

GENERALIZED FIBONACCI NUMBERS LIBRARY

HP49/49G/49G+/50G

(Library ID 1749) - Current Version 2.0

Table of Contents

1. Introduction	1
2. Using the Gfib20 Library	2
3. Precautions	7
4. The Algorithm	7
5. The Pseudocode	10
6. The Source Code	11
7. Future Updates	19
8. Contact Info	19

❶ INTRODUCTION

The Generalized Fibonacci Numbers Library is coded entirely in SystemRPL and calculates the values of Fibonacci Numbers, Lucas Numbers, and other Generalized Fibonacci-type recursive sequences. This library should be compatible with any HP49 series calculator (HP48GII, HP49, HP49G, HP49G+, HP50G) though I've only tested it on an HP50g, the Emu48 emulator, and the Debug4x emulator, the software used to write the library. Note that there are currently two versions of the library in the zipped download from my website so read the Installation instructions to determine which library should be installed on your particular device. Library creation was done on the HP50g with **CRLIB**, the built-in Development Library.

The latest public version of the Generalized Fibonacci Numbers Library can be downloaded from www.hpcalc.org at this URL:

<http://www.hpcalc.org/details.php?id=7150>

For the very latest revisions or for updates that have not yet made it onto HPCalc, the library may also be downloaded from my own website by using this link:

<http://members.bex.net/jtcullen515/math3.htm>

which is the '*Special Sums of Generalized Fibonacci Numbers*' page of my site. Just look for the download link in the second paragraph of that page.

This library is based loosely on the work of Gerald Hillier who released Fibo49 1.0 some years ago. He wrote an excellent SysRPL implementation of Fibonacci Number generation. You can find his library at HPCalc at the following URL:

<http://www.hpcalc.org/details.php?id=5101>

One of the routines Gerald wrote involves collecting the binary bits of a number and expressing them as a series of TRUE and FALSE statements in list form. This is such a handy little routine that I've included it inside this library with only a minor modification. With some optimizations, a new core algorithm, and a few rearrangements in the routines, I have been able to reduce the execution time in half for these types of calculations. In the process I was able to also extend the functionality of the algorithm used.

The Fibonacci Numbers are a recursive sequence where you start with two initial values, here zero and one, and then each term thereafter is equal to the sum of the two previous terms. This can be restated as, for the Fibonacci sequence:

$$F_0 = 0 \quad , \quad F_1 = 1 \quad , \quad F_N = F_{N-1} + F_{N-2}$$

and the sequence proceeds as **0, 1, 1, 2, 3, 5, 8, 13, 21, ...** and so on. The Lucas Sequence is similar in definition but it begins with a different pair of starting values:

$$L_0 = 2 \quad , \quad L_1 = 1 \quad , \quad L_N = L_{N-1} + L_{N-2}$$

In general, a simple recursive sequence such as the Lucas and Fibonacci sequences are defined by:

$$G_0 = A \quad , \quad G_1 = B \quad , \quad G_N = G_{N-1} + G_{N-2}$$

where **A** and **B** are also arbitrary integers, able to take on positive or negative values. In terms of initial starting values, the Fibonacci and Lucas sequences could be expressed as **G(N,0,1)** and **G(N,2,1)** such that their definitions could be restated as **F(N)=G(N,0,1)** and **L(N)=G(N,2,1)**.

These basic definitions and syntax help are included with the GFib library, on the second page of the soft menu, just hit the **NEXT** button and you'll see the **Vers** and **Help** tabs.

