variables:

To assign a value to an existing PPM variable, the Assign method is used. This is accomplished using the following C++ code:

```
        x->Assign(100);
```

or

```
        x->Assign(100LL);
```

The first instance of Assign uses an overloaded version which takes an integer argument. The second instance uses an overloaded version that takes an __int64 integer argument. Be careful, if you need to initialize your PPM variable with large values, then be sure to end those values with the "LL" specifier as shown.

You may want to assign an integer larger than an __int64 type. If so, then you may use a string input and specify an integer as large as you wish. This method looks like:

```
        x->Assign("12345678901234567890");
```

Keep in mind that in order for the assign to succeed, the PPM type must have a range large enough to contain the integer. You may increase the range of the PPM type by specifying more digits using the NUM_PPM_DIGS define in config.h

You can also assign an existing PPM value to a PPM variable, using the overloaded Assign which takes a PPM * pointer as an argument. This version acts much like an equal sign:

```
        PPM *x_copy = new PPM(0);
        x_copy->Assign(x);
```

The command above copies the value of the PPM variable x to the PPM variable x_copy. It does not alter the normal power modulus (which is an advanced feature anyways). Use this method for basic assignment of one PPM variable to another for applications.

Advanced Assign Methods

One side effect of the basic Assign(int x) and Assign(__int64 x) methods is that they reset the PPM variable to a "full power based" number format. If the user is experimenting with advanced operations, there may be need to use the AssignPM() method. Depending on which overload is used, the

AssignPM() method serves to support variable power modulus variables, which were derived. If using a variable power variable, and the user needs to assign an integer value, the following might be used:

```
x->AssignPM(1234567890LL);
```

In the example above, the variable x may have a modified power in one or more of its modulus. Using the AssignPM(__int64 x) method will not revert the variable x modulus back to its normal settings. What if the user wants to assign a PPM variable y, which has a modified power modulus, to another PPM variable? The user can use the AssignPM(PPM *ppm) method:

```
x->AssignPM(y);
```

In this case, the PPM variable x assumes the variable power modulus format of y, as it must.

Note: Variable power based modulus is an advanced concept, and can be confusing. However, during normal use of the RNS-APAL software, there is no need to worry about the power of the digit modulus. The reason is that each high level method used in the library works on fully *normalized* formats, and returns fully normalized formats. Therefore, the user need not be concerned about this feature. However, for researchers looking to understand the inner workings of the integer divide algorithms, the fractional scaling methods and Goldschmidt division, the use of variable power modulus comes into play, and side effects of the Assign method is therefore important to understand.

Printing a PPM variable in native format

One way to determine if a number is properly assigned to a PPM variable is to print that variable. The RNS-APAL library provides several options for printing a PPM variable. The most primitive method of printing a PPM variable is to print the variable in its native residue format using the "print string", or Prints() method. To print the native residue number, use the following:

```
cout << "The raw RNS number in decimal: " << x->Prints(DEC) << endl;
cout << "The raw RNS number in hexadecimal: " << x->Prints(HEX) << endl;
```

The Prints() routine prints modulus in the Rn[] array in order of increasing index. The problem with the format above is that you may not know the modulus of each digit. To more easily see the modulus associated with each digit, the PrintDemo() routine can be used:

```
cout << "The raw RNS number in decimal: " << endl;
x->PrintDemo(10);
```

Note that the PrintDemo() routine does not output a string, instead, it must be placed on its own line. The output of the PrintDemo routine for an 8 digit RNS number having the value 100 is:

```
32 27 25 7 11 13 17 19
-- -- -- - -- -- -- --
4  19 0  2 1  9  15 5
```

In the output above, the first row shows each digit modulus. In this example, the first modulus is 32, and the last digit modulus is 19. The second row is a header line acting to separate modulus values from digit values. The third row is the value of the digit printed beneath each respective digit modulus. The

PrintDemo routine can also display the modulus and digit values in hexadecimal and binary format, among other radix formats.

A more advanced form of this routine is called PrintDemoPM. This routine can indicate whether a modulus is a partial modulus, or whether a digit is skipped. In the following example, the value 100 is "divided" (using inverse multiplication) by the base modulus 2, and then divided by the full modulus value 25, and then printed using the following code:

```
x->Assign(100);
x->ModDiv(2);
x->ModDiv(25);
x->PrintDemoPM(DEC);
```

The output of the code above produces the output:

```
16 27 25 7 11 13 17 19
== -- -- - -- -- -- --
2  2  *  2 2  2  2  2
```

Using the PrintDemoPM method, several distinct features are evident. The first feature is the substitution of a double line header for a single line header under the digit modulus=16. This double line indicates that the original (normal) modulus has been modified (reduced). For this example, the digit modulus 16 has been reduced from its original normal modulus 32.

For the digit modulus 25, the digit value is un-defined which is indicated by an asterisk in the digit value position. Whenever a simple inverse multiplication procedure is used to divide out an RNS value by the value of a full modulus, the digit becomes undefined. (The multiplication of an RNS value by a multiplicative inverse of a modulus is only valid if the digit of that modulus is zero. This manual refers to such a multiplication as MODDIV, or "modular division").

By the way, if we use PrintDemo instead of PrintDemoPM for the example above, we still get an asterisk for the skipped digit, and the base=2 modulus still reflects the current power of the digit modulus (16); but it doesn't indicate the reduction of the digit modulus using a double line. It will look like:

```
16 27 25 7 11 13 17 19
-- -- -- - -- -- -- --
2  2  *  2 2  2  2  2
```

Don't worry about this detail. RNS numbers with partial power modulus are advanced, and normally used in internal calculations, or by advanced RNS-APAL users.

Printing a PPM variable in fixed radix format

Obviously, printing a native RNS value is not very friendly for humans. Often, the user will want to print the value of an RNS number in a decimal or hexadecimal *fixed radix* format. RNS_APAL makes this process easy. The following code will print the decimal value of the integer RNS variable x:

```
PPM *x = new PPM(1234);
cout << "The decimal value of x is: " << x->Print(10) << endl;
```

The output of the code above will produce:

```
        The decimal value of x is: 1234
```

We can also print to a hexadecimal or binary fixed radix format. The argument of the Print() method accepts the radix of the number system to convert to and print. Several simple defines have been provided to make the argument more readable; they are DEC for decimal, HEX for hexadecimal, and BIN for binary. Attempting to print a radix that is not supported results in an error. Right now, the most popular radix supported is decimal, hexadecimal and binary. The following code:

```
        PPM *x = new PPM(1234);
        cout << "The decimal value of x is: " << x->Print(DEC) << endl;
        cout << "The hexadecimal value of x is: " << x->Print(HEX) << endl;
        cout << "The binary value of x is: " << x->Print(BIN) << endl;
```

The output of the code above produces:

```
        The decimal value of x is: 1234
        The hexadecimal value of x is: 0x4d2
        The binary value of x is: b:10011010010
```

Note that decimal numbers are printed as decimal digits only, hexadecimal numbers are preceded with a '0x' and include decimal digits and digits 'a' through 'f', and binary numbers are preceded with 'b:' and include binary digits '1' and '0'.

Converting a PPM variable

Often it is necessary to convert a PPM variable to a common C++ integer type. RNS_APAL provides the "Convert()" method to do this. The Convert routine returns a __int64 value (which is "long long" type), and because PPM variables are only positive, the Convert() routine should only be used when the PPM variable value is within the positive range of the __int64 variable. The following code shows how to use the Convert() routine:

```
        PPM *x = new PPM(1234);
        long long y = x->Convert();
        printf("conversion of x is: %lld\n", y);
```

When executed, the code produces the following output:

```
        conversion of x is: 1234
```

To address the issue of returning a signed type, the RNS_APAL library now supports a method called "uConvert()". This method is same as Convert(), but returns an unsigned long long type. The following code demonstrates using this method:

```
        PPM *x = new PPM(1234);
        unsigned long long y = x->uConvert();
        printf("conversion of x is: %llu\n", y);
```

Basic PPM Arithmetic Methods: Add(), Sub(), Mult()

The PPM class supports arithmetic operations on PPM integer types. These arithmetic operations include addition, subtraction, multiplication and division. There are two basic method overloads supported, they include an overload with arguments of 1) an integer type, and 2) a PPM type.

The integer argument overloads are intended to be used to add small integer offsets to a PPM variable, or more commonly, to add the value of another PPM or MRN *digit* to a PPM variable. These overloads may also be used as *helper functions* during code development. The PPM argument versions are intended to be used to perform general purpose (unsigned) integer arithmetic in residue format. These versions simulate how a hardware ALU will perform *register to register* residue arithmetic.

To add any two PPM types, the Add method can be used as shown in the following code:

```
PPM *a = new PPM(100);
PPM *b = new PPM(123);
cout << a->Print(DEC) << " + " << b->Print(DEC) << " = ";
a->Add(b);
cout << a->Print(DEC) << endl;
```

The output of the code above is:

```
100 + 123 = 223
```

Note that the structure of the PPM arithmetic methods support a single argument. In all cases, a PPM variable is operated on by calling the arithmetic operation using the calling variables pointer, "this", and passing a single argument to the desired operation. The result of the calling structure is the arithmetic methods look a lot like assembly instructions of an accumulator based CPU; for example, an accumulator based CPU will normally perform an arithmetic Add operation on the accumulator, and therefore need only specify a single operand register, adding this register contents to the accumulator. Future versions of the RNS-APAL library will consider adding dual argument methods, and will also consider using overloaded C++ operators.

Two other easily understood arithmetic methods are the integer subtraction and integer multiply methods. The following code shows how these basic operations work in the context of RNS arithmetic application code. In this example, in config.h, NUM_PPM_DIGS is set to 8:

```
PPM *x = new PPM(100);
PPM *y = new PPM(20);
PPM *z = new PPM(5);
x->Sub(y);                  // subtract 20 from 100
x->Mult(z);                 // multiply 5 by result of above
x->Mult(x);                 // square the result of the next line above
cout << "the answer of the calculation is: " << x->Print(DEC) << endl;
```

When we run the code, we get the output:

```
the answer of the calculation is 160000
```

PPM arithmetic with skipped digits

All three basic integer operations, Add(), Sub(), and Mult() take a single argument, and pass any "skipped digits" of the argument directly to the calling variable as skipped digits. In other words, invalid digits of the argument will invalidate the same digit position in the calling PPM variable. If enough "non-redundant digits" survive a given operation, then the result of the operation will hold. However, if any operation invalidates more digits than needed to correctly represent its result, the result is erroneous. Manipulating PPM values having skipped digits is an advanced feature, as the novice user does not need to worry about this feature.

To demonstrate the arithmetic computation of values having skipped digits, the code above is modified:

```
PPM *x = new PPM(100);
PPM *y = new PPM(20);
PPM *z = new PPM(5);
x->Rn[4]->SkipDigit();            // for x, invalidate the digit at index 4
y->Rn[5]->SkipDigit();            // for y, invalidate the digit at index 5
z->Rn[6]->SkipDigit();            // for z, invalidate the digit at index 6
x->Sub(y);                        // subtract 20 from 100
x->Mult(z);                       // multiply 5 by result of above
x->Mult(x);                       // square the result of the line above
printf("the native RNS result is:\n\n");
x->PrintDemo();
cout << "the answer of the calculation is: " << x->Print(DEC) << endl;
```

When run, the output of the code above shows the native PPM value, and also prints the value of the native PPM value, which is still the same as the previous code example:

```
the native RNS result is:

32 27 25 7 11 13 17 19
-- -- -- - -- -- -- --
4  19 0  2 *  *  *  5

the answer of the calculation is 160000
```

In the output above, the asterisk indicates that the digit position is invalid. During execution, the x PPM variable started with an invalid digit at index position 4 (the digit of modulus 11). When the y PPM variable is subtracted, which has a skipped digit at index position 5, the Sub() operations passes the skipped digit position of digit index 5 to x (modulus 13). Finally, when the PPM variable x is multiplied by the z PPM variable, which has a skipped digit at digit modulus 17, it's skipped digit is passed to the x PPM variable. When x is squared, no new skipped positions are passed to x by operating on itself.

The calculation remains valid (result = 160000) because the remaining non-skipped digit modulus, namely the modulus 32, 27, 25, 7, & 19, still represent a range > 160000, since 32*27*25*7*19=2872800, and 2872800 > 160000. If we multiply our result by the value 18, the result will not be correct, since the correct value, 2880000, exceeds the maximum range 2872800. In this case, we have exceeded the range of all valid digits and we should not expect to contain this result correctly.

Basic PPM Arithmetic and Partial Power Modulus

The three basic arithmetic PPM operations, Add(), Sub(), and Mult() check that the argument and the calling PPM variable have the same "PowerValid" digit modulus structure. If they don't, the operation will halt with an error.

This check is now strictly enforced (before it was not) to reduce confusion and the chance of difficult to track errors when using the library. Later, we'll introduce methods on how to derive a PPM variable with a matching modulus structure so that basic arithmetic operations work. For basic users, there is no reason to worry about this detail if you will not be directly manipulating PPM variables having partial power modulus.

The PPM ModDiv Method

We covered the three basic PPM operations, and so you might be asking, what about the divide operation? Before I begin, let me state a few things. For one, integer division is known to be a complex operation in RNS, so we are in new territory here. Secondly, I've made some basic changes to the *naming conventions* of certain operations versus what is commonly found in the field of RNS research.

In the last section, we dealt with so called "PAC" methods. PAC stands for *parallel array computation*, which indicates that if the operation was performed in hardware, such arithmetic operations require only a single step, without carry. It is PAC operations which have excited researchers in the field of RNS, since it provides promise that arithmetic operations could be made faster, require less power, and contain its own form of redundancy.

So the first question that might be asked is whether there is a type of PAC division. The answer is yes, but with qualifications. In the field of RNS research, such a PAC division operation is referred to as "inverse multiplication", which is carried out by multiplying a value by the "multiplicative inverse" of the "dividing" value. Inverse multiplication may or may not result in a valid result depending on the mathematical properties of the values themselves, and also on the properties of the chosen RNS number format. Inverse multiplication is the domain of mathematics, namely, *modular arithmetic*, and such mathematics and history is well outside the scope of this manual. However, the operation of inverse multiplication is vital to the operation of the RNS-APAL library.

To simplify the explanations, RNS-APAL uses inverse multiplication for providing an arithmetic method for performing "PAC division". I call this method "ModDiv", which stands for "modular division". (Professional researchers would not use the term "division", but I do!) There are a couple of mandatory conditions for using the ModDiv() arithmetic method. For one, the ModDiv() method is only valid when dividing a PPM value by one of its digit modulus, or one of the partial powers of the digit modulus. Furthermore, the ModDiv() method can only be applied when the PPM value contains a "zero digit" in the position of the dividing modulus. Well, that's a mouthful.

What this really means is that before using ModDiv(), the PPM value must be *known beforehand* to be evenly divisible by the divisor (modulus). This might seem silly, but it is quite useful. It is easy to determine by inspection if an RNS number is evenly divisible by one of its modulus, since that digit will be zero. It is easy to make any RNS value *divisible* by one of its modulus because we can always subtract that modulus' digit value from the PPM value. This process forms the underlying operation of *mixed radix conversion*, where each digit subtracted forms a mixed radix digit, converted least significant digit first. Applying this procedure repeatedly until the PPM value goes to zero will convert the entire PPM value to mixed radix format.

ModDiv() is a simple function to understand once we show a few examples. So let's perform a few examples, again with NUM_PPM_DIGS in the file *config.h* set to 8. Consider the following code:

```
x->Assign(12*7);
printf("the native PPM value is:\n");
x->PrintDemo(DEC);
x->ModDiv(7);
printf("after ModDiv, the native PPM value is:\n");
x->PrintDemo(DEC);
cout << "and its decimal value is: " << x->Print(DEC) << endl;
```

The output of the code above is:

```
the native PPM value is:
32 27 25 7 11 13 17 19
-- -- -- - -- -- -- --
20 3  9  0 7  6  16  8

after ModDiv, the native PPM value is:
32 27 25 7 11 13 17 19
-- -- -- - -- -- -- --
12 12 12 * 1  12 12 12

and its decimal value is: 12
```

Note that the PPM variable x is assigned a value 12*7=84. The value 84 has a single factor of 7 and so it is evenly divisible by 7. As shown by the native PPM print, the modulus 7 digit is equal to zero. Therefore, it is legal to divide the PPM value by 7, since 7 is a digit modulus used in our PPM variable. When we perform the ModDiv(7) arithmetic method, the value is transformed as shown above. A few things are worth noting. For one, the digit position at modulus 7 is now "skipped", or undefined. This always happens as a result of ModDiv, that is, dividing a PPM value by a full modulus value will make that digit position invalid. Another thing we notice is that the value of the PPM is now 12, which is what it should be given that we divided by 7.