② Using The Gfib v2.0 Library

♦ FIBONACCI NUMBERS

All functions in the library assume that the device is being operated in RPN and that it is not in approximate mode. I make no guarantees regarding proper functioning in Algebraic Mode. Place an integer **N** on the stack and type the command (or use the soft-menu) **IFIB**. The full integer result is quickly returned. A UserRPL function with a less efficient algorithm returns all 209 digits of IFIB(1000) in about 5.4 seconds. Fibo49 is a much better implementation in SysRPL with a better algorithm and returns Fibo(1000) in about 0.88 seconds. The **IFIB** function in my GFib library returns the result in

just about 0.48 seconds. In general, I will get results with the GFib library in about half the time of the Fibo49 library. The first few values of the Fibonacci sequence are:

N	0	1	2	3	4	5	6	7	8	9	10
F(N)	0	1	1	2	3	5	8	13	21	34	55

For example, to calculate the 7'th term of F_N , enter the following:

7 IFIB enter

and your HP will respond with the value **13**. You may use the modular version of this function by adding the index **N** and the modulus **M** to the stack and calling **MFIB**. To find the value of the ninth Fibonacci number Mod 6, enter the following:

9 6 MFIB enter

and your HP will respond with the value **4**. Negative results, due to the internal MOD function, are corrected by the program before the result is displayed so you always get a positive answer.

♦ LUCAS NUMBERS

Again, place an integer **N** on the stack and enter the **ILUC** command to retrieve the integer value of the desired Lucas Number, **L(N)**. The initial terms of the Lucas sequence are 2 and 1. The first few values are:

L	0	1	2	3	4	5	6	7	8	9	10
L(N)	2	1	3	4	7	11	18	29	47	76	123

For example, to calculate the 6'th term of L_N , enter the following:

6 ILUC enter

and your HP will respond with the value **18**. You may also use the modular version of this function by adding the index N and the modulus M to the stack and calling **MLUC**. For example, to find the value of the tenth Lucas Number mod 19, enter the following:

10 19 MLUC enter

and the HP will respond with the value **9**. Negative results, due to the internal MOD function, are corrected by the program before the result is displayed so you always get a positive answer.

♦ Probablistic Primality Testing with Lucas Numbers

You can use the modular Lucas function as a nifty little probablistic prime test. For a prime number P , the value of $L(P) \bmod P = 1$ but there are a few pseudoprimes as well so use this along with another test to better confirm primality. As an example; find out if **5246789** is a probable prime or not by entering the following:

5246789 5246789 MLUC enter

and the HP will respond with **1** in about 0.62 seconds. The result of **1** simply means that the number **5246789** is probably prime; this is not a deterministic test. If the result had been other than **1**, then the test will have determined that the number **5246789** is definitely *NOT* a prime. However the Lucas test (and this is not the Lucas-Lehmer) is very handy if used in conjunction with a simple Fermat Test since Carmichael Numbers often confound the Lucas Test. Take for instance the number **739444021** which is

factored as **277 * 1381 * 1933** but is tough to detect as composite. Enter the following into the HP:

739444021 739444021 MLUC enter

and the HP spits out **433484853** in about 0.89 seconds. Since the result is *NOT* equal to one, we have determined that the number **433484853** is definitely *NOT* a prime, something that a simple Fermat Test has a hard time doing with this particular number and others like it.

There are pseudoprimes that fool both a simple Fermat Test to a single base as well as this simple Lucas Test. Taking a look at the single base pseudoprimes; numbers **N** that are composite yet meet the criteria for probable primality to a single base. For example, base-2. Then for a composite **N**, we would have $2^{N-1} \bmod N = 1$. What's the likelihood that such a number would also be a Lucas pseudo-prime? Quite a good chance. In fact, there are 115 examples below 140,000,000 and the first few are: **2465, 219781, 228241, 252601, 399001, 512461, ...** and the last few are: **..., 130497361, 132239521, 133157701, 133800661, 139592101, and 139952671**. Strong pseudoprimes, composite **N** that pass the Rabin-Miller test to a single base (again, 2 for example) are even more rare. There are 26 such numbers within the same range that are base-2 strong pseudoprimes as well as Simple Lucas pseudoprimes. The first few are: **252601, 741751, 1909001, 2757241, ...** and the last few are: **..., 133157701, 133800661, 139592101, and 139952671**.

As the index **N** gets very large, so does the modulus since we take the P'th Lucas Number Mod P. An example would be the 141'st Cullen Number (obviously my favorite numbers and this particular Cullen Number is a prime). The formula for the number is:

$$C_{141} = 141 \cdot 2^{141} + 1$$

with the full 45-digit decimal expansion of:

$$C_{141} = 393050634124102232869567034555427371542904833$$

How long do you suppose the Lucas Test takes to chew up this one? I have the Cullen function programmed on my HP as **CULL(N)** and so I ran the test with the following:

<< 141 CULL DUP MLUC >>

and the HP spit a single **1** back at me in 18 seconds flat. That's not bad at all. It took 36.9 seconds for the HP's **ISPRIME?** function to declare it prime - though the internal test is a lot closer to being 'deterministic' than this simple Lucas Test. There are plenty of applications for the Lucas and other Fibonacci type sequences; primality testing is just one of them. Have fun!

♦ GENERALIZED FIBONACCI NUMBERS

This function requires three integers to be placed on the stack. First, place the index **N** followed by the initial terms **A** and **B**. Type **IGIB** (or use the soft-menu) to calculate the value of **G(N,A,B)**, also written as **G_{A,B}(N)**. Generalized Fibonacci numbers work the same way as the regular Fibonacci numbers; take two initial terms and each term after that is the sum of the two previous terms. As an example, consider the Generalized Fibonacci sequence with **A = 2 & B = 4**, the values of **G₀** & **G₁**. The first few values of this particular sequence are:

N	0	1	2	3	4	5	6	7	8	9	10
G(N,2,4)	2	4	6	10	16	26	42	68	110	178	288

For example: to calculate the 9'th term of **G_{2,4}**, enter the following:

9 2 4 IGIB enter

and your HP will respond with the value **178**. The Generalized sequences have a modular version as well. To find the value of the 3,000'th Generalized Fibonacci with the initial terms of **A = G₀ = 15** and **B = G₁ = -9** taken Mod **3491**, enter the following:

3000 15 -9 3491 MGIB enter

and the HP responds with the value of **1544** and this was done in just a hair under 1/3 of a second. Modular calculations are much quicker since the ZINT calculations in the core routine are working on

much smaller numbers and so take little time to finish the calculations per bit of N.

*NOTE THAT THE FIBONACCI, LUCAS, AND GENERALIZED FIBONACCI SEQUENCES MAY HAVE THEIR INDICES EXTENDED TO THE NEGATIVE INTEGERS AND THE LIBRARY **-DOES-** ACCEPT NEGATIVE INDICES. FOR THE GENERALIZED FIBONACCI SEQUENCES, NEGATIVE INTEGERS FOR THE INITIAL TERMS OF THE SEQUENCE ARE ACCEPTED AS WELL. - JTC*

③ Precautions

As with the installation of any SystemRPL library, don't have anything on your calculator that you don't mind losing. If you do, take the proper precautions; backup your data and then backup your backups! You have been warned! Transfer the file to your calculator, COPY it to the proper port, and attach it with the usual **ON+F3** key combo.

Although I've taken care not to change settings in the calculator and integer arguments for these functions are checked before the functions are executed, there are no checks to see if the index may be too large of an integer, possibly taking long periods of time to calculate a result or even overflowing the calculators memory. As a general rule, the **IFIB** function begins to take a long time to finish after about **N=10,000** since the full integer representation is returned and begins to increase up to the several thousand digit range.

Testing has not been thorough for all HP49 series calculators. The library uses ZINTS so your calculator must be able to recognize and work with that data type with the given entry points in the source code. I can say that the library works on the HP50g, HP49G+, the Debug4x emulator, and the Emu48 PC emulator.

Program output is only as good as the input provided. I have taken care to allow only the proper data types in the proper order; ZINT data types or converted values, with the proper signs, checks, and so on. I cannot test every possible input scenario but I'm confident that a slip of the thumb should not bring on a disaster. If you have any trouble though, please let me know the details of the 'incident'.