Now let's divide the same starting PPM value (84) by the value of 4. In this case, the value of the modulus 32 digit is not zero since 84 is not evenly divisible by 32, the value 32 being the full value of the base two modulus. However, the PPM variable x is divisible by two *powers* of the base modulus 2. Therefore, the ModDiv(4) operation is legal. Here is the code that illustrates this operation:

```
x->Assign(12*7);
printf("the native PPM value is:\n");
x->PrintDemo(DEC);
x->ModDiv(4);
printf("after ModDiv, the native PPM value is:\n");
x->PrintDemo(DEC);
cout << "and its decimal value is: " << x->Print(DEC) << endl;
```

The output of the code above is:

```
the native PPM value is:
32 27 25 7 11 13 17 19
-- -- -- - -- -- -- --
20 3  9  0 7  6  16  8

after ModDiv, the native PPM value is:
8 27 25 7 11 13 17 19
- -- -- - -- -- -- --
5 21 21 0 10 8  4  2

and its decimal value is: 21
```

Note that the output is correct, namely, the result of dividing 84 by 4 is 21. However, also notice that the two's base modulus digit is not skipped; instead, the two's base modulus has been reduced from 32 to 8. In other words, the *power* of the two's base modulus has been reduced by two, from 2^5=32 to 2^3=8 since we divided the PPM value by two powers of the base modulus 2. This reduction in the power of the base two's modulus is referred to as a partial power modulus. The resulting PPM number is now a new number system, and has been derived from the original full power modulus PPM value, or "normal" PPM value. (We can also consider the upper powers of the derived PPM value to be skipped, or ignored, although we do not use that terminology here).

Note that we cannot divide the remaining PPM value (21) by any power of two. The reason is the remaining value is not *evenly divisible* by any power of two. But what if we wanted to divide the remaining PPM value by the value 4 again? There are few helper functions that let us do this. Here is the code to perform such an operation:

```
x->Assign(12*7);
printf("the native PPM value is:\n");
x->PrintDemo(DEC);
x->ModDiv(4);
printf("after ModDiv, the native PPM value is:\n");
x->PrintDemo(DEC);
cout << "and its decimal value is: " << x->Print(DEC) << endl;

int offset = x->Rn[0]->GetPowOffset(2);
cout << "subtracting PPM value by the offset: " << offset << endl;
x->Sub(offset);
x->ModDiv(4);
printf("after another ModDiv, the native PPM value is:\n");
x->PrintDemo(DEC);
cout << "and its decimal value is: " << x->Print(DEC) << endl;
```

The output of the code above is:

```
the native PPM value is:
32 27 25 7 11 13 17 19
-- -- -- - -- -- -- --
20 3  9  0 7  6  16  8
```

```
after ModDiv, the native PPM value is:
8 27 25 7 11 13 17 19
- -- -- - -- -- -- --
5 21 21 0 10 8  4  2

and its decimal value is: 21

subtracting PPM value by the offset: 1
after another ModDiv, the native PPM value is:
2 27 25 7 11 13 17 19
- -- -- - -- -- -- --
1 5  5  5 5  5  5  5

and its decimal value is: 5
```

In the modified code example above, a second ModDiv() operation is possible because we subtracted the PPM value (21) by enough to make it evenly divisible by two powers of the base modulus 2. In this case, subtracting the value of 1 does the trick. But we cannot in general determine *by inspection* how much to subtract from the PPM value to make it evenly divisible by 4, so we used an interesting helper function called "GetPowOffset(2). The argument for the GetPowOffset() function is the number of powers to round to, and so we specified 2 since we want to divide by 2^2. The GetPowOffset() function is actually a member of the base two modulus digit class PPMDigit. So we must know the index of the twos base modulus digit to get the right helper function. Also note that at the end of the second ModDiv(4), the resulting base two modulus is equal to two, or 2^1. The complete calculation made by the code is: ((84/4)-1)/4 = 5.

The ModDiv function is simple, but its use is more advanced than using the simple Add(), Sub(), or Mult() arithmetic operations. ModDiv() and its associated helper functions comprise the heart of more complex operations, such as base extension, and arbitrary division.

The arbitrary integer divide PPM method

So I explained the PAC version of digit divide, but noted it has limitations. Is there a method that can perform a division by any value? The answer is again yes! This type of division is referred to as *arbitrary integer division*, but it is not a PAC operation. Arbitrary integer division in residue format is one of the most complex arithmetic operations, and has been a long time subject of study in the field of RNS arithmetic. I can happily say that within the context of RNS-APAL, it is now a solved problem. We'll discuss later what this means.

In this manual, we refer to arbitrary integer division as a "slow operation". In other words, arbitrary integer division requires multiple clock cycles, even in hardware. It is not a PAC operation. Funny thing, we also refer to binary integer division as somewhat complex, and it is also known to be a slow operation among other binary operations of Add, Subtract and Multiply. The simple fact is that arbitrary division is complex and time consuming, regardless of the number system used.

The focus of this manual is to show how to use the RNS-APAL library, and not to disclose and discuss the inner workings of all its methods. Therefore , I will mention a few things about its implementation, but move quickly to show how to use the method, which is very easy. Also, let me say that the integer division method is unique, and is disclosed in patent application US 2013/0311532 A1. This patent reference discusses the method in detail, and describes and proposes variations of the algorithm.

OK, as for what it means to have solved the problem of arbitrary integer division, there is a few caveats. As mentioned in the introduction section of this manual, the RNS-APAL library is based on the concept of a "natural residue number system". This definition is a residue number system comprising P number of prime modulus starting with the prime number 2, and including all successive prime numbers until P number of digits is reached. There is no penalty for using powers of the prime modulus, in fact, it's even better to do so.

The value of defining a natural residue system is immense, as it defines a mathematical standard for all residue numbers, and this allows much easier mathematical analysis. However, it has other benefits as well. It turns out that there is an advantage to having digit modulus comprised of low value primes. The reason is simple: most integers are evenly divisible by small prime numbers. If this were not true, my integer division methods would not work.

So a word of caution: if you want the integer division method in RNS-APAL to work, you must work with an RNS number system that has a few properties. For one, it is strongly suggested, even mandated, that a base two modulus is included. From there, it is strongly recommended to have a few more low valued base modulus, such as the base modulus 3 and the base modulus 5. Is it possible to break these rules and have the integer divide work? Maybe, but there is a point where the integer divide method will break. More research into what modulus values are necessary to ensure stable operation is needed.

Interestingly, my research is not focused on special residue systems, such as the famous three modulus residue system having modulus values of the form: 2^{n-1}, 2^n, 2^{n+1}. In this system, if we select n=6, then we have the three modulus values: 63, 64, & 65. It is interesting to note that these three modulus are divisible by 3, 2, and 5 respectively. This meets the basic requirements of the RNS-APAL integer divide routine. In all the time I've spent writing RNS-APAL, I've never tested this modulus set, but the library does allow you to do so. In summary, you can create a custom modulus system in RNS-APAL by setting the correct #defines and array values in config.h and config.cpp. If you choose to use an auto-generated RNS, then you'll get a power based, natural RNS having NUM_PPM_DIGS number of digits which works with the integer divide methods in RNS-APAL. A natural non-power based RNS should also work.

So why would I not study the famous three modulus RNS? The answer is I prefer to work with *extendable* RNS number systems. In fact, I like infinitely extendable residue number systems simply because binary and decimal are also infinitely extendable. The natural residue system provides such a system. Also, there is no penalty to making an extendable natural residue number system a "power based" residue system. This is the RNS I prefer to work with.

In an earlier section, I describe the process of finding the largest full modulus of a given RNS number system, and determining the minimum number of binary bits needed to represent the largest modulus. Next, I propose extending the power of every other modulus to a maximum power which would be represented in the same number of binary bits. I refer to this RNS as a full power based residue system, and I refer to the binary bit width as the RNS *digit width*, or Q, of the number system. Q is an important theoretical number. More information as to significance of Q can be found in one of my papers, entitled: Introduction of the *Residue Number Arithmetic Logic Unit With Brief Computational Complexity Analysis*, which is published on the arXig.org repository, and may be found at arxig.org/abs/1512.00911.

OK, enough with the caveats, conditions and bloating, let's do some integer division in RNS! The following code performs an arbitrary integer division using our simple 8 digit residue number system:

```
PPM *x = new PPM(12345);
PPM *y = new PPM(29);
PPM *rem = new PPM(0);
cout << x->Print(DEC) << " divided by " << y->Print(DEC) << " is ";
x->DivStd(y, rem);
cout << x->Print(DEC) << " with remainder " << rem->Print(DEC) << endl;
cout << "the native PPM result looks like: " << x->Prints(10) << endl;
wait_key();
```

The output of the code above is:

```
12345 divided by 29 is 425 with remainder 20
the native PPM result looks like: 9 20 0 5 7 9 0 7
```

The PPM integer division method is called DivStd; it has the form: *int DivStd(PPM *divisor, PPM *remainder).*

Therefore, the integer divide method takes a PPM type divisor, and it also takes a pointer to contain the PPM remainder result. The quotient replaces the dividend in the calling PPM variable as shown. I use the name DivStd() because of the need to test with other variations of the integer divide method. Right now, the DivStd() method calls the DivPM5() method, which is a new and unique method at the time of this writing. One of its strengths is the ability to perform arbitrary integer division *without* requiring any redundant digits. This is of important significance, since the integer division may constitute a primitive operation for other complex operations. The ability to divide out the entire native residue format makes the coding of other complex operations much simpler.

A few last words for the integer divide method. There is interest in the study of this integer divide and other divide methods. Namely, to understand how they work, and to optimize their performance. One simple way to "look inside" the integer operation is to enable its *display trace*. The display trace is a somewhat cryptic dump of symbols and numbers which represent the steps taken by the integer divide method. The display trace can be enabled by setting the `x->ena_dplytrc = 1;` the code above is modified to do this:

```
PPM *x = new PPM(12345);
PPM *x_copy = new PPM(x, -1);
PPM *y = new PPM(29);
PPM *rem = new PPM(0);
printf("the divide display trace dump:\n\n");
x->ena_dplytrc = 1;
x->DivStd(y, rem);
printf("\n");
cout << x_copy->Print(DEC) << " divided by " << y->Print(DEC) << " is ";
cout << x->Print(DEC) << " with remainder " << rem->Print(DEC) << endl;
```

```
    cout << "the native PPM result looks like: " << x->Prints(10) << endl;
    wait_key();
```

The output of the code above is:

```
    the divide display trace dump:

    +d -1n 2[]: -1n 3[]: -2n 5[]: <E:7> {3} <1> *(2)
    +d 2[]:3[]: -1n 5[]: <E:3> {3} <1> *(2) *(2)

    12345 divided by 29 is 425 with remainder 20
    the native PPM result looks like: 9 20 0 5 7 9 0 7
```

The display trace dump feature is explained as follows:

+d means the divisor was incremented by one unit;
-Xn means the numerator was decremented by X units;
M[]: means the divisor and numerator was divided by the value M;
<E:N> means a base extension was performed, and took N clocks;
{N} means N number of ModDiv cycles before Extend was performed;
<1> means the divisor has been reduced to one;
<0> means the dividend was reduced to zero;
*(N) means that a comparison having N clock cycles was performed;

If you look at the DivStd() integer method in code, you'll see the method simply calls another divide method. The reason is the PPM integer divide routine is still under development. The user is free to "comment in" another version of the divide routine for study. For example, the DivPM3 routine is a version which decrements the divisor instead of incrementing the divisor. This version requires a redundant digit to operate, so if you don't maintain a redundant digit, your divide may fail when dividing numbers near the top end of your RNS range. The DivPM4 version has debug print() methods inserted, so if you comment in this version, you can see that.

There are a lot of improvements that can be made to the divide routines. Many potential improvements are based on the desire to reduce base extends and comparisons. Much of the discussion regarding improving the integer divide methods are found in patent application US 2013/0311532 A1. In future revisions of RNS-APAL, we hope to find the "best of breed" integer divide routine, but until then, we've set the library to use the DivPM5 version, which we've found (so far) is stable and does not require a redundant digit.

PPM Value Testing and Comparison Methods

It's important to perform a check to determine if a value is zero. Most CPU's have a flag that indicates a zero is obtained as a result of an operation. While it's possible to use a compare method to check for zero, this approach is less efficient. Therefore, the RNS-APAL library includes several routines to perform a check on a PPM variable to determine if the result is zero, or if the result is one.

RNS-APAL provides a method to determine if a PPM value is zero called Zero(). It will pass back TRUE if the PPM value is zero, or FALSE if not. There is subtle differences between the Zero() check method in RNS-APAL, and an equivalent check for zero in a binary computer. The difference is the Zero() method ignores skipped digits. If not, the RNS-APAL library would require a value to be base extended before a check for zero, and this is very in-efficient.

RNS-APAL also includes a check for the value of one, called One(). This check is handy for division routines. Again, it could be replaced by a more complex Compare() method, but this is less efficient. Like Zero(), the One() check method ignores skipped digits. Both routines work for any RNS format, because a zero digit and a one digit is less than any valid RNS modulus.

Here is sample code that shows the use of the One() and Zero() methods; the sample is using an RNS with NUM_PPM_DIGS set to 8:

```
x->Assign(5);
x->Rn[3]->SkipDigit();      // skip a digit for demonstration
while(!x->Zero()) {
    if(x->One()) {
        cout << "x is one" << endl;
   }
   else {
        cout << "x = " << x->Prints(DEC) << endl;
   }
   x->Decrement();
}
```

The output of the code above is:

```
x = 5 5 5 * 5 5 5 5
x = 4 4 4 * 4 4 4 4
x = 3 3 3 * 3 3 3 3
x = 2 2 2 * 2 2 2 2
x is one
```

PPM Integer Compare Methods

In order that we develop programs that use the arithmetic methods of the PPM class, it's important that we compare values. RNS-APAL provides us two basic routines for comparison. The most basic routine is a routine to detect if two numbers are equal, called IsEqual(). This routine is important, since it comprises an identity check. Many algorithms may terminate on the fact the one value equals another. Therefore, there is no reason to perform a full blown comparison in these cases. The IsEqual() method is a true PAC method if implemented in hardware, and therefore, it is much more efficient to use IsEqual() than to use Compare.

The IsEqual() method supports skipped digits, but only if both arguments have skipped digits in the same digit positions. Otherwise, the IsEqual() method cannot guarantee an equality check. Future revisions of IsEqual() might flag the case when values having skipped digits are in different positions as an error case. (It does not at the time of this writing) The user should be careful when using this routine to compare values with skipped digits. If the PPM values being compared have no skipped digits, then the routine works as intended. Therefore, *it is advisable to use this routine on fully extended values only*. The following code shows the use of the IsEqual() method:

```
        x->Assign(5);
        y->Assign(1);
//      x->Rn[3]->SkipDigit(); // don't compare values with skipped digits !!
        while(!x->Zero()) {
           if(x->IsEqual(y)) {
                cout << "x is one" << endl;
           }
           else {
                cout << "x = " << x->Prints(DEC) << endl;
           }
           x->Decrement();
        }
```

The output of the code above is:

```
        x = 5 5 5 5 5 5 5 5
        x = 4 4 4 4 4 4 4 4
        x = 3 3 3 4 3 3 3 3
        x = 2 2 2 4 2 2 2 2
        x is one
```

RNS-APAL provides an arbitrary, unsigned integer compare method called Compare(). The Compare() method takes a PPM value argument, and compares it against the calling PPM value. If the calling PPM is greater, then the method returns TRUE, otherwise, it returns FALSE. The comparison method uses mixed radix conversion, converts both arguments simultaneously, and compares mixed radix digits least significant digit first.

The Compare() method should be used with fully extended, normal PPM values. This means that PPM values with partial power modulus should not be compared using Compare(). Furthermore, values should not contain skipped digits. The following code illustrates the comparison method:

```
        x->Assign(500);
        y->Assign(498);
        i = 5;
//      x->Rn[3]->SkipDigit();            // don't compare values with skipped digits !!
        while(i) {
           if(x->Compare(y)) {
                cout << "x is greater than y" << endl;
           }
           else {
                cout << "x not greater than y" << endl;
           }
           i--;
           x->Decrement();
        }
```

The output of the code above is:

```
        x is greater than y
        x is greater than y
        x is not greater than y
        x is not greater than y
        x is not greater than y
```

There are other variations of the Compare() method. One variation passes a clock variable by reference, so the user can determine the number of clocks required to perform the compare. This is useful for development and optimization of routines. Another comparison method, called ComparePart(), only compares the first N number of digits. This is useful for some routines, such as fractional multiplication.

In the future, more Compare() routines are expected, such as versions that handle skipped digits, and versions which handle partial power PPM variables. Until the need for those routines is fully established, the user should use the Compare() method. If the user needs to compare partial power PPM values, the PPM values should be normalized first using the Normalize() method. If the user needs to compare values with skipped digits, then the user should base extend the values using ExtendPart2Norm() before performing the comparison. The user should remember that all high level routines should return fully extended, normal PPM results. Therefore, in the context of application algorithms, or even advanced primitive routines, there is no over-riding need for advanced versions of Compare().

Normalizing PPM Values

In the preceding section, we introduced the need to "base extend" values with skipped digits, and to "normalize" values with partial power modulus. These routines are used when manipulating PPM variables in advanced routines, such as fractional division and integer division. In this section, we introduce these methods and show how to use them.

PPM Base Extension

The base extension process recovers skipped digits. The simple mathematics explanation is that invalid digits are recovered such that the original residue number (the value with skipped digits) remains the same value, albeit with redundant digits. So one way to describe base extension is to say that invalid digits are recovered but become redundant digits.

Base extension is one of the most important operations in residue arithmetic. In fixed radix arithmetic, base extension is similar to adding zeros to the left of the most significant digit. Obviously, we have no reason to do this, since we know these "zero" digits will not alter any arithmetic operation. A better analogy is the binary sign extension operation. In this case, if we cast a negative 16 bit value into a 32 bit register, we need to sign extend the result, since we know that the most significant digits should be 1; but in doing so, we realize that all the ones are redundant and will not affect the value of the original 16 bit negative number.

In the same way, the process of residue digit extension is seen to be "filling out" a residue register, such that the original value remains un-changed. But because residue numbers have no fixed digit significance, and because the "radix" (modulus) of each digit position differs, it's understood that the recovered digit values are un-predictable, i.e., they cannot be ascertained by inspection or simple static insertion. This is where the base extension procedure comes into play.

One of the new concepts of the RNS_APAL library, and indeed the new residue processors derived from these libraries, is the process of "simultaneous digit base extension". While mathematically not new, in practice, it is relatively novel. In fact, it is now realized that new efficiencies of residue arithmetic are predicated on the idea of "holding off" base extension, and then performing base extension on as many digits as possible at once. By doing this, we speed the base extend algorithm by reducing the number of times base extension is activated, and furthermore, when we finally perform base extension, our base extension is shorter because the number of valid digits is reduced. So this is a win-win strategy. Power based residue number systems help us achieve this goal. This is a primary strategy for speeding up the integer division method discussed earlier.

Base Extend Methods

RNS-APAL provides us several different base extend methods. One of the methods, ExtendNorm(), is an earlier version of the algorithm developed when the library did not support power based modulus. It's descendant is ExtendPart2Norm(). We may have just as well deprecated the ExtendNorm() method, and replaced it with a single "do all" method, called Extend(). (So, to this end, we have included an Extend() method, and it simply calls ExtendPart2Norm().) During early days of development, it was confusing to have several different types of Extend method, as they were called Extend(), Extend2(), and so on. So the names were changed to be more clear on what they are capable of, hence the name ExtendPart2Norm().

To be precise, the ExtendNorm() method only works with full power based (normal) PPM values. The ExtendPart2Norm() will work with these values as well, but also works with partial power PPM values, and has the effect of extending both skipped digits, and extending partial powers to full powers. This is an important function, since we want to recover the full modulus of the PPM in anticipation of saving base extend cycles as discussed above.

For this manual, let's use the Extend() method, but remember it really just calls the ExtendPart2Norm() method, which obviously stands for "extend partial to normal".

```
x->Assign(1234);
cout << "the original value is: " << x->Prints(HEX) << endl;
x->Rn[0]->SkipDigit();
cout << "skipping first digit : " << x->Prints(HEX) << endl;
x->Extend();
cout << "after base extension:  " << x->Prints(HEX) << endl;
wait_key();
```

The output of the code above is:

```
the original value is: 12 13 9 2 2 c a 12
skipping first digit : * 13 9 2 2 c a 12
after base extension:  12 13 9 2 2 c a 12
```

Note that in the code above, the original value is modified by invalidating the first digit. When we print this value, we see the asterisk in the first digit. After calling Extend(), the first digit is recovered, and the original digit value, 0x12, is printed.

```
x->Assign(1236);
cout << "the original value is: " << x->Prints(HEX) << " = " << x->Print(10) << endl;
x->ModDiv(4);
cout << "dividing by 4 is OK: " << x->Prints(HEX) << " = " << x->Print(10) << endl;
x->Extend();
cout << "after base extension:  " << x->Prints(HEX) << " = " << x->Print(10) << endl;
wait_key();
```

The output of the code above is:

```
the original value is: 14 15 b 4 4 1 c 1  = 1236
dividing by 4 is OK : X|5 c 9 1 1 a 3 5  = 309
after base extension:  15 c 9 1 1 a 3 5  = 309
```

Note that we start with a value that is evenly divisible by 4. After performing a ModDiv(4) operation on the number, we have divided by four, but we have also reduced the power of the base two modulus by two. When we use the Prints() method, we print a native RNS format, but the method also provides us with a useful feature, that is, the first digit is printed with a preceding "X|" string. This means that the digit has a reduced power modulus, i.e., the "X" is printed to indicate the upper powers are invalid.

We used the hexadecimal form of Prints(HEX) for a reason; it allows us to easily see the binary format of each digit. When the Extend() method is called, we see the upper powers of the base two modulus restored. Note however, that unlike the skipped digit example, the original value is not restored. The reason is that the original value has been divided by 4, and during the ModDiv operation, it lost its most significant two powers of the base two modulus. Calling Extend() restored the upper two powers of the base two modulus, and now we clearly see the upper digit of the base two modulus using the HEX digit print format. In this case, using Extend(), we have restored "redundant powers" of the base two modulus.

If we were to use the ExtendNorm() method above, we would get an error message since we are attempting to base extend a non-normal PPM value. Again, don't use the ExtendNorm() method, it will likely be deprecated in the future.

Normalize Extend Method

A more advanced form of digit extend is the Normalize() method. The normalize method base extends a PPM value, but returns the value back to the format of the base number system it was derived from. This brings up an advanced topic, that is, the subject of derived partial modulus RNS. In the prior section regarding advanced Assign methods, we introduced the AssignPM() method. This method will create a derived partial power number system, and we used it to demonstrate how to derive a partial power PPM value. In this example, we'll use another method, a special PPM constructor designed to derive new values that take on the properties of the PPM value passed to it.

First, we'll develop some code that demonstrates a derived partial power number type. We do this by reducing a full power based variable by the ModDiv method, then we instantiate another PPM variable by copying the parameters of the partial power number. By doing this, we have a derived partial power number system that will not revert back to a full power based number system when we invoke the Extend() method. Here is such code:

```
x->Assign(1236);
cout << "the original value is: " << x->Prints(HEX) << " = " << x->Print(10) << endl;
x->ModDiv(4);
cout << "dividing by 4 is OK : " << x->Prints(HEX) << " = " << x->Print(10) << endl;
PPM *derived = new PPM(x, -1);
cout << "printing derived : " << derived->Prints(HEX) << " = " << derived->Print(10) << endl;
derived->Extend();
cout << "after base extension: " << derived->Prints(HEX) << " = " << derived->Print(10) << endl;
derived->Normalize();
cout << "after normalize:  " << derived->Prints(HEX) << " = " << derived->Print(10) << endl;
wait_key();
```

The output of the code above is:

```
the original value is: 14 15 b 4 4 1 c 1  = 1236
dividing by 4 is OK : X|5 c 9 1 1 a 3 5  = 309
printing derived : 5 c 9 1 1 a 3 5  = 309
after base extension: 5 c 9 1 1 a 3 5 = 309
after normalize:  15 c 9 1 1 a 3 5  = 309
```

In the code above, we process the x PPM value like before. After dividing the value by 4 using the ModDiv(4) method, the PPM value has a reduced base two modulus, and shown in the second line of the printout, same as the previous example. Next, we derive a PPM value from the partial power PPM x value using a special PPM constructor which takes the PPM x value as an argument: PPM(x, -1); we called the new PPM value "derived". Note that we pass a -1 as the second argument to the PPM constructor. This tells the constructor to use the value of the first argument. Otherwise, if the argument is non-negative, the value of x is assigned to the new derived PPM value. In both cases, the PowerValid modulus format of the first argument is assumed as the new "normal" format of the derived PPM value.

When we print the PPM derived value, we do not see the X| string preceding the digit 5 of the two's power modulus. Even after calling the Extend() method, the PPM derived value remains the same, it does not revert back to the original format of x. This is what we refer to as a derived partial power PPM value. Once the PPM value is derived, it is no longer considered to be a partial power PPM value, it is considered to be a full power derived modulus format.

If we desire to return the derived PPM value back to its base modulus format, we call the Normalize() method. This is shown in the last line of the printout, and as seen, the value is returned back to the original modulus format of the x PPM value, from which it was derived.

The main reason for supporting derived types is to support advanced algorithms, such as the fractional scaling method to be introduced later. The fraction al scaling algorithm introduces the notion of sliding point operations in RNS, and also introduces a dynamic fractional range capability. The derived partial power types aids us by allowing us to derive a new RNS modulus type, and then perform basic PPM operations of the derived type without altering its new "normal" format.

Signed Integer Residue Representation and Arithmetic

Signed quantities are important to support if one requires general purpose arithmetic. In the case of residue numbers, the process of performing signed arithmetic has been unclear and seemingly impractical. However, to be fair, many good techniques have been proposed that work in a small scope of examples. In Szabos and Tanaka's pivotal book, reference is made to using the method of complements for supporting signed residue quantities. Additionally, in their book, reference is made to using a sign magnitude approach, that is, using sign flags to indicate the sign of a residue number. In my approach, I combine the two techniques, but in the final analysis, the method of complements is the more important, indispensible technique.

Because residue numbers don't support carry, it is difficult to ascertain the sign of a residue value using *inspection alone*. In the case of binary, when using two's complement notation, we essentially "park" a sign bit in the most significant bit position, where this bit position coincides perfectly with the most significant bit of the upper range of the numbers representing negative numbers. This convenience makes it possible to inspect a two's complement number and readily ascertain, by inspection, if the number is negative. This technique works for all basic arithmetic operations (addition, subtraction, and multiplication) unless the range of the number system has been exceeded.

To be more clear, the method of complements allows us to process signed *residue* values using standard arithmetic processes, and we can expect the resulting value be correctly "signed" as well. (By signed, we mean the result is correct in terms of the method of complements representation.) Again, this is true as long as we do not overflow our number system range. As mentioned above, the two's complement system of binary has the property that once the two's complement arithmetic is complete, we may simply inspect the most significant bit to determine the sign of the result. This is not true with residues; while we get a correct result in terms of method of complements, we cannot easily ascertain the sign of the residue result by simple inspection of the residue value.

If we also support a sign flag for our residue number, and we manage the value of that sign flag throughout various operations, it may be possible to ascertain the sign of a residue value by inspection alone. However, a problem arises in certain cases when the sign of the result cannot be managed without further calculation. For example, if we add a negative residue number to a positive residue number, we cannot know whether the result is negative or positive without further calculation. In this case, the sign flags of the arguments cannot give us the value of the resulting sign. It is with these cases we have a problem with the sign flag, since we do not know whether to set or clear it. In these cases, it has been suggested to use another bit, called a "sign valid" bit, which indicates whether the sign flag is to be trusted, that is, if set it indicates the sign flag is valid. In this case, since we cannot know the value of the sign flag, we clear the sign valid bit to indicate the sign bit is "not valid". In other cases, if we know the value of the resulting sign flag, we set the sign valid bit.

I have implemented this approach in the library, and also in the Rez-9 processor. As time has progressed, it's become more clear as to the benefit of this approach. To make a long story short, there are some operations where we *must* know the sign of an operand before the operation takes place.

One example operation is the square root. We cannot take the square root of a negative number using ordinary real numbers. A more common case is integer division. While we can *mathematically* divide negative numbers, the algorithm of division only works on the "magnitude" of two numbers, since division is the process of sub-dividing a dividend magnitude into equal portions depending on the magnitude of a divisor. If we try to divide a negative number directly, we get the *magnitude of the complement* divided and this is incorrect.

So to make a long story shorter, we can save time if we know beforehand a value is negative before being divided, since it takes much less time to complement a negative number than to determine the operands sign using a "sign extend()" function. (A sign extend function is essentially a compare operation, and this is a time consuming process in residues.) However, an important discovery made in the development of residue arithmetic is that certain key operations will "recover" the sign flag. So, in summary, even if the sign flags of the arguments are known, we have 1) operations (and cases) that invalidate the sign flag, 2) operations where we can manage, maintain or derive the sign flag, and 3) operations that recover, or generate, sign flags. Operations which generate sign flags is a boon for the sign magnitude approach, and this feature can be used to save clock cycles in a residue processor. More about these operations when we cover fractional multiplication later.

So to approach a fast conclusion to a seemingly simple, but actually quite complex reality, we find that it is not necessary to know the sign of a value by inspection, but it is advantageous if we *do know*. This makes some processes faster. I hope you've appreciated my short dissertation on signed residue values!

Now let's get to the library methods for signed residue integer arithmetic.

SPPM Signed Integer Class

To represent signed integer residue numbers, a class object called SPPM was developed. This stands for "signed partial power modulus". As with the PPM class, we require a method to create a signed integer residue number type. Because we need an unsigned integer type to serve as an underlying primitive for our signed integer type, it makes sense that the SPPM type be a derived class of the PPM type. This allows us to use PPM class methods when operating on the *magnitude* of the signed integer type.

To declare a signed integer residue type, we can pass a signed __int64 type to the SPPM constructor:

```
SPPM *si = new SPPM(-100);
```

In the code above, the variable "si" is created and is initialized with the value -100. To print the signed integer residue, we can use the Print(int radix) method, similar in name to that found in the PPM class:

```
cout << "the signed si: " << si->Print(DEC) << endl;
```

The output of the code above is:

```
the signed si: (V)-100
```

Note that the print output has the string "(V)" preceding the signed decimal integer. This indicates that the si variable has a valid sign flag, i.e., the sign valid bit is set TRUE. When using the SPPM constructor, all values are created with a sign valid bit set TRUE. Furthermore, the SPPM Print() method always prints the sign of the value, whether the sign is '-' or '+'. This allows the user to unambiguously determine that the value printed is a signed integer, and of course, allows the user to positively determine the sign of the value. However, an important point is that the sign symbol is NOT determined by the inspecting and printing the sign bit value.