④ The Algorithm

The algorithm is a left to right form of binary exponentiation where doubling and single-steps of the index is performed. The algorithm is based on the following relations:

$$F_{2N-1} = F_N^2 + F_{N-1}^2 \quad \text{Relation (1)}$$

$$F_{2N} = 2 F_{N-1} F_N + F_N^2 \quad \text{Relation (2)}$$

$$F_{2N+1} = F_N^2 + F_{N+1}^2 \quad \text{Relation (3)}$$

and, if performed correctly, results in a calculation taking only half the number of multiplications than other algorithms; the rest of the calculation being a handful of additions per bit of N.

The procedure is to save a pair of Fibonacci Numbers, called f0 and f1, which are initialized at F0=0 and f1=1 at the beginning of the algorithm. Temporary holders for the squares of these values are called U and V and are used since the majority of the calculation concerns f0^2 and f1^2, though we also need the original values of f0 and f1 to remain on the stack.

We scan the bits of the index N from MSB to LSB but disregard the highest bit since it is already known to be a binary '1'. Depending on the value of this bit, we will calculate the value of F[2N] for a bit value=0 and F[2N+1] for a bit value=1. This is the basic binary left-to-right exponentiation.

The calculation in part depends on whether or not the index N is an even or an odd number, since the double angle formulae for the fibonacci numbers differ for even vs odd indices. In particular, I refer to the properties of the Lucas Numbers in relation to the calculation of the double-angle formulae for the Fibonacci Numbers:

$$F_{2N+1} = 5 F_N^2 + 2 (-1)^N \quad \text{Relation (4)}$$

where the value of the second term is held in a temp variable called C. This seems like a difficult step but there are only two possible values for C, and using this relation cuts the number of required multiplications in half... and the time taken to do the ZINT multiplications in SysRPL represents the bulk of the time required for the algorithm to finish.

The initial value of C is -2 since we begin with f0=0 and f1=1 which is the position of 'one' in the Fibonacci sequence. This is odd so we use a value of -2 for our initial C.

For any index N, we can find the value of F[2N] by:

$$F_{2N} = F_N^2 + F_{N-1}^2 \quad \text{Relation (5)}$$

Relation {4} and Relation {5} give us the values of F[2N+1] and F[2N]. What we need though is F[2N-1] since the algorithm works on the principle of updating F[N] and F[N-1] through the values of f0 and f1 respectively. This is gotten by taking the relation:

$$F_{2N-1} = F_{2N+1} - F_{2N} \quad \text{Relation (6)}$$

and combining it with the first three relations to derive expressions for the required Fibonacci Numbers.

What you end up with is the following values, expressed in terms of $U=f0^2$ and $V=f1^2$:

ODD BIT	EVEN BIT
F0 = 3V – 2U + C	F0 = U + V
F1 = 4 V - U + C	F1 = 3V – 2U + C

After the loop for that bit is completed, the value of C is updated. For an odd bit where the index is just doubled, we now have an even index and so C takes the value of +2. For an index doubled and incremented by one, $(-1)^n$ is a negative value and so C must be a -2. The value of C is applied during the subsequent loop.

When the algorithm completes the loops for all bits in the index N, the value of f0 holds $F[N-1]$ and f1 holds the value of $F[N]$. For calculating Fibonacci numbers, we just take the value of f1 which is $F[N]$ directly.

For Lucas Numbers, we use a simple relation between the generalized Fibonacci sequences and the corresponding Fibonacci Numbers. The Lucas sequence, in general terms, is the recursive sequence with $A=2$ and $B=1$, where A is the value of $G[0]$ and B is the value of $G[1]$. The Lucas numbers are given by:

$$L_N = 2 F_{N-1} + F_N \quad \text{Relation (7)}$$

for the same reason that the Generalized Fibonacci numbers are given by:

$$G_N = A \cdot F_{N-1} + B \cdot F_N \quad \text{Relation (8)}$$

For negative arguments, the following formula is useful:

$$G_{-N} = (-1)^N [A \cdot F_{N+1} - B \cdot F_N] \quad \text{Relation (9)}$$

but is only really useful for describing the properties of the Generalized Fibonacci numbers with a negative index. I have taken shortcuts to actually implement it. See the below source code for more detail; it should be self-explanatory.