The reason is the print() method determines and prints the sign of the SPPM value based on its magnitude, NOT based on the sign flag. So in summary, it is the method of complements that controls the sign symbol of the print() method; *this guarantees the true signed value is always printed.* Moreover, if the sign valid bit is FALSE, the sign flag is invalid, and the Print() method displays the "(I)" string preceding the true value. The following code demonstrates this feature:

```
SPPM *si = new SPPM(-100);
si->SignValid = FALSE;
cout << "si = " << si->Print(DEC) << endl;
```

The output of the code above is:

```
si = (I)-100
```

In this output, the string "(I)" precedes the actual signed value which indicates the sign flag is invalid. For this test, we "forced" the sign valid flag to FALSE in the second line of the code above. (The constructor always returns SignValid = TRUE). Note that the flag variables should be private class members, but I kept them public for testing purposes. Future releases will make the SignFlag and SignValid SPPM class variables private. To read the SignFlag and SignValid variables, use the GetSignFlag() and GetSignValid() methods respectively.

So what happens if we force the sign flag to be incorrect, yet keep the sign valid bit TRUE? Let's try this by creating more test code:

```
SPPM *si = new SPPM(-100);
si->SignFlag = POSITIVE;
cout << "si = " << si->Print(DEC) << endl;
```

The output of the code above is:

```
si = (V)(E)-100
```

In this output, the string "(V)(E)" precedes the number. The (V) part indicates the SignValid flag is TRUE, which should indicate the sign is valid, while the (E) part indicates the sign flag is actually in error. That is, the range of the magnitude of the value, which is *measured* by the Print() method, does not agree with the state of the sign flag. Thus, the Print() method detects if the sign flag is set properly. If not, the (E) symbol will be printed. As it turns out, the use of sign magnitude notation, with the addition of a sign valid flag, has unexpected benefits, and that is to detect discrepancies in the processing of numbers. While this method may not detect all problems, it has proved very useful in the many types of iterative programs we've run; the occurrence of an error code is a tell-tell sign that an overflow has occurred, or a bug in our algorithm has occurred. For this reason, the sign magnitude approach is useful for residue processors, and also for the RNS-APAL library.

If we want to print the native format of the SPPM variable, we can use the Prints() method. Here is the example used above and adding the Prints() method and using a simple 8 digit RNS system:

```
SPPM *si = new SPPM(-100);
si->SignFlag = POSITIVE;
cout << "native si: " << si->Prints(DEC) << endl;
cout << "si = " << si->Print(DEC) << endl;
```

The output of the code above is:

```
native si: (V)(E)-28 8 0 5 10 4 2 14
si = (V)(E)-100
```

Note the output of the Prints(DEC) method also shows an (E) error code. This is because the native print method measures the magnitude of the value and checks the magnitude against the state of the sign flag. The Prints() method also explicitly prints the Sign Valid state. If the sign flag matches the measured sign, the sign printed by Prints() reflects the sign flag state. If an (E) code is printed, the sign flag is set opposite to the sign printed.

SPPM Assign Methods

Of course, like the PPM unsigned integer type, there is a need to assign SPPM types from one variable to another, or simply assign an initializing value to an SPPM type. However, unlike the PPM types, there is no concern for partial power modulus of an SPPM type. The reason is simple: negative residue numbers are always expected to be fully extended. Remember, it is not legal to *directly* divide negative residue numbers, so not even the ModDiv() method is defined for signed quantities. Moreover, we don't expect to have "skipped digits" in an SPPM type, since this indicates that a modulus has been divided out. If one is to manipulate the modulus powers of a number, or divide out a modulus resulting in a skipped digit, that must be done at the PPM class level. Note that the SPPM class doesn't stop you from doing this, but direct division of negative numbers will result in errors in your code!

So thankfully, the discussion of Assign() methods for SPPM types is rather straight forward. The SPPM Assign() method has three overloads, one that takes a long long integer, one that takes another SPPM type, and one that takes a signed decimal or hexadecimal string type. Here is some examples of the SPPM::Assign() methods:

```
SPPM *si = new SPPM(0);
SPPM *sj = new SPPM(-1);
si->Assign("-12345");
cout << "si = " << si->Print(DEC) << endl;
si->Assign(sj);
cout << "now si = " << si->Print(DEC) << endl;
sj->Assign(100);
si->Assign("0x1234");
cout << "sj = " << sj->Print(DEC) << endl;
cout << "si base 16 is now: " << si->Print(HEX) << endl;
```

The output of the code is:

```
si = (V)-12345
now si = (V)-1
sj = (V)+100
si base 16 is now: (V)+0x1234
```

SPPM Arithmetic Methods

SPPM arithmetic methods are similar to PPM arithmetic methods because they are designed to emulate PAC, or parallel array computation. The basic PAC arithmetic operations are Add(), Sub(), and Mult(). In addition, the SPPM arithmetic methods are designed to take best use of the sign flags of the arguments. The result is that basic arithmetic PAC operations on signed SPPM values will produce a valid *magnitude* result (provided there is no overflow), and they will do the best they can with returning the correct sign of the result given the information they have.

In the best of cases, the SPPM PAC operations can *maintain* the validity of the sign flag of a result provided both arguments themselves have valid sign flags. In the worst of cases, the SPPM PAC operations will invalidate the sign flag of the result, even in cases when the sign flag of both arguments is valid. In general, SPPM operations cannot *generate* a valid sign flag, that is, they cannot return a valid sign flag when one or both arguments have an invalid sign flag. The only exception to this rule is when the result is zero, in which case the sign flag is always known to be positive.

Let's begin with a few code examples to illustrate the use of SPPM PAC operations:

```
SPPM *si = new SPPM(10);
SPPM *k = new SPPM(5);
si->Add(k);
cout << "si = " << si->Print(DEC) << endl;
```

The output of the code above is:

```
si = (V)+15
```

Keep in mind that when an SPPM is instantiated, it's sign flag is valid. If we add two positive numbers together, our result is positive, and the sign flag is valid. Now let's perform a subtraction:

```
SPPM *si = new SPPM(10);
SPPM *k = new SPPM(5);
si->Sub(k);
cout << "si = " << si->Print(DEC) << endl;
```

The output of the code above is:

```
si = (I)+5
```

Note this time that the subtraction of two positive signed quantities produces a correct result, but this time the sign flag is invalid. *The subtraction of two positive quantities always invalidates the sign flag unless the result is zero*. The reason is that PAC residue operations do not perform carry from digit to digit, so there is no way we can know the sign of the result without further processing. Let's try adding negative numbers:

```
        SPPM *si = new SPPM(-10);
        SPPM *k = new SPPM(-5);
        si->Add(k);
        cout << "si = " << si->Print(DEC) << endl;
```

The result of the code above is:

```
        si = (V)-15
```

In this case, the addition of two negative numbers produces a correct, and also a valid flag. The addition of two negative values will always produce a valid sign flag provided the arguments have valid sign flags and an overflow of the number range does not occur. But as you might guess, the subtraction of these same two negative quantities invalidates the sign flag:

```
        SPPM *si = new SPPM(-10);
        SPPM *k = new SPPM(-5);
        si->Sub(k);
        cout << "si = " << si->Print(DEC) << endl;
```

The result of the code above is:

```
        si = (I)-5
```

Interestingly, there are more cases. For example, consider the case when we add a negative number with a positive number:

```
        SPPM *si = new SPPM(10);
        SPPM *k = new SPPM(-5);
        si->Add(k);
        cout << "si = " << si->Print(DEC) << endl;
```

The result of the code above is:

```
        si = (I)+5
```

If we add a positive number to a negative number, the sign will be invalidated. However, if we subtract a positive number from a negative number:

```
        SPPM *si = new SPPM(-10);
        SPPM *k = new SPPM(5);
        si->Sub(k);
        cout << "si = " << si->Print(DEC) << endl;
```

The result of the code above is:

```
        si = (V)-15
```

In this case, if we subtract two values, each with a different sign, then the resulting sign will be valid.

For multiplication, the rules are a little different. Provided that both arguments' signs are valid, and no overflow occurs, the Mult() method will always return the correct sign of the result. Here is a simple example:

```
SPPM *si = new SPPM(-10);
SPPM *k = new SPPM(5);
si->Mult(k);
cout << "si = " << si->Print(DEC) << endl;
si->Mult(si);
cout << "si = " << si->Print(DEC) << endl;
```

The result of the code above is:

```
si = (V)-50
si = (V)+2500
```

If the sign of one or both arguments is invalid, the SPPM PAC operations will always return an invalid sign flag. Here is an example to illustrate this:

```
SPPM *si = new SPPM(10);
SPPM *k = new SPPM(5);
k->SignValid = FALSE;
si->Mult(k);
cout << "si = " << si->Print(DEC) << endl;
```

The output of the code above is:

```
si = (I)+50
```

The last example of this section illustrates that if the result is zero, the sign should be set valid regardless:

```
SPPM *si = new SPPM(0);
SPPM *k = new SPPM(-5);
k->SignValid = FALSE;
si->Mult(k);
cout << "si = " << si->Print(DEC) << endl;
```

The output of the code above is:

```
si = (V)+0
```

As seen above, by definition, the sign of zero is positive.

Division of SPPM types

We mentioned earlier that *direct* division of SPPM types is not valid. However, mathematically, the division of signed integers *is* valid. Therefore, as in the case of PPM types, RNS-APAL provides a division method called Div() to handle the division of signed SPPM values. Note this operation is not a PAC type. SPPM division is considered a slow, multi-cycle operation. But hey, when you need to divide, you need to divide!

A couple of notes regarding the SPPM Div() procedure. We are using sign magnitude notation (sign flags) because in many cases we either know the sign of a value, or we can manage the sign of a value, or we can generate the sign as a result of a prior calculation or assignment. Therefore, since the division algorithm only divides the magnitude of a value, it helps if we *know* the sign of the arguments beforehand. If we know the sign of the arguments, and if any of the arguments are negative quantities, we only need to use a Complement() operation before we divide. Otherwise, if we do not know the sign of either argument, it is necessary to use a "sign extend" method. Note that the Complement() method is a PAC operation, so it is fast, while the sign extend operation is a slow, multi-cycle operation.

Obviously, there is a benefit if we know the sign of the arguments when we divide. The SPPM method works in this way. If an argument has a valid sign flag, and the sign flag of the argument is negative, it will be complemented before division. Otherwise, if the sign is not known, the argument must be sign extended to determine its sign. Again, if the argument is determined to be negative, the value will be complemented. If for some reason the user attempts to divide a number that has a valid sign flag, but the wrong sign polarity, i.e., it prints as an (E) error, the division will yield an incorrect result. In some designs, the divide operation might always sign extend results, but this would reduce the need for maintaining and supporting sign magnitude notation.

Bottom line for RNS-APAL: since we must trust sign flags, we must manage sign flags with care. *It is the intent of the RNS-APAL to manage sign flags for you automatically*, so you're good as long as *you're* not directly manipulating the sign flags.

Signed Div() Mathematics:

The SPPM Div() method accepts a signed divisor and returns a signed quotient, and also returns a signed remainder (by reference). The following definition provides the basis for the sign of the quotient and remainder:

(Quotient * Divisor) + Remainder = Dividend (Eq. 1)

This simple relationship means the sign of the remainder will assume the sign of the dividend. The sign of the quotient follows well known math rules, and so depends on the sign of the divisor and dividend.

Let's try a few examples to illustrate the use of the SPPM Div() method:

```
SPPM *dividend = new SPPM(-101);
SPPM *divisor = new SPPM(5);
SPPM *r = new SPPM(0);
SPPM *q = new SPPM(0);
dividend->Div(divisor, r);
q->Assign(dividend);
cout << "q = " << q->Print(DEC) << ", remainder = " << r->Print(DEC) << endl;
```

The output of the code above is:

```
q = (V)-20, remainder = (V)-1
```

If we switch signs of the divisor and dividend, we get the following:

```
SPPM *dividend = new SPPM(101);
SPPM *divisor = new SPPM(-5);
SPPM *r = new SPPM(0);
SPPM *q = new SPPM(0);
dividend->Div(divisor, r);
q->Assign(dividend);
cout << "q = " << q->Print(DEC) << ", remainder = " << r->Print(DEC) << endl;
```

The output is now:

```
q = (V)-20, remainder = (V)+1
```

Note the quotient remains the same, but the sign of the remainder has changed as a result of ensuring the basic mathematics of signed division holds according to equation X. Division is also prone to application errors, most notably, the case of division by zero. Here is an example for handling this case:

```
SPPM *dividend = new SPPM(-101);
SPPM *divisor = new SPPM(0);
SPPM *r = new SPPM(0);
SPPM *q = new SPPM(0);
int error = dividend->Div(divisor, r);
if(error == DIVIDE_BY_ZERO) {
   cout << "Error: divide by zero!" << endl;
   }
else {
   q->Assign(dividend);
   cout << "q = " << q->Print(DEC) << ", remainder = " << r->Print(DEC) << endl;
}
```

The output of the code above is:

```
Error: divide by zero!
```

SPPM Sign Control and Testing

As you might expect, the SPPM class supports several sign test and sign manipulation methods. One of the most basic sign related operations is the Negate() method. The SPPM Negate() method changes the sign of the value. In particular, the magnitude of the value is complemented, and if the sign valid flag is TRUE, the sign flag is inverted. The Negate() method also checks for a zero value, and if the value is zero, it will set the sign flags appropriately in this case.

```
SPPM *a = new SPPM(-100);
cout << "a = " << a->Print(DEC) << endl;
a->Negate();
cout << "a = " << a->Print(DEC) << endl;
a->SignValid = FALSE;
cout << "a = " << a->Print(DEC) << endl;
a->Negate();
cout << "a = " << a->Print(DEC) << endl;
```

The output of the code above is:

```
a = (V)-100
a = (V)+100
a = (I)+100
```

a = (I)-100

Note that when the sign flag is valid, the Negate() function inverts the sign flag as well as complements the value. When the sign flag is invalid, the value is only complemented. In no case is the sign flag checked for validity. For this reason, the Negate() method is a PAC method, and is considered a fast hardware method.

```
SPPM *a = new SPPM(-100);
cout << "a = " << a->Print(DEC) << endl;
a->Abs();
cout << "a = " << a->Print(DEC) << endl;
```

The output of the code above is:

```
a = (V)-100
a = (V)+100
```

If we invalidate the sign flag of the SPPM variable, then the Abs() method will sign extend the value to determine if the value needs to be complemented. Therefore, the Abs() method always returns a value with a valid positive sign flag. Here is the test code to demonstrate this:

```
SPPM *a = new SPPM(-100);
a->SignValid = FALSE;
cout << "a = " << a->Print(DEC) << endl;
a->Abs();
cout << "a = " << a->Print(DEC) << endl;
```

The output of the test code is:

```
a = (I)-100
a = (V)+100
```

There are many cases where we need to determine the sign of a value because the sign flag is not valid. SPPM supports two methods: CalcSign() and CalcSetSign(). The CalcSign() method measures the magnitude of the value, and determines the sign based upon this comparison. It does not *set* the value's sign flags as a result. This allows the variable sign to be known, and allows us to set the state of the sign in another way. The CalcSetSign() is essentially a "SignExtend()" operation, which means the sign of the value's magnitude is determined, the value's sign is set appropriately as a result of this determination, and the sign valid flag is set TRUE.

Let's show an example that illustrates the use of these two methods:

```
SPPM *a = new SPPM(-100);
a->SignValid = FALSE;
int sign = a->CalcSign();
cout << "a = " << a->Print(DEC) << endl;
if(sign == NEGATIVE) {
   cout << "value is negative" << endl;
}
else {
   cout << "value is positive" << endl;
}
a->CalcSetSign();
cout << "a = " << a->Print(DEC) << endl;
```

The output of the code above is:

```
    a = (I)-100
    value is negative
    a = (V)-100
```

The sign of the SPPM variable 'a' is invalidated for the test. If we then test the sign by calling the CalcSign() method, we find the sign of the value is negative. When the SPPM value is printed after the call to CalcSign(), the sign flag is still invalid as shown by the (I) symbol. Thus, the CalcSign() method did not change the variable's sign valid flag. However, after calling the CalcSetSign() method, we see that the value now has a valid sign flag indicated by the (V) symbol.

In future versions of RNS-APAL, the SignValid and SignFlag members will be private. If the user needs to query these members, then the following two methods are provided: GetSignFlag() and GetSignValid(). The user may use the #defines in SPPM.h to test for valid values of these flags, they are `NEGATIVE`, `POSITIVE`, `SIGN_VALID`, `SIGN_INVALID`.

Another important method is GetAbsRange() which returns the absolute range of the positive numbers of the SPPM type. If you want to compare against this range, say to determine if a magnitude is mapped to the positive range or the negative range, you may want to decrement this value by one. Doing so provides the last positive number. The following code prints the positive range for a simple 8 digit RNS with no redundant digits:

```
    SPPM *a = new SPPM(0);
    PPM *range = new PPM(0);
    a->GetAbsRange(range);
    range->PrintDemo();
    printf("\n");
    cout << "the range of a = " << range->Print(DEC) << endl;
    a->PPM::Assign(range);
    cout << "a = " << a->Print(DEC) << endl;
    a->PPM::Increment();
    cout << "a = " << a->Print(DEC) << endl;
    a->Complement();
    cout << "a = " << a->Print(DEC) << endl;
```

The output of the code above is:

```
    32 27 25 7 11 13 17 19
    -- -- -- - -- -- -- --
    15 26 24 6 10 12 16 18

    the range of a = 3491888399
    a = (V)+3491888399
    a = (V)(E)-3491888400
    a = (V)(E)-3491888400
```

There are a few things about the output worth noting. For one, the GetAbsRange() method passes back the last positive value via the PPM argument. This value is printed in RNS native format using the PrintDemo() method first. The observant reader will see a pattern of the residue digits versus their modulus values. The PPM value is then printed in decimal format. If we cast this value back into the SPPM 'a' variable, and we print the signed value, we see it prints as a positive value. However, if we increment the native PPM value by one (using PPM::Increment()), and we print the SPPM value, we see that it is printed as a negative value, but with an error flag, which indicates the sign flag is wrong, since we didn't adjust for this. This is what we expect. One value (in magnitude) past the last positive value should land us in the negative range, and it does.

Furthermore, we see that there is one more negative number in the negative range than there is in the positive range. This discrepancy is not new, as this situation exists in a standard "even" number system because zero is mapped to the positive range, and this leaves one more value for the negative range than the positive range. This inequity is important in that it allows the method of complements to work. However, this "last" negative number is not a "good" negative number, it's a bit of a heretic, since it's the only number that if we complement will not return its positive value. It's the phantom number in our system; complementing this number gives us the same number as shown in our test code.

Our library doesn't currently detect this bad negative number, but I may add a check for this in the future. Basically, this last negative number is actually an overflow value, but we can't expect to trap overflows using this value, because it is unlikely our result will land at exactly this number. So it's not worth trapping for the vast majority of calculations.

Comparison of SPPM types

Obviously, there is a need to compare signed integer quantities. The SPPM class provides us a signed comparison method called Compare(). The SPPM::Compare() method provides us the possibility of performing a fast PAC compare if both signs flags are valid, and if they differ. Therefore, the Compare() method uses sign flags, and therefore sign flags must not be in an error state for Compare to work properly. The compare method will sign extend any argument with an invalid sign, and then perform the compare. Therefore, as a side effect, the Compare() method may sign extend the calling variable as well as the argument variable. Therefore, the Compare() method is an operation which *generates* a sign flag.

The Compare() method compares the calling SPPM variable with a SPPM argument. If the calling SPPM variable is greater than the SPPM argument, the method passes back a TRUE condition. Otherwise, the Compare() method passes back false. The Compare method does not handle a check for equality. If the user requires an equality check, use the PPM::IsEqual() method. Here is a code example showing the use of the Compare() method:

```
SPPM *a = new SPPM(-100);
SPPM *b = new SPPM(100);
int greater_than = a->Compare(b);
if(greater_than) cout << "a is greater than b" << endl;
if(!greater_than) cout << "a is not greater than b" << endl;
a->Mult(a);
```

```
greater_than = a->Compare(b);
if(greater_than) cout << "a is greater than b" << endl;
if(!greater_than) cout << "a is not greater than b" << endl;
```

The output of the code is above:

```
a is not greater than b
a is greater than b
```

Fractional Representation in Residues and the SPMF Class

It was previously thought that the residue number system is an "integer only" number system. This view is a modern view, and has been propagated in many research papers and texts, including recent publications. However, we now know this is not correct. The residue number system can provide an extremely rich fractional number system which includes a dynamic set of "radices", or modulus. In fact, residue fractions can support perfect ratios of their fractional modulus, including multiplicative combinations of them. For example, our basic power based systems support perfect ratios such as 1/2, 1/3, 1/5, 1/7 and 1/11 exactly, while they also support perfect ratios of products of their fractional modulus, such as 1/(2*3), 1/(3*5), 1/(2*7), 1/(7*11), etc.

Residue fractions are much like fixed point binary or decimal fractions, in that both a whole number and fractional portion (less than one) is supported. The RNS-APAL library supports fractional residue types using the SPMF class, which stands for Signed Partial Modulus Fraction. Like their decimal counterparts, residue fractional numbers support a "fraction point", and they are always signed. The SPMF class is derived from the SPPM signed integer class. While it is not the objective of this manual to explain the mathematics of residue fractions, we will provide a few more paragraphs explaining them. Additional in-depth descriptions of residue fractions can be found in the authors original patent application US 2013/0311532 A1.

The existence of residue fractions calls into question the basic operations that can be applied to those fractions. To make a long story short, we enjoy several PAC (parallel array computation) operations on residue fractions. These PAC operations include addition of residue fractions, subtraction of residue fractions, and multiplication of a residue fraction by an integer (integer scaling). But we also encounter some new "slow methods", which include multiplication of a residue fraction by another residue fraction, and of course, the division of fractional residues.

One very interesting discovery of residue fractions is in the processing of product summations. Because of the property of residue numbers, we may use PAC operations to process all products and all summations, and we only need to normalize the final result to obtain a final result. So the story gets interesting, since operations can be combinations of PAC and slow methods. In the case of product summation, the residue method reduces the computational order of matrix multiplication with respect to fractional precision from $O(n^3)$ to $O(n^2)$. This fact is one of the newly discovered strengths of the residue number system, and provides the *new* motivation to pursue fractional residue systems.

The trick to supporting residue fractions is to define a set of modulus that will serve as our fractional range. In the RNS-APAL library, we choose to define the first 'F' number of digit modulus to define our fractional range. The number F is determined in RNS-APAL by the #define `SPMF_FRACTION_DIGS`. This define is found in *config.h* file. By setting this define to the desired number of fractional digits, the fraction point is effectively changed. It is important to note that the difference between SPMF_FRACTION_DIGS and NUM_PPM_DIGS forms the remaining "whole" number part of the residue number, and also any required *redundant digits*. For fractional processing, we have a simple rule, and that is, there should be a "range squared" number of digits in the system. This means that given the total range of our residue fraction, which consists of both the fractional range, and the whole number range of our representation, the overall RNS number system used for processing must be at least squared this value.

This range squared requirement is not new. When we multiply "32 bit "single word" binary values, we can expect to store the result in a 64 bit "double word" register. RNS-APAL must adhere to the same concept. The one main difference is that with residues, we choose to represent our "single width" values using a "double width" representation during processing cycles. In other words, we carry double width representation around at all times. However, this is only a design choice, since we may choose to carry a single width representation, but expect to base extend the singe range representation to a double width representation before we multiply. Base extension takes extra time, and so we are trading off the requirement of extra digit hardware for operation in less time; this tradeoff is common with residue processing.

Assigning and Printing SPMF Fractional Residue Variables

To use the SPMF fractional residue class, we need to instantiate a variable, assign a fractional value, and then be able to print the value. In order to duplicate these results, the user must compile the library with the same parameters. I have set the `#define NUM_PPM_DIGS 16` and `SPMF_FRACTION_DIGS 5.` I have also set `#define ABS_SIGNED_RANGE NUM_PPM_DIGS` which means there are no redundant digits in our system. Finally, I have commented out `//#define PRINT_REMAINDER.` The code below simply creates an SPMF variable, assigns a fractional value, and then prints that value:

```
        SPMF *fr = new SPMF(0);
        fr->AssignFP("100.125");
        cout << "the value of fr: " << fr->Print(DEC) << endl;
        cout << "the raw rns value of fr: " << fr->PPM::Prints(DEC) << endl;
        wait_key();
```

The output of the code above is:

```
        the value of fr: (V)+100.125000+
        the raw rns value of fr: 8 0 0 0 0 6 9 8 16 9 12 28 15 2 15
```

The output shows that a variable called "fr" is instantiated with an argument of zero. This is a bit misleading. The argument of the SPMF constructor simply passes this argument to the SPPM base class, and therefore, the argument is a raw SPPM integer argument, not a fixed point argument! Therefore, for all practical purposes, the SPMF constructor should simply be called with a "0" as its argument unless you know how to scale integer values into fixed point fractional values. Future versions of RNS-APAL may overload the SPMF constructor, and allow the user to pass float types, or other fractional types. Looking forward, RNS-APAL may include constructor overloads that allow the fraction point position to be dynamically altered. But the user has enough to digest at the moment, so we'll leave those ambitions for another day.

The easiest way to get an RNS fractional number initialized is to use the `void AssignFP(string fval)` method which stands for "assign fixed point". This method accepts a string representing a decimal fractional value. This value may include a preceding + or - sign, a whole integer part, a decimal point, and a trailing fractional part. At this time, only decimal numbers are supported. If the user inputs an invalid string, the method prints an error message and does nothing. It should be noted that the AssignFP method is really a *conversion* method. Unlike the integer class Assign methods discussed previously, which assign (and convert) an exact decimal or binary integer equivalent to a residue type, the AssignFP method may assign a value which is only approximate. Fractional residue numbers have a unique number space, and this space is not the same as a binary fraction space, or a decimal fraction space. Therefore, the AssignFP converts a number which is represented *exactly* in a decimal fraction space to your chosen residue fraction space, and depending on the number, this may involve an approximate conversion, not an exact conversion. This may take some time to get used to.

So the first question might be "what good is a residue fraction if it cannot represent decimal fractions exactly"? You may already know the answer. This story has two sides; that is, there are plenty of fractions that residue numbers can represent exactly that the decimal number system may only approximate. A simple example is the value 1/3. In decimal notation, this value repeats indefinitely. In residue format, as long as we have a single power of modulus 3 in our fractional range, the residue fraction of 1/3 will be exact. To make a long story short, this implies that using decimal notation to enter residue fractional values is actually *a severe limitation*.

In fact, for the same approximate (fractional) range, residue fractions support many more exact ratios having a different denominator than decimal or binary. For binary fractions, only ratios with a power of two in the denominator can be *exactly* represented. For decimal, only ratios with a power of 2, 5, or 10, can be exactly represented. For residue, as long as we have a *power* of a modulus mapped to our fractional range, we can support ratios with any *combination* of those modulus powers. That's a lot of exact representations, exponentially more. The reason is that for fixed radix systems, *combinations* of the *same radix* are reduced to the number of powers only. For residues, the modulus are different, and so (product) combinations of modulus exist and grow exponentially.

Printing Residue Fractions

As discussed above, the AssignFP method has interesting subtleties, the main reason being it's really a conversion method. The same is true of the `string SPMF::Print(int radix)` method; it is a conversion method which converts a residue fraction to a fixed radix fraction. Therefore, in general, the Print() method can only convert a residue fraction value as an approximate value in binary, decimal, or hexadecimal fractional space. To make things a bit worse, the print method has certain limitations, and may introduce minute errors of its own.

Because I wanted to present a "purest" view of residue arithmetic, I chose to develop algorithms that process entirely in residue format, without carry, even in the case of reverse conversion, and so the Print() method reflects this desire. In other words, converting a residue fraction to a fixed radix fraction could be accomplished with many other algorithms, many of them using carry, since after all, the fixed radix number system being converted *to* relies on carry. The in-accuracies of the present algorithm occur in the very last significant places, and so this is not a big deal. Recent enhancements to the Print() algorithm has improved this situation. In addition, other conversion routines, perhaps ones using a traditional arbitrary precision decimal or binary library, can be developed and included in the future.

There is a lot of room for all kinds of conversion routines for research and study, including support for rational types.

But as long as we're using the existing Print() method, we should be aware of these in-accuracies, as well as understand the other features it supports. If we look back at our previous example, we see the decimal output of our residue fraction looks good, but it has some trailing zeros, and a strange + sign at the end. The trailing + sign indicates to the user there is a remainder in the conversion process. Therefore, the Print() conversion method is producing a value that is actually in excess of the original decimal value string used in the AssignPM method.

Is it an in-accuracy of the input conversion, or is it an in-accuracy of the output? It turns out the in-accuracy is in the output conversion. We know this (at least I do) by viewing the raw RNS value. In the raw output, we see a lot of zeros in the first (fractional) digits. This is a sure sign that the 1/8 fractional portion is perfectly represented, and we know it can be, since our number system supports 6 powers of modulus 2, more than enough to support the value $1/2^3$. So here is an interesting point. If we're processing (in residue format) with this residue fraction, we have an exact value. It's only when we convert the value that we have the error, and that is good with respect to residue processing. Once again, this problem can be solved with another conversion other than the Print() method I've included for purest reasons.

What if we want to view the remainder? The Print() method allows us to do this, albeit as an approximation. To enable this feature, we un-comment the `#define PRINT_REMAINDER` in config.h file. This activates this feature. Printing the number again, we get the output:

```
        the value of fr: (V)+100.125000000~
```

In the output above, the remainder of the conversion calculation is extrapolated and is converted to two or three decimal digits for most cases. The tilde at the end of the number indicates the number is only approximate. In our simple case, these remaining digits are still zero! In this case, the conversion is holding its own, and is doing a good job. Later, we will print the value "ump", which needs remainder digits printed.

An interesting feature of the Print() method is that it converts to the end of the precision of the residue fraction. This is good. The number of decimal fraction digits (minus a few digits when ~ is present) is an indication as to the precision of the number, unless the number is exactly converted (no + or ~), in which case the number of fraction digits may be shorter than the precision. If we keep the setting of the Print() method so that remainder digits are not printed, we might see the + symbol at the end of the string. If we don't see a + sign, the fractional number is *likely* exactly converted.

Is our Print() method to be trusted? We only need to increase the precision of our system to answer this. Here is some sample code to produce the square root of the number 7 and print it using the Print() method. In this sample, we set `#define NUM_PPM_DIGS 54`, and we set `SPMF_FRACTION_DIGS 18`. Don't expect blinding calculations, there are a number of inefficiencies in these methods, but fast speed is not our objective in this release of RNS-APAL:

```
        SPMF *fr = new SPMF(0);
        fr->AssignFP("7");
        cout << "the value of fr: " << fr->Print(DEC) << endl;
        fr->Sqrt();
        cout << "the square root of fr: " << fr->Print(DEC) << endl;
        wait_key();
```

The output of the code above is:

```
        the value of fr: (V)+7.
        the square root of fr: (V)+2.645751311064590590501615753639+
```

With the chosen RNS precision settings, my Windows calculator only gives me only one more digit of precision. The output above shows that RNS-APAL can be very accurate indeed! The Print() method holds nicely as long as significant digits are present. What is phenomenal about this result is realizing the entire calculation is accomplished in residue format (using Newton's method), which represents fully modular digital arithmetic, without carry from digit to digit! This was not thought possible by many researchers! We'll re-visit the square root function later. Note that the AssignFP() method will accept integer (string) arguments, but these values are converted to fixed point fractions, albeit, the fraction portion is zero. In the example above, the value "7" is interpreted as "7.0".

Range Requirements of SPMF Types:

The *fractional range* of our P digit residue fraction with F number of fractional digits is:

$$R_F = m_0 * m_1 * .... * m_{F-1} \qquad \text{(Eq. 2)}$$

The *whole number* range of our P digit residue fraction is therefore:

$$W_R = m_F * m_{F+1} * ... * M_{P-1} \qquad \text{(Eq. 3)}$$

The *total range* of our fractional representation is therefore:

$$R_T = R_F * W_R \qquad \text{(Eq. 4)}$$

Therefore, the *extended range*, or "double width representation", must be that:

$$R_E = R_T^2 \qquad \text{(Eq. 5)}$$

In the relationship above, it is seen that we need at least a "range squared" representation for processing fractional digits. Normally, if we are using different modulus to implement our extended range residue number, we really need:

$$R_E > R_T^2 \qquad \text{(Eq. 6)}$$

The reason is that different modulus are not going to provide us with an equality, so we must exceed the square of our fractional number range of R_T.