5 The Pseudocode

The entire library depends on the core routine which has just one job; given an index N , return $F(N)$ and $F(N-1)$. In the Generalized Fibonacci Numbers Library, this core has been split into two parts; collecting the binary bits, as per Gerald Hillier's code, and then calculating the binary exponentiation for each bit collected. For those of you who are interested in implementing this in another programming language, here is the pseudo-code for the algorithm:

```
Define Function Fib(N) -----  
  
Local Variables: u,v,f0,f1,i,tt,c,s  
s= {} which is an empty list  
  
While n>0                'This collects binary bits of N  
  If (N mod 2)=1 Then      'and assembles them in reverse  
    s = {1} + s           'order in list s  
  Else  
    s = {0} + s  
  End  
  N = floor(N/2)  
Wend  
  
tt = dimension(s)         'Initialize core routine  
f0 = 0 : f1 = 1 : c = -2  
  
For i=2 to tt             'Calc loop for each bit  
  u = f0^2 : v = f1^2  
  If s[i]=1 Then  
    f0 = 3v-2u+c : f1 = f0+v+u : c = -2  
  Else  
    f0 = u+v : f1 = 3v-2u+c : c = 2  
  End  
Next i  
Return f1                 'F(N)=f1 & F(N-1)=f0  
  
End Define Function -----
```

If you wanted the above function to return the Lucas Numbers, then just replace '**Return f1**' with the generalized form for the Lucas Sequence; '**Return 2*f0+f1**' per Relation {7} given in the 'ALGORITHM' section above. In a like manner, to return the value of a generalized sequence with the initial starting values of A and B, change the definition of the function to '**Define Function Fib(N,A,B)**' and change the '**Return f1**' to '**Return A*f0+b*f1**' per Relation {8} also given in the 'ALGORITHM' section above..

⑥ The Source Code

The source code of the Generalized Fibonacci Library is being freely provided in case anyone out there has any ideas for making this library perform even better. There are three visible files; **IFIB**, **ILUC**, and **IGIB**. There are also two hidden utility functions; bits and bobs .. yes, I actually named them that way! The 'bits' routine, a faster version of Hillier's code, fetches the binary bits of N, and the 'bobs' routine calculates the Fibonacci pair used by the rest of the library. As far as execution time of the below routines, during the calculation of a problem, it of course varies on the size of N. Proportionally though, it is consistent unless the N provided to the program is extremely large.

As an example, during the calculation of IFIB(5000) in Debug4x, about 0.01sec is spent calculating the preliminaries. The 'bits' routine finishes its job in about 0.04 sec. The majority of the execution time is spent in the 'bobs' loop, which takes 4.40 sec, and just under 0.01 sec to finish up and be ready for display. I show, using **TEVAL**, that the entire calculation takes 4.402 sec in all. So it's almost impossible to find time that's NOT spent in the 'bobs' loop, at least by inspecting the overall timing figure.

IFIB

RPL

INCLUDE GFib.h

ASSEMBLE

CON(1)8

RPL

xNAME IFIB :: (Calculate the N'th Fibonacci Number)

CK1 ::

FLASHPTR CK1Z :: (or make ZINT if dec=0 else ERR)

Z-2_FALSE FALSE (Lam1= N<0 ?)