What does this mean to RNS-APAL? It means we need to pay attention to the settings of NUM_PPM_DIGS and SPMF_FRACTION_DIGS defines in config.h. Failure to pay attention to these settings will yield incorrect results. RNS-APAL, unlike many arbitrary fixed radix arithmetic libraries, does NOT extend the number of digits to fit an increasing number range. We must set the maximum range of the number system at compile time using the config.h file. Future versions of RNS-APAL could address this issue, but in reality, this issue underlies one of the main differences of residue processing versus fixed radix processing.

The SPMF type holds the value of our fraction point position in two member variables, `int NumFractDigits`, which holds the current fraction point, and `int NormalFractDigits`, which holds the normal, or derived fraction point position. These member functions should be private, but these values can be returned to the user using `int GetCurFractPos(void) and GetNormFractPos(void)` methods. Note we do not expect the fraction point position to change, since RNS-APAL is a fixed point library at this time. However, some methods, most notably the scaling methods, will change the position of the fraction point internally during processing, but still returns a normal fraction position when complete. Future, more advanced versions of RNS-APAL may allow the user to dynamically change the fixed point position, but at this time, the user must change the fraction point at compile time. Changing the fraction point changes the fractional range, and therefore changes the precision of the SPMF fractional type.

Future versions of RNS-APAL might aid the user by automatically defining the number of PPM digits that are needed for any particular fractional format, but at this time, it does not. However, I have included several simple methods that help determine the range of your system. (You choose your system by setting the #defines discussed above in config.h). The first method is called "GetMaxRootFraction(SPMF *max_val); it returns the maximum value of an SPMF argument that when multiplied by itself will still produce a valid result. The function derives the "maximum" value by taking the square root of the maximum RNS range, and casting this value back into an SPMF type. Another maximum range function is the GetMaxRepFract(SPMF *max_val) which returns the maximum fractional representation for an SPMF type. However, even adding the smallest value will cause an overflow to this value, it's the largest positive fractional number represented. The last function is called GetUmp(SPMF *ump); this method returns the so defined "unit of most precision", which is the smallest fractional number that may be represented.

Let's get away from theory for the moment, and get on to show examples using the SPMF fractional residue class. In the examples below, I have set the `#define NUM_PPM_DIGS 16` and `SPMF_FRACTION_DIGS 5.` I have also set `#define ABS_SIGNED_RANGE NUM_PPM_DIGS` which means there are no redundant digits in our system. Finally, I have commented out `//#define PRINT_REMAINDER.` We can instantiate an SPMF type and print the maximum value in the fractional representation using the code below:

```
        PPM *range = new PPM(0);
        SPMF *fr = new SPMF(0);
        range->GetFullRange(range);
        fr->GetMaxAbsFraction(fr);
        cout << "max RNS magnitude: " << range->Print(DEC) << endl;
        cout << "max val as INT:    " << fr->PPM::Print(DEC) << endl;
        cout << "max val is:  " << fr->Print(DEC) << endl;
        fr->PPM::Increment();
        cout << "overflow is: " << fr->Print(DEC) << endl;
        wait_key();
```

The output of the code above is:

```
        max RNS magnitude: 328498717450075650878399
        max val as INT:    164249358725037825439199
        max val is:  (V)+7053930406318191.499999+
        overflow is: (V)(E)-7053930406318191.5
```

So what does the output of the code above tell us? In the first two lines, we instantiate an unsigned PPM integer value "range", and we instantiate an SPMF fractional variable "fr". The PPM range variable is provided to print out the entire range of the RNS system. This is done by calling the GetFullRange() method. In a similar manner, the maximum (positive) range of the fractional representation is provided by a call to GetMaxAbsFraction() method. The full RNS integer range is first printed. It shows the full "underlying" RNS range is a 24 digit decimal number. Next we print the maximum fractional residue value in terms of an integer value. Note that in order to do this, we cast the SPMF type as a PPM type, effectively treating the SPMF value as a PPM so we can use the PPM::Print() method. Next, we print the maximum absolute fractional type using the SPMF::Print() method. We get a 23 digit fractional format value with nearly a .5 fractional portion.

After the three print statements, we increment the fractional value "fr" by a single PPM unit, and we then print the fr value once again. Here we see a similar number is printed, but this time, the number has an (E) symbol printed as a prefix, which indicates the sign flag is in error. This was an intentional test. We provided this test to "push" the raw (positive) fractional value into the negative fractional range. We see the resulting number is the last negative number in the fractional range, which does have an exact .5 fractional portion. This is exactly what we should expect. Our range methods have provided us with the last numbers in a range, and we have shown that to be true using a simple test.

Looking back at the maximum residue fraction, we might consider what is the largest fractional format that can be multiplied by itself. If we use a calculator, we might surmise this value is the square root of the maximum positive fractional value. We would be wrong. The reason is we have not considered that fractional multiply needs to support an extra fractional range for the purpose of normalization, and therefore, RNS-APAL provides a method which returns the true maximum fractional value that when multiplied by itself, returns a valid result. The following code demonstrates this:

```
        SPMF *fr = new SPMF(0);
        printf("please wait, calculating square root of range ...\n");
        fr->GetMaxRootFraction(fr);
        cout << "raw int of the root is: " << fr->PPM::Print(DEC) << endl;
//      fr->Negate();
        cout << "the max root is: " << fr->Print(DEC) << endl;
//      fr->PPM::Increment();
//      fr->PPM::Decrement();
        fr->Mult(fr);
        cout << "the root squared: " << fr->Print(DEC) << endl;
```

The output of the code above is:

```
please wait, calculating square root of range ...
raw int of the root is: 573148076373
the max root is: (V)+24614.687537+
the root squared: (V)+605882842.568377+
```

The output of the code reveals a few things. For one, the square root process is accomplished entirely in residue format. It takes a few seconds of time, as the RNS-APAL library is not too fast at this initial release. The result of this range calculation is printed in both integer format as well as fractional format. Close analysis finds the maximum range of the fractional type is dependent on the square root of the underlying integer RNS representation. When we print the maximum range as a fractional type, we see the number that when multiplied by itself, will result in a valid fractional type. In the test code above, there are a few lines of test code commented out. For example, un-commenting the line "fr->Increment()", will produce the following result:

```
please wait, calculating square root of range ...
raw int of the root is: 573148076373
the max root is: (V)+24614.687537492+
the root squared: (V)+0.002104+
```

Here we see the maximum fractional root is exceeded by a mere single PPM unit. This causes the squaring of the (increased) root to overflow, and produce an incorrect result. The user may also test negative numbers by un-commenting the lines with "fr->Negate()" and with "fr->PPM::Decrement()". This produces a test of the negative range, which also overflows the capabilities of our number system. If we only uncomment the line "fr->Negate()", we get a correct result.

None of this is new. This is a result of fixed point fractional processing; we would see a similar result if we operated on fixed point values in decimal or binary, and we operated only on a fixed width representation. If you need to multiply larger values, the user must increase the range of the RNS system by adding digits using the NUM_PPM_DIGS setting in config.h. Another thing worth noting is the slight in-accuracy of the final result. This is also nothing new. If the user needs more accuracy, then the user must increase the number of digits in the fractional range of the residue fractional type. This is accomplished using the `SPMF_FRACTION_DIGS` setting in config.h.

For an example, let's increase the range of the RNS system by adding two digits to the fractional range, and four digits to the entire range. So, we have `#define NUM_PPM_DIGS 20` and `SPMF_FRACTION_DIGS 7`. We will use the same "old" values above and test the result:

```
SPMF *fr = new SPMF(0);
fr->AssignFP("24614.687537");
fr->Mult(fr);
cout << "the old root squared: " << fr->Print(DEC) << endl;
wait_key();
```

The result of the code above is:

```
the old root squared: (V)+605882842.54414312149+
```

Obviously, this result is more accurate, (My Windows calculator gives 605882842.544143126369) and thus it pays to maintain some redundant digits in the RNS representation to preserve the accuracy of the calculations. This type of in-accuracy arises because both the whole portion and the fractional portion have a large number of significant digits. Again, this is nothing new in a fixed point calculation, or even a fixed mantissa calculation (in floating point). The error is mainly a result of losing value and significance of the whole portion times the fractional portion. Again, choosing the number of fractional digits is important if we wish to have more accurate results. But if we increase the number of fractional digits, we (generally) must also increase the entire range of the RNS number so that overflow does not occur, so that the minimum operational range is supported during normalization.

The calculations above have more *precision* than a double float. However, a double float can provide more *range* of calculation. This is an interesting tradeoff that is well understood in a *fixed radix* arithmetic comparison. However this comparison is really an "apples to oranges" comparison with residue processing, since there is no time penalty for increasing precision of arguments for addition and subtraction (in hardware). Much more is needed to study the tradeoffs of very wide word processing in residues. However, very wide word residue processing is exactly what I'm advocating in terms of residue arithmetic. I believe it's the direction that takes most advantage of residue processing, and the PAC operations that provides so much hope to RNS researchers.

We saw some results of processing large residue values. What about small values? The RNS-APAL provides a simple function to provide us with the smallest value in our fractional representation, we refer to this value as "ump", or unit of most precision. Mathematically, we know this value exactly. It is given by:

$$\text{ump} = 1/R_F = 1/(m_0 * m_1 * .... * m_{F-1}) \qquad \text{(Eq. 7)}$$

where F is the setting of our `SPMF_FRACTION_DIGS.` Basically, ump in our system is the reciprocal of the product of all fractional digit modulus. In terms of a PPM value, it's a single unit. Using our decimal conversion method Print(), we can get a close approximation to this value. Let's use the last setting of `#define SPMF_FRACTION_DIGS 7`, and we also need to set `#define PRINT_REMAINDER.` Here is the sample code:

```
        SPMF *fr = new SPMF(0);
        fr->GetUmp(fr);
        cout << "ump is: " << fr->Print(DEC) << endl;
```

The output of the code above is:

```
        ump is: (V)+0.000000000000588~
```

According to my Windows calculator, I can find a more exact version of ump, but our converted value is very close. The Print() method was recently modified to provide more accurate least significant digits, but the user should be aware there is a possibility of error on the very last digit or so as it will not resolve the residue fraction beyond the estimated precision of the value. Moreover, the Print() method will print to the equivalent precision of the residue fractional number, convert a few extra digits to resolve any remainders of the conversion, and stop. The "588" printed in the value of ump above is entirely made up of "remainder" values during conversion. Therefore, the value "ump" is quite small.

General Arithmetic of SPMF fractional values

We spent considerable time explaining the SPMF AssignFP() and Print() methods, and we also introduced the range and precision settings of the residue fractional system. In so doing, we introduced the Mult() method in our examples a bit early. In this section, we formally visit the SPMF arithmetic methods, since it is these methods that form the backbone of general purpose arithmetic.

Fractional Residue Arithmetic PAC methods

It turns out that residue fractions take advantage of certain PAC (parallel array computation) arithmetic methods. These arithmetic operations are addition, subtraction, and multiplication by an integer (scaling). Because these operations are PAC, and because they are signed, we need only rely on the SPPM primitives to perform these operations. In order to make the syntax more readable, we have created SPMF versions of these functions which simply call their SPPM base class counterparts. Let's start by showing an example of the SPMF::Add() routine:

```
SPMF *a_fr = new SPMF(0);
SPMF *b_fr = new SPMF(0);
a_fr->AssignFP("100.25");
b_fr->AssignFP("200.75");
a_fr->Add(b_fr);
cout << "a + b = " << a_fr->Print(DEC) << endl;
wait_key();
```

The output of the code above is:

```
a + b = (V)+301.
```

The output shows what we expect. That is, the sum of 100.25 and 200.75 is the value 301. What might seem odd is how a carry can occur from the fractional portion to the whole portion. The answer is in the encoding of the fractional SPMF value. Our SPMF fraction is like any fixed point fraction, that is, the whole portion is "scaled" by the fractional range. For example, given the decimal fraction 100.25, we see that the whole portion "100" is essentially 100 times 100/100. In other words, the value 100 is really 10000/100. When we add the fractional portion to the whole portion, we add 25/100. The register in our computer (or variable) doesn't store the 100 in the denominator, it's implied (its defined by the fraction point position). The value stored in our register is simply 10025. The same holds true for our fractional residue representation, that is, the whole value of our fractional residue is scaled, or multiplied, by our fractional range, where we add the fractional portion without scaling. The math is simple:

For any residue fraction Y, consisting of a whole portion "w" and a fractional portion n, we "build" or encode our fractional residue representation using the equation:

$$Y = (w * R_F) + n \qquad \text{(Eq. 8)}$$

The term R_F is our fractional range as defined in an earlier section. It is by means of this "encoding" that any fractional representation exists, and fractional encoding is why addition works without carry from digit to digit in residues. Let's look at a subtraction example:

```
SPMF *a_fr = new SPMF(0);
SPMF *b_fr = new SPMF(0);
a_fr->AssignFP("100.25");
b_fr->AssignFP("200.75");
a_fr->Sub(b_fr);
cout << "a - b = " << a_fr->Print(DEC) << endl;
wait_key();
```

The output of the code above is:

```
a - b = (I)-100.5
```

The result is what we expect, that is, the value of 200.75 subtracted from 100.25 is -100.5. However, we see the Print() method indicates the sign flag is invalid. In an earlier section we discussed this possibility, that is, when adding or subtracting numbers, it is possible to lose the state of the sign flag. This example is such a case. The print() method is powerful, as it determines the true sign of the result based on magnitude comparison, and also prints the state of our sign tracking. The Print() method provides us the whole story. It should be noted that the result, despite its invalid flag status, can be directly entered into many other residue operations which still yield a valid result. The following code shows this:

```
SPMF *a_fr = new SPMF(0);
SPMF *b_fr = new SPMF(0);
SPMF *c_fr = new SPMF(0);
a_fr->AssignFP("100.25");
b_fr->AssignFP("200.75");
c_fr->AssignFP("123.5");
a_fr->Sub(b_fr);
a_fr->Add(c_fr);
cout << "a - b + c= " << a_fr->Print(DEC) << endl;
wait_key();
```

The output is:

```
a - b + c = (I)+23.
```

Note that once an argument's sign is invalid, it will cause an invalid flag in any additional result for the operations of add, subtract, and multiply by an integer. It is only when a value with an unknown sign needs to be scaled or divided is it imperative that the sign of the argument be known, or "extended". In RNS-APAL, operations that require the sign to be known will automatically perform this operation. However, we can also manually perform a sign extend by calling the SPPM::CalcSetSign() method, as introduced earlier. The following example shows the use of this function to clean up the sign flag of our result:

```
SPMF *a_fr = new SPMF(0);
SPMF *b_fr = new SPMF(0);
SPMF *c_fr = new SPMF(0);
a_fr->AssignFP("100.25");
```

```
        b_fr->AssignFP("200.75");
        c_fr->AssignFP("123.5");
        a_fr->Sub(b_fr);
        a_fr->Add(c_fr);
        a_fr->CalcSetSign();
        cout << "a - b + c= " << a_fr->Print(DEC) << endl;
        wait_key();
```

The output then becomes:

```
        a - b + c = (V)+23.
```

The (V) above indicates the sign is set to positive, and it is also flagged as valid. Since there is no (E) error symbol, we know the Print() method agrees with the sign flag settings.