3NULLLAM{}_BIND (Lam2= N even? : Lam3= C term)

FLASHPTR DupZIsNeg? (Note if N<0 in Lam1)

IT :: TRUE 1PUTLAM ;

FLASHPTR DupZIsEven? (Note if N is even in Lam2)
 IT :: TRUE 2PUTLAM ;

FLASHPTR ZABS (N=|N| positive indices only)

DUP Z2_ Z< ?SEMI (if N<2 then F[N]=N so done)

xbits (L2R Binary Decoding Routine)
 xbobs (Core L2R F[2n]/F[2n-1] Routine)

SWAP DROP (Fetch F[N])

1GETLAM IT ::
 2GETLAM IT ::
 xNEG ; ;

ABND
 ; (End ^CK1Z)
 ; (End CK1)
 ; (End IFIB)

ILUC

RPL

INCLUDE GFib.h

ASSEMBLE

CON(1)8

RPL

xNAME ILUC :: (Calculate the N'th Lucas Number)

CK1 ::
 FLASHPTR CK1Z :: (or make ZINT if dec=0 else ERR)

Z-2_FALSE FALSE (Lam1= N<0 ?)
 3NULLLAM{}_ BIND (Lam2= N even? : Lam3= C term)

FLASHPTR DupZIsNeg? (Note if N<0 in Lam1)
 IT :: TRUE 1PUTLAM ;
 FLASHPTR DupZIsEven? (Note if N is even in Lam2)
 IT :: TRUE 2PUTLAM ;

FLASHPTR ZABS (N=|N| positive indices only)

DUP Z1_ Z> ITE :: (Special case if N=0 or N=1)

xbits (L2R Binary Decoding Routine)
 xbobs (Core L2R F[2n]/F[2n-1] Routine)

SWAP DUP (Fetch L[N]=F[N]+2*F[N-1])

```

FLASHPTR QAdd
FLASHPTR QAdd ;
::
  DUP Z0_ Z> ?SKIP      ( if N=1 then L[N]=1 )
  :: DROP Z2_ ;          ( if N=0 then L[N]=2 )
; ( End ITE )

```

```

1GETLAM IT ::
2GETLAM ?SKIP ::
xNEG ; ;

```

```

ABND
; ( End ^CK1Z )
; ( CK1 )
; ( End ILUC )

```

IGIB

```
RPL
```

```
INCLUDE GFib.h
```

```

ASSEMBLE
CON(1)8
RPL
xNAME IGIB ::      ( Calculate the N'th Generalized Fib )

CK3 ::
FLASHPTR CK3Z ::   ( or make ZINT if dec=0 else ERR )
ROT FLASHPTR ZABS   ( N=|N| )

DUP Z1_ Z> ITE ::
  xbits              ( L2R Binary Decoding Routine )
  xbobs ;            ( Core L2R F[2n]/F[2n-1] Routine )
::
Z1_ Z0_
ROT Z0_ Z= ?SKIP SWAP
; ( End ITE )

ROT FLASHPTR QMul   ( Fetch G[N]=B*F[N]+A*2*F[N-1] )
UNROT FLASHPTR QMul
FLASHPTR QAdd

; ( End ^CK3Z )
; ( End CK3 )
; ( End IGIB )

```

MFIB

```
RPL
```

```
INCLUDE GFib.h

ASSEMBLE
CON(1)8
RPL
xNAME MFIB ::          ( Calculate F[N] mod M )

CK2 ::
FLASHPTR CK2Z ::      ( or make ZINT if dec=0 else ERR )

Z0_Z-2_FALSE FALSE    ( Lam1= N<0 ? : Lam2= N even? )
4NULLLAM{} BIND       ( Lam3= C term : Lam4= modulus )

FLASHPTR ZABS          ( M=|M| positive modulus only )
4PUTLAM

FLASHPTR DupZIsNeg?    ( Note if N<0 in Lam1 )
IT :: TRUE 1PUTLAM
FLASHPTR ZABS          ( N=|N| positive indices only )
FLASHPTR DupZIsEven?   ( Note if N is even in Lam2, if N<0 )
IT :: TRUE 2PUTLAM ;

DUP Z1_Z> IT ::        ( if N<2 then F[N]=N so done )

xbits                  ( L2R Binary Decoding Routine )
xmbobs                 ( Core L2R F[N-1]/F[N] Routine )

SWAP DROP              ( Fetch F[N] )
;

1GETLAM IT ::          ( Correct the sign of the result )
2GETLAM IT ::
xNEG ;

4GETLAM xMOD
DUP Z0_Z< IT ::        ( Put mod calculation positive )
4GETLAM FLASHPTR QAdd ;

; ( End ^CK2Z )
; ( End CK2 )
; ( End MFIB )
```

MLUC

RPL

INCLUDE GFib.h

ASSEMBLE
CON(1)8

```

RPL
xNAME MLUC ::          ( Calculate L[N] mod M )

CK2 ::
FLASHPTR CK2Z ::      ( or make ZINT if dec=0 else ERR )

Z0_ Z-2_ FALSE FALSE  ( Lam1= N<0 ? : Lam2= N even? )
4NULLLAM} BIND        ( Lam3= C term : Lam4= modulus )

FLASHPTR ZABS          ( M=|M| positive modulus only )
4PUTLAM

FLASHPTR DupZIsNeg?    ( Note if N<0 in Lam1 )
IT :: TRUE 1PUTLAM
FLASHPTR ZABS          ( N=|N| positive indices only )
FLASHPTR DupZIsEven?   ( Note if N is even in Lam2, if N<0 )
IT :: TRUE 2PUTLAM ; ;

DUP Z1_ Z> ITE ::     ( Special case if N=0 or N=1 )

  xbits                ( L2R Binary Decoding Routine )
  xmbobs                ( Core L2R F[N-1]/F[N] Routine )

SWAP DUP               ( Fetch L[N]=F[N]+2*F[N-1] )
FLASHPTR QAdd
4GETLAM FLASHPTR QAddMod ;
::
DUP Z0_ Z> ?SKIP      ( if N=1 then L[N]=1 )
  :: DROP Z2_ ;        ( if N=0 then L[N]=2 )
; ( End ITE )

1GETLAM IT ::          ( Correct the sign of the result )
2GETLAM ?SKIP ::
xNEG ; ;

4GETLAM xMOD
DUP Z0_ Z< IT ::      ( Put mod calculation positive )
4GETLAM FLASHPTR QAdd ;

ABND
; ( End ^CK2Z )
; ( CK2 )
; ( End MLUC )

```

M G I B

RPL

INCLUDE GFib.h

ASSEMBLE

CON(1)8
RPL
xNAME MGIB :: (Calculate the N'th Generalized Fib Number)

CK4 ::
FLASHPTR CK3Z :: (or make ZINT if dec=0 else ERR)
4 PICK
FLASHPTR CK1Z :: (or make ZINT if dec=0 else ERR)
DROP

Z0_ Z-2_ FALSE FALSE (Lam1= N<0 ? : Lam2= N even?)
4NULLLAM} BIND (Lam3= C term : Lam4= modulus)

FLASHPTR ZABS (M=|M| positive modulus only)
4PUTLAM

ROT (Bring down the index N)
FLASHPTR DupZIsNeg? (Note if N<0 in Lam1)
IT :: TRUE 1PUTLAM
FLASHPTR ZABS (N=|N| positive indices only)
Z1_ FLASHPTR QAdd (For N<0, must inc N by one)
FLASHPTR DupZIsEven? (Note if N is even in Lam2, if N<0)
IT :: TRUE 2PUTLAM ; ;

DUP Z1_ Z> ITE :: (Special case if N=0 or N=1)

xbits (L2R Binary Decoding Routine)
xmbobs ; (Core L2R F[N-1]/F[N] Routine)

::
ROT Z1_ Z0_ (if N=1 then f1=1 : f0=0)
ROT Z0_ Z= ?SKIP SWAP (if N=0 then f1=0 : f0=1)
; (End ITE)

1GETLAM IT :: (Correct the sign of the result)
2GETLAM ITE :: (and fix the ordering of the pair)
xNEG SWAP ;
:: SWAP xNEG ; ;

ROT 4GETLAM (Fetch $G[N]=B \cdot F[N]+A \cdot 2 \cdot F[N-1]$)
FLASHPTR QMulMod
UNROT 4GETLAM FLASHPTR QMulMod
4GETLAM FLASHPTR QAddMod