We mentioned the operation of multiplying a fractional value by an integer value. This is yet another PAC operation that fractional residue types benefit from. There are two overloads, one which takes an __int64 argument, and the other a pointer to an SPPM argument. The overload taking the __int64 argument is a helper function. Let's show an example of the more pure method:

```
        SPMF *a_fr = new SPMF(0);
        SPPM *scale = new SPPM(0);
        a_fr->AssignFP("100.2575");
        scale->Assign("12");
        a_fr->Mult(scale);
        cout << "a * scale = " << a_fr->Print(DEC) << endl;
        wait_key();
```

The output of the code above is:

```
        a * scale = (V)+1203.09
```

The output is what we expect. The multiplication of a residue by an integer is a powerful operation which may underlie other operations, such as square root. In hardware, this operation is very fast since it may occur in a single clock cycle, regardless of the precision of the operands. Closer look at the source code shows us that all the SPMF PAC operations are calls to the equivalent SPPM method. In other words, we didn't need any SPMF class methods for our PAC operations, we only needed to call the methods SPPM::Add(), SPPM::Sub(), and SPPM::Mult(). However, RNS-APAL includes methods for basic SPMF arithmetic, so the user need not remember to call the base class methods. What this shows is that PAC operations are essentially integer operations, where we treat the fractional type as an integer. This means the manipulation of the underlying signed integer represents a powerful method in the manipulation of more complex representations, such as residue fractions. More about this later.

What if we simply want to increment a fractional value by one or more whole units? Can we use the PPM:Increment() method? The answer is no. Fractional residue values are scaled by the fractional range, so we must account for that. If we want to add a single whole unit to a fractional value, we should add a single fractional range. RNS-APAL provides two helper methods to make the increment process easier. Here is some sample code:

```
        PPM *range = new PPM(0);
        SPMF *fr = new SPMF(0);
        fr->AssignFP("100.25");
        fr->Add1(1);
        cout << "the result is: " << fr->Print(DEC) << endl;
        fr->Sub1(1);
        cout << "the result is: " << fr->Print(DEC) << endl;
        wait_key();
```

The output of the code above is:

```
        the result is: (V)+101.25
        the result is: (I)+100.25
```

Keep in mind that the rules for invalidating flags still holds as it does for the underlying SPPM method. As an example in the output above, incrementing a positive value will maintain the variables flag status, while decrementing a positive SPMF value will invalidate it's sign flag. Use the SPMF::Add1() and SPMF::Sub1() methods for quick incrementing or decrementing of SPMF types, or alternatively, assign the desired increment to another SPMF variable and use the basic SPMF::Add() or SPMF::Sub() methods.

Direct Multiplication of SPMF Residue Fractions

We are now ready to discuss one of the most important "slow" operations in RNS-APAL, that is, the multiplication of two SPMF fractional types. When we think about it, the process of multiplying a fractional type with another fractional type has two components; there is some multiplication going on, and there is some division going on. For example, the whole portions are multiplied by one another, which increases the value, whereas each whole portion is also multiplied by a fractional portion, which "divides" down by each whole portion, and finally all results are added together. As another example, if we multiply the multiplicative inverse of a number with another number, we have essentially performed a division operation. So to be sure, the SPMF multiply method is quite powerful. We regard the multiplication operation of fractional types as the most important of all general purpose arithmetic operations, and indeed, it forms the basis for general purpose computation.

By slow, we are referring to a method which is linear with respect to the number of RNS digits, in hardware. In software, the operation is of the $O(n^2)$ variety. This relates in nearly the same way as a digit oriented hardware fixed radix multiplier, or digit oriented software fixed radix multiplier. This is good, since the purpose of residue processing is to take advantage of PAC operations while maintaining the standard speed of fractional operations, such as multiply.

There are several variations of the SPMF multiply operation. Some of these are included to benefit the researcher. During the years of development, more than one technique of fractional multiplication has been developed, but mathematically, they all serve to normalize the result, much like we do when we multiply two fixed point fractional values in decimal. If you recall the procedure of multiplication of decimal fractions, we must count the number of total digits past the decimal place, and then place the decimal place that many positions to the left of the least significant digit. This is normalization in fixed radix format. Furthermore, if we insist that the final number "fit" into the same representation as the starting operands, we truncate some portion of the least significant fractional digits. If we're clever, we take a look at the truncated least significant digits and perform a round up if the value of the truncated digits are closer to the fractional range than to zero. The SPMF multiply does exactly this operation. Mathematically, we have the multiplication of Z = X * Y, (all residue fractions) as:

$$Z/R_F = X/R_F * Y/R_F = (X * Y)/(R_F * R_F) \qquad \text{(Eq. 9)}$$

$$(X * Y)/(R_F * R_F) = ((X * Y)/R_F)/R_F \qquad \text{(Eq. 10)}$$

In the equation X1 above, R_F is the fractional range as defined previously. This value is an implied value, and is represented by the fraction point position. In X1, the multiplication of residue fractions results in an integer multiply of X and Y, but with an implied fractional range "squared" in the denominator. This would not be in the proper format. But the resulting value is important, and is called the "intermediate product". To obtain the correct result, we must normalize the intermediate product, which is described in equation X2. In essence, we must divide the integer product by the fractional range in order to normalize the intermediate product. One way to do this is to use the integer divide method developed for RNS-APAL. This would be slow, and difficult to pipeline in hardware. A more effective approach is explained in US patent application US 2013/0311532 A1 (among others I have authored).

As described in this publication, a unique and novel approach to fractional residue normalization involves converting the intermediate product to mixed radix, truncating the mixed radix digits associated with the fractional range R_F, and then re-converting the remaining mixed radix number back to residue format. In this operation, the mixed radix digits are simply a container for the intermediate result, albeit, in a weighted number format, which allows easy division by truncation. When the truncated mixed radix number is reconverted to residue format, our normalization is essentially complete, i.e., we have a normalized, fractional result. However, the result has not yet been rounded. The patent application explains several variations for normalization and rounding, including using the integer divide method, and using several advanced forms of mixed radix conversion.

In addition, a new technique is invented for the process of using mixed radix conversion based normalization for determining the sign of the result despite knowing anything about the sign of the operands! By taking the complement of the intermediate value (a PAC operation), and by converting both the original intermediate value and its complement simultaneously, we can perform an integrated comparison of both values. If the complemented intermediate value is found to be lesser, it implies the original value was negative. We can only keep the lesser (in magnitude) of the comparison for the normalization process, since *only positive values can be divided*. If the complemented value is less, its magnitude is positive, and once we re-convert back to residue, we re-complement the value, since we know the result *should be* negative. On the other hand, if the original intermediate product is the lesser in magnitude, it is truncated, and re-converted to residue, and we know the final answer is positive. Our software (and hardware) sets the sign flags accordingly. Therefore, the new fractional multiplier will *generate valid sign flags* for the result *without any knowledge of the signs of the operands*. *This is of huge importance to general purpose processing in residues*.

For rounding, during the process of mixed radix conversion, a secondary comparison is made on the fractional digits only, against half the fractional range ($R_F/2$). This secondary comparison is performed twice, simultaneously, since we do not know if the original intermediate value, or its complement, will be selected. If the secondary comparison of the selected quantity shows the fractional portion is greater (or equal) to the fractional range, the final residue result is rounded up by adding a single "ump" value before any possible complementing operation. Other methods also exist for rounding.

So to wrap up a lengthy story, the patent application describes many variations of these multiplication processes, and I have implemented several of them in RNS-APAL for researchers to investigate. Furthermore, a more advanced technique, which re-converts mixed radix digits directly back to residue format as they are generated is also included. This method saves clock cycles in hardware. This is described in patent application US patent application US 2013/0311532 A1.

Because of the various methods, I've created a container method called MultStd(). The user should use this method for general use. This method contains calls to several various fractional methods commented out. Un-comment only one of the methods for the purpose of using that particular fractional multiply method. Refer to the comments for each method to get an ideas as to the properties and algorithm specifics of each.

OK, let's demonstrate the MultStd() method with a few examples. For this example, we'll use the settings of `#define NUM_PPM_DIGS 18` and `#define SPMF_FRACTION_DIGS 6`, we'll also comment out `//#define PRINT_REMAINDER`, and finally we'll comment in the Mult4() method in MultStd() located in the SPMF.cpp file.

```
        SPMF *a_fr = new SPMF(0);
        SPMF *b_fr = new SPMF(0);
        a_fr->AssignFP("100.2575");
        b_fr->AssignFP("987.654");
        a_fr->MultStd(b_fr);
        cout << "a * b = " << a_fr->Print(DEC) << endl;
        cout << "the raw rns result is: " << a_fr->Prints(DEC) << endl;
        wait_key();
```

The output of the code above is:

```
        a * b = (V)+99019.7209051+
        the raw rns result is: (V)+2 17 14 27 9 11 . 10 14 7 1 11 27 21 4 32 35 48 11
```

The output is what we should expect. In addition, we have also printed the result in its native rns format. Notice that a fraction point is printed. The fraction point is used to separate the RNS digits associated to the fractional range from the remaining RNS digits. Note that RNS-APAL prints fractional digits first. The Prints() method does not attempt to check the validity of sign flags in the same way the Print() method does because it only prints the state of the sign flag and string of RNS digits. Now let's modify the code to invalidate the sign flags of the arguments and change the arguments to a negative value:

```
SPMF *a_fr = new SPMF(0);
SPMF *result = new SPMF(0);
SPMF *b_fr = new SPMF(0);
a_fr->AssignFP("-100.2575");
a_fr->SignValid = FALSE;
result->Assign(a_fr);
b_fr->AssignFP("-987.654");
b_fr->SignValid = FALSE;
result->MultStd(b_fr);
cout << a_fr->Print(DEC) << " * " << b_fr->Print(DEC) << " = " << result->Print(DEC) << endl;
cout << "the raw rns result is: " << result->Prints(DEC) << endl;
wait_key();
```

The output of the code is:

```
        (I)-100.2575000+ * (I)-987.6540000+ = (V)+99019.7209051+
        the raw rns result is: (V)+2 17 14 27 9 11 . 10 14 7 1 11 27 21 4 32 35 48 11
```

RNS-APAL supports a fractional multiplication routine that *generates* valid sign flag information even if the operand signs are not known. This is not simply a call to a sign extend routine, the fractional multiply integrates the sign extend operation as fundamental step. This allows the fractional multiply to deal with any operand combination, and it also allows a profound increase in processing speed as we'll talk about next. In my view, this subtle but important capability now makes general purpose residue processing practical.

Intermediate Product to Normal Conversion

If we take a look at the source code of the fractional multiplication methods, we see the first operation is an *integer multiply* of both fractional arguments. This is also shown in equation XX above, i.e., X * Y is an integer multiply. This integer multiply is a PAC operation. To be clear, the routine treats the fractional values as integers, and performs an integer multiply creating a product result. I refer to the product as an *intermediate product*. However, as discussed, the resulting product is not normalized. This is the main time consuming process of the fractional multiplication.

One of the great advantages of residue arithmetic is the ability to separate *complex operations* into separate PAC operations and longer normalization operations. This allows for a number of improvements including greater speed, better accuracy, and integrated sign extension. Two basic arithmetic operations that benefit from this separation is multiply and accumulate (MAC) and product summation.

For multiply and accumulate (MAC), we might expect that the fractional multiply be a long operation, and any subsequent addition operation be a PAC operation anyways, regardless of any ability to separate PAC from normalization. So what is the benefit? The benefit for MAC operations is to recover the final state of the sign flags. If you recall from our earlier discussions, if we execute a fractional multiply operation, we get a fully sign extended result. However, if we add a value to that result, we might lose that sign information. By integrating the multiply and add operations at the PAC level, and then apply a final normalization of the intermediate product and addition, we get a fully sign extended result for any combination of operands!

To demonstrate this idea, we introduce the intermediate to normal conversion method, called I2N_Convert(). This routine is actually based on the MultStd() methods in RNS-APAL; thus, we can use any of the available multiply algorithms for the normalization method. RNS-APAL did not need to be structured in this way. I could have included any number of different normalization methods as separate methods. However, the normalization procedure *is the bulk* of the MultStd() procedure. By simply calling the MultStd() routine with an argument of one unit (ump), we "by-pass" the integer multiply of each fractional multiply algorithm (since any value multiplied by one is itself). So essentially, we take advantage of only the normalization portion of the fractional multiply method. Another way to think about this is to view the MultStd() method as a combination of integer multiply and I2N_Convert() normalization. It might have been more instructive to code it this way, and perhaps future versions of RNS-APAL will re-organize code this way.

Let's first demonstrate the I2N_Convert() normalization method by performing a fractional multiplication at a more primitive level:

```
SPMF *a_fr = new SPMF(0);
SPMF *b_fr = new SPMF(0);
a_fr->AssignFP("100.25");
b_fr->AssignFP("35.67");
a_fr->PPM::Mult(b_fr);
cout << "the intermediate product: " << a_fr->PPM::Print(DEC) << endl;
a_fr->I2N_Convert();
cout << "the normalized fraction is: " << a_fr->Print(DEC) << endl;
wait_key();
```

The output of the code is:

```
the intermediate product: 327656825474263084800
the normalized fraction is: (V)+3575.9175000+
```

In the code above, the intermediate product is formed through a PPM::Mult() method, and is printed as an unsigned PPM variable using the PPM::Print() method. This is correct. The intermediate product is not known to be positive or negative, and is not normalized, so it is appropriate to treat the intermediate product as a PPM variable. This is how advanced RNS-APAL users will treat any such intermediate product. However, once a (fractional) normalization is performed, the resulting value is a valid signed SPMF variable, and may be processed as any normal SPMF value and printed using the SPMF::Print() method.

Multiply and Accumulate (MAC)

Now let's look at the MAC operation discussed above. To demonstrate, first we'll use a separate multiply method then apply a separate add method:

```
SPMF *a_fr = new SPMF(0);
SPMF *b_fr = new SPMF(0);
SPMF *c_fr = new SPMF(0);
a_fr->AssignFP("100.25");
b_fr->AssignFP("35.67");
c_fr->AssignFP("-43.98");
```

```
        a_fr->MultStd(b_fr);
        a_fr->Add(c_fr);
//      a_fr->MAC(b_fr, c_fr);
        cout << "the multiply then add result is: " << a_fr->Print(DEC) << endl;
        wait_key();
```

The result of the code above is:

```
        the multiply then add result is: (I)+3531.9375000+
```

Note that while the result is correct, the sign flag is invalid. We learned about this side effect earlier, when we demonstrated that a PAC addition can invalidate the sign flag of the result in certain cases. Now let's show the effect of using the MAC operation which has been commented out above. Here is the new code:

```
        SPMF *a_fr = new SPMF(0);
        SPMF *b_fr = new SPMF(0);
        SPMF *c_fr = new SPMF(0);
        a_fr->AssignFP("100.25");
        b_fr->AssignFP("35.67");
        c_fr->AssignFP("-43.98");
//      a_fr->MultStd(b_fr);
//      a_fr->Add(c_fr);
        a_fr->MAC(b_fr, c_fr);
        cout << "the MAC result is: " << a_fr->Print(DEC) << endl;
        wait_key();
```

The result of the code above is:

```
        the MAC result is: (V)+3531.9375000+
```

Note that we've substituted the MAC operation for the two separate operations of multiply and add. The result is the same, but the sign of the result is now known. In other words, by integrating the PAC addition operation with the PAC multiply operation, and then normalizing, we get a valid result with a valid sign flag. This is important, since many times in arithmetic, we need to multiply a quantity, then add a quantity, and if we follow with a compare of divide, the number of operations may be reduced. The MAC operation doesn't increase speed by itself, but may increase performance by eliminating the need to sign extend a result later. If we look at the source code for the MAC instruction, we see it uses the I2N_Convert() method after using basic PAC operations of multiply and add. However, before the add operation, we must convert the additive operand to an intermediate format. To do this, we multiply the additive operand by the fractional range using a simple PAC multiply.