DUP Z0_ Z< IT :: (Put mod calculation positive)
4GETLAM FLASHPTR QAdd ;

ABND
; (End CK1Z)
; (End CK3Z)
; (End CK4)
; (End MGIB)

m b o b s
RPL
INCLUDE GFib.h
ASSEMBLE
CON(1)8
RPL
xNAME mbobs :: (Core L2R F[2n]/F[2n-1] Modular Routine)
 (In: List, L2R Bit Pattern of N)
 (Out: F[N-1],F[N] // on stack)

Z0_ Z1_ xPICK3 (LST,Z0,Z1,LST)
LENCOMP #1+ (LST,Z0,Z1,LEN+1)

#1 DO (DO from 1 to LEN+1 , LST,Z0,Z1)

DUP 4GETLAM FLASHPTR QMulMod
SWAP DUP 4GETLAM FLASHPTR QMulMod
2DUP
FLASHPTR QSub
DUP FLASHPTR QAdd
3PICK FLASHPTR QAdd
UNROT 4GETLAM FLASHPTR QAddMod
SWAP 3GETLAM 4GETLAM FLASHPTR QAddMod
3PICK INDEX@ NTHCOMPDROP
ITE ::
DUP UNROT 4GETLAM FLASHPTR QAddMod
Z-2_ 3PUTLAM
;
::
Z2_ 3PUTLAM
;
LOOP
ROT DROP
; (End mbobs)
b i t s
RPL
INCLUDE GFib.h

```
ASSEMBLE
CON(1)8
RPL
xNAME bits :: ( L2R Binary Decoding Routine      )
               ( In: ZINT representing N          )
               ( Out: List, L2R Bits less high bit )

ZERO SWAP
BEGIN
  Z2_ FLASHPTR BESTDIV2
  OVER Z0_ Z<
  WHILE ::
    Z1_ Z= SWAPROT #1+ SWAP ;
  REPEAT
  2DROP reversym {}N

; ( End bits )
```

b o b s

```
RPL

INCLUDE GFib.h

ASSEMBLE
CON(1)8
RPL
xNAME bobs :: ( Core L2R F[2n]/F[2n-1] Integer Routine )
               ( In: List, L2R Bit Pattern of N )
               ( Out: F[N-1],F[N] // on stack )

Z0_ Z1_ 3PICK ( LST,Z0,Z1,LST )
LENCOMP #1+ ( LST,Z0,Z1,LEN+1 )

#1 DO      ( DO from 1 to LEN+1 , LST,Z0,Z1 )

FLASHPTR ZSQ_
SWAP FLASHPTR ZSQ_
2DUP
FLASHPTR QSub
DUP FLASHPTR QAdd
3PICK FLASHPTR QAdd
UNROT FLASHPTR QAdd
SWAP 3GETLAM FLASHPTR QAdd

3PICK INDEX@ NTHCOMPDROP

ITE ::
DUP UNROT FLASHPTR QAdd
Z-2_ 3PUTLAM
;
```

```
::  
Z2_3PUTLAM  
;  
  
LOOP  
  
ROT DROP  
  
; ( End bobs )
```

7 Future Updates

There may be some algorithmic tweaking to gain some additional performance improvement though, without a new algorithm or an assembly implementation, there is not much room for improvement as far as speed goes. About 99.5% of the time during the execution of the algorithm is spent within the inner loop where there are ZINT calculations; two squarings and a half-dozen additions per bit of N. There is a thought though of including one other useful sequence in this library; the Perrin sequence.

8 Contact Information

Jim Cullen
Email: jtcullen515 'at' bex.net
Website: <http://members.bex.net/jtcullen515>
August 22, 2009

*Generalized Fibonacci Numbers Library v2.0 Help File procuced with the help of OpenOffice.org Writer 3.1.1 for word processing and PDF export
For more information on OpenOffice freeware, please visit <http://www.openoffice.org>*