Fractional Product Summation in Residues

A more impressive example is that of product summation. Product summation forms the core of many AI and scientific processing algorithms. The product summation forms the new motivation to persue residue based ALU's and CPU's. In the following example, we will perform a product summation at a low level, which clearly demonstrates the separation of PAC methods and normalization. In this example, we will perform a product summation of two pairs of fractional numbers:

```
        SPMF *a_fr = new SPMF(0);
        SPMF *b_fr = new SPMF(0);
        SPMF *c_fr = new SPMF(0);
        SPMF *d_fr = new SPMF(0);
        a_fr->AssignFP("100.25");
        b_fr->AssignFP("35.67");
        c_fr->AssignFP("-43.98");
        d_fr->AssignFP("1987.345");
        a_fr->PPM::Mult(b_fr);
        c_fr->PPM::Mult(d_fr);
        a_fr->PPM::Add(c_fr);
        a_fr->I2N_Convert();
        cout << "the product summation is: " << a_fr->Print(DEC) << endl;
        wait_key();
```

The result of the code above is:

```
        the product summation is: (V)-83827.5156000+
```

When we execute the code above, the result is correct, and it has a valid sign flag. What is amazing is the bulk of the arithmetic product summation operations are PAC! These operations are performed using PPM::Mult() and PPM::Add() methods. Any number of product summations can be performed using only PAC operations and which require only a single I2N_Convert() method at the final stage. This means that complex operations, such as dot products, can be performed much quicker, more efficiently, and more accurately.

The reason for high accuracy is the there is no loss in precision during the PAC operations, since all product summations are performed in a fully extended "double" precision, where the only loss occurs when the final intermediate product summation value is normalized, resulting from a single rounding operation. Note that any number of products may be summed, each product only requiring a single PAC multiply, and each summation requiring only a PAC addition. This feature of residue processing may ultimately prove important to scientific processing, artificial intelligence, and other emerging scientific processing methods requiring precise, fast and accurate product summations and matrix operations.

Division of SPMF Fractional Values

We now move on to fractional division. Division is one of the slower methods in most number systems, and is typically avoided in many algorithm designs, however, the need to perform division is still paramount. I would say efficient division completes a *core set of arithmetic* for RNS. As an RNS researcher, I'm always looking for ways to make division faster.

One method for fractional division relies on using a method for performing arbitrary integer division. If we use our representation for fixed point fractions, we can derive the necessary operations for fractional division. For this analysis, we will perform the fractional operation N / D = Q:

$$Q/R_F = (N/R_F) / (D/R_F) = N/R_F * R_F/D = [(N * R_F)/D]/R_F \qquad \text{(Eq. 11)}$$

Once again, the term R_F is the fractional range of our fixed point number. This value is an implied value when it occurs in the denominator of our calculations, since this value is implied by the position of the fraction point. We must re-arrange terms for the final answer such that a single power of the fractional range occurs in the denominator, which "normalizes" our result. This is done algebraically on the far right hand side of equation yy. We find by our re-arrangement we must multiply N by the fractional range, R_F, and (then) divide by D. This is how integer division is used to perform fractional division. During this process, if we take the remainder of the integer division process, and compare this value against half the divisor value, we can determine if a "round up" is required. Alternatively, it's easier in RNS if we take the remainder, multiply it by two, and compare this directly to the divisor. If it's equal or greater, a round up is performed by adding *ump* to the final value before any possible complementing (if the result is negative).

Note there is a range requirement for our calculation even if we consider the integer divide method requires no additional range, i.e., no redundant digits. This range requirement is derived from the fractional value N, that when treated as an integer and multiplied by the fractional range R_F, does not overflow the underlying representation (PPM number). Because the fractional range, R_F, in the denominator is implied, the actual calculation to be performed is the numerator portion, the value in brackets: $[(N * R_F)/D]$. The user is encouraged to view the source code for fractional reside division.

RNS-APAL provides a basic fractional division procedure called SPMF::Div(), which uses the integer divide method PPM::DivStd(). The Div() method requires a single argument: SPMF::Div(SPMF *divisor); thus, the value being divided calls the Div() method with the desired fractional divisor. Unlike the integer divide methods, there is no remainder argument passed. Result rounding is performed automatically. This method for fractional division in residue format using integer division is briefly considered in US Patent application US 2013/0311532 A1.

Let's provide a code example of this method of fractional division. We will stick with the same settings in the config.h file as before, that is `#define NUM_PPM_DIGS  18, and SPMF_FRACTION_DIGS  6`:

```
SPMF *num = new SPMF(0);
SPMF *div = new SPMF(0);
SPMF *quot = new SPMF(0);
num->AssignFP("1060.75");
div->AssignFP("65.78");
quot->Assign(num);
quot->Div(div);
cout << num->Print(DEC) << " divided by " << div->Print(DEC) << " is " << quot->Print(DEC) << endl;
cout << "the raw quot: " << quot->Prints(DEC) << endl;
wait_key();
```

The result of the code above is:

```
        (V)+1060.75 divided by (V)+65.78 is (V)+16.1257221+
        the raw quot: (V)+63 18 8 43 2 1 . 4 14 22 22 1 1 13 14 38 30 52 58
```

As expected, the result is correct. A couple of things are worth noting. For one, the divide procedure will not rely on the sign flags. This procedure will calculate the sign flags and will ensure the magnitudes are correctly divided according to the method of complements notation. I didn't have to do this. I might have relied on sign flags. The user can write another version which does rely on sign flags. However, if the sign flag is invalid, since the sign of either operand must be known prior to division, a sign extend operation will be needed for any operand with an unknown sign.

It's worth noting the Div() method is exact. Given a particular internal fractional representation (in residue), the division method provides the closest answer possible. There is no approximation, since the method is powered by an integer divide operation. Unfortunately, the method is slow, since the integer divide routine is the slowest of all basic arithmetic routines. Don't get confused by the accuracy of the examples which use the Print(DEC) conversion, since converting operands *into residue* from a fractional decimal notation, and furthermore, printing the output using a fractional decimal notation is only approximate. For any representation *inside* our residue number system, things are processing very accurately, just as accurate as any other number system of the same approximate precision.

Before we explore an alternative to the Div() method, let's first review the fractional Inverse() method, which is also powered by the integer divide method.

```
        SPMF *num = new SPMF(0);
        SPMF *copy = new SPMF(0);
        num->AssignFP("210");
        copy->Assign(num);
        num->Inverse();
        cout << "inverse of " << copy->Print(DEC) << " is " << num->Print(DEC) << endl;
        cout << "raw inverse: " << num->Prints(DEC) << endl;
        num->MultStd(copy);
        cout << "test: " << num->Print(DEC) << endl;
        wait_key();
```

The output of the code above is:

```
        inverse of (V)+210. is (V)+0.0047619+
        raw inverse: (V)+32 18 15 7 0 0 . 10 5 7 24 2 31 3 37 44 52 11 10
        test: (V)+1.
```

For the example above, we chose a fractional value that would produce a perfect fractional inverse. In this case, we chose the value 210, which is a product of the first four modulus (2, 3, 5 & 7). When we calculate the inverse, we get an exact inverse, since one property of the fractional residue system is that it can represent many different denominators in its fractional range. In many cases, we can detect such perfect fractions by seeing one or more zero's in the fractional digits of the raw residue number. Any product of the modulus and their powers associated with the fractional range will result in an exact fractional inverse. When we multiply the fractional inverse by the original value, we get an exact value of 1.0.

Goldschmidt Division and Fractional Scaling

But what if we're looking for a faster fractional divide routine, could we produce one? The answer is yes. One common method for performing division is to use Newton's method, or to use the similar Goldschmidt method. In these methods, we use fractional multiplication to perform fractional division.

In RNS-APAL, we use the Goldschmidt method to perform fractional division. In future revisions, the Newton's approach may also be added.

In binary, before performing the division process, we must scale the divisor to a value less than one, otherwise, the Goldschmidt algorithm will not converge. In particular, because binary is base two, it is possible to scale the divisor to a value between 0.5 and 1.0. Since the goal of Goldschmidt division is to multiply the numerator and divisor by a constant until the divisor converges to 1.0, it is advantageous that the divisor starts as close to 1.0 as possible.

Two issues are presented to residue fractions when developing a fractional division routine. The first problem is developing a scaling routine and the second issue is developing a scaling routine that provides an initial scaled value as close to 1.0 as possible. In binary, a simple shifting procedure is used to accomplish the goal of providing an initial scale of the divisor between 0.5 and 1.0. In US patent application US 2013/0311532 A1, the method of scaling a residue fraction is provided and discussed. This guide will not delve too far into these procedures, but we show the operation of at least one of the scaling procedures first. Before we do that, it should be noted that to scale a residue fraction to a value between 0.5 and 1.0, we must ensure that a two's modulus is supported, and that the two's modulus is the largest modulus of the RNS system. Fortunately, because the power two modulus fits perfectly into a binary digit encoding, this is always possible. When selecting the automatic RNS generation feature of RNS-APAL, and making sure the power based modulus is enabled, the power two's modulus is always the largest modulus in the system. This is a requirement for Goldschmidt division method to work.

RNS-APAL V0.1 includes several scaling methods, but they all essentially do the same thing. Some may not work. This is a result of the Goldschmidt algorithm being under development. Therefore, I have included a wrapper method called scalePM(), which should call the best performing method. A warning to the user is that we are showing off advanced routines which are essentially internal routines. In other words, the scalePM() routine is not a top level routine, but an internal routine to be used by Goldschmidt and Newton division methods. Therefore, we need to use special routines to print non-normal values, which are values with a non-normal modulus, and a non-normal fraction point position. But having said this, the RNS researcher should be interested in this advanced topic, and the RNS-APAL library can show off some of these functions.

The PrintAbsPM() routine was designed to allow the printing of non-normal fractional formats, but it still has bugs, so we have reverted back to an earlier, more simple print routine based on double floating point numbers, called PrintFPM(). Therefore the user should take care not to extend the fractional range too far, or the PrintFPM method will fail (it will exceed double float precision).

Not that we've stated many of the caveats, let's see an example of the ScalePM() method. Again, we've used the settings `#define NUM_PPM_DIGS  18` and `#define SPMF_FRACTION_DIGS  6`:

```
    int width;
    SPMF *num = new SPMF(0);
    SPMF *div = new SPMF(0);
    num->AssignFP("1060.75");
    div->AssignFP("65.78");
```

```
        div->scalePM(div, num, width);
        printf("div: %f\n", div->PrintFPM());
        printf("num: %f\n", num->PrintFPM());
        cout << "raw div: " << div->PPM::Prints(DEC) << endl;
        cout << "new two's modulus power: " << width << endl;
        cout << "new fraction point is: " << div->GetCurFractPos() << endl;
        wait_key();
```

The output of the code above is:

```
        (V)+div: 0.814613
        (V)+num: 13.136223
        raw div: X|0 0 22 0 0 0 13 4 0 9 19 12 15 4 22 51 15 34
        new two's modulus power: 4
        new fraction point is: 8
```

Note that if we divide the scaled values, we get the same ratio as the two original fractional numbers. So our scaling routine is working. Also note that divisor is a number between 0.5 and 1.0. If the user tries other numbers, similar results will be obtained. Don't be surprised if a few errors are printed by the PrintFPM() methods, since these methods are expecting a fully normalized value for the purpose of sign detection, but the magnitude printed should be correct. Note the scaling routine may modify two properties of our fractional residue representation, namely the two's modulus may be changed, and the fractional point position may be increased. In our example, both are changed; the two's modulus power has been reduced from 6 to 4, and the fraction point position has been increased from 6 to 8. The scalePM() method is designed to guarantee a minimum value for the fraction point to be at least the original normal fraction point. The two's modulus power may be modified to be as small as 1. The actual digits of the number are NOT changed. This is similar to binary shifting of the fraction point. The scalePM() method only modifies the two's modulus power and the fraction point position.

If the user is interested in some debug output of the scaling method, simply replace the scalePM() method with scalePM6(). This version includes a step by step print of the measurement of the divisor, and the decisions made to adjust the divisor to be a value between 0.5 and 1.0.

Now let's show an example of the Goldschmidt division method, called GoldDiv(). The GoldDiv() method essentially operates on the scaled operands as shown in the previous example. Therefore, this routine is complex in that it must operate on non-normal modulus and non-normal fraction point position. The GoldDiv() method accepts a single divisor parameter as did the SPMF::Div() method.

```
SPMF *num = new SPMF(0);
SPMF *div = new SPMF(0);
SPMF *quot = new SPMF(0);
num->AssignFP("1060.75");
div->AssignFP("65.78");
quot->Assign(num);
quot->Div(div);
cout << num->Print(DEC) << " divided by " << div->Print(DEC) << " is " << quot->Print(DEC) << endl;
wait_key();
```

The result of the code above is:

```
(V)+1060.75 divided by (V)+65.78 is (V)+16.1257221+
the raw quot: (V)+62 17 7 42 1 0 . 3 13 21 21 0 0 12 13 37 29 51 57
```

Note that the GoldDiv routine provides a result that is similar, but not exact to the Div() method demonstrated earlier. This is a consequence of the routine's accuracy used to perform the Goldschmidt algorithm and using a fixed point fractional representation. The result is still highly accurate. However, future improvements, similar to those used by IBM for its 370 series floating point Goldschmidt routine, will be added to the RNS-APAL library later. This will, in most cases, provide an exact result.

The user should also be aware of certain combinations of operands that may lead to values that are out of range, or even values that are significantly skewed, which lead to additional in-accuracy. These issues are the same issues any number system would experience when using Goldschmidt with a fixed precision.

We will close our section on Goldschmidt division with an example of *fractional format* normalization. By normalization, we mean the re-normalizing of the number system format; this should not be confused with the term normalization when used with *intermediate product* normalization, i.e., a key aspect to fractional multiplication. RNS-APAL includes a method called NormFract() which performs fractional format normalization; the user should be aware this method is again used internally by the Goldschmidt() routine. However, we can demonstrate its use by applying the normalization to the scaled values in the previous demo:

```
int width;
SPMF *num = new SPMF(0);
SPMF *div = new SPMF(0);
num->AssignFP("1060.75");
div->AssignFP("65.78");
div->scalePM(div, num, width);
printf("div: %f\n", div->PrintFPM());
cout << "raw div: " << div->PPM::Prints(DEC) << endl;
cout << "new two's modulus power: " << width << endl;
cout << "new fraction point is: " << div->GetCurFractPos() << endl;
div->NormalFract();
printf("\n");
printf("norm div: %f\n", div->PrintFPM());
cout << "norm raw div: " << div->PPM::Prints(DEC) << endl;
cout << "norm two's modulus power: " << div->Rn[0]->PowerValid << endl;
cout << "norm fraction point is: " << div->GetCurFractPos() << endl;
wait_key();
```

The output of the code above is:

```
(V)+div: 0.814613
raw div: X|0 0 22 0 0 0 13 4 0 9 19 12 15 4 22 51 15 34
new two's modulus power: 4
new fraction point is: 8

(V)+norm div: 0.814613
norm raw div: 31 8 11 14 5 11 5 5 11 22 27 32 36 5 46 2 3 9
norm two's modulus power: 6
norm fraction point is: 6
```

As can be seen from the example code, the scaled value from the earlier example has been normalized so that is has a normal fraction point position, and normal two's power modulus. This normalization procedure is used to adjust the output of the Goldschmidt routine so that the final answer takes on a normal fractional representation.

Example High Level SPMF Fractional Routines

In this section, we visit high level routines that may be constructed from lower level routines. In this section, we explore an example square root method, called Sqrt(). The Sqrt() method uses Newton's method to calculate the square root of a fractional residue type. This high level routine illustrates the use of immersed fractional residue arithmetic. Since Newton's square root method is iterative, we know the accuracy of residue arithmetic is holding, and just as well as any other fixed radix number type. Here is some sample code of using the Sqrt() method:

```
SPMF *fr = new SPMF(0);
fr->AssignFP("12345.6789");
cout << "the value of fr: " << fr->Print(DEC) << endl;
fr->Sqrt();
cout << "the square root of fr: " << fr->Print(DEC) << endl;
fr->Mult(fr);
cout << "the squared root of fr: " << fr->Print(DEC) << endl;
wait_key();
```

The output of the code above is:

```
the value of fr: (V)+12345.6789000+
the square root of fr: (V)+111.1111106+
the squared root of fr: (V)+12345.6789000+
```

The code above shows the square root calculation, and also shows the result of the calculation when the root is squared. The user is encouraged to view the source code of the `SPMF::Sqrt()` method. This is a prime example of using the RNS-APAL library to construct high level arithmetic operations.