X-GSB
An Extended Subroutine Handler For The HP34C

How to get a classic calculator to do complicated things

This challenge presented itself to me when I recently began tutoring Math and Physics and picked up my old HP34C programmable calculator. I bought it in 1979 and I fell in love again. I re-read the manual fully and thoroughly, had a look at some old programs, and started fiddling with some new programming. My enthousiasm led me to wanting to program a recursive subroutine. But that usually requires many subroutine levels. A lot more than the GSB and RTN instructions of the HP34C can handle: only 6 levels.

I decided to accept this as a programming challenge to me. Of course, with todays superior processing power readily available in hand held calculators, software programs, apps and websites with calculators - my venture has no real world use. But I found it very satisfying to work on such an old machine, requiring ingenuity to fit a program into the small program memory and to have only a few registers.

And I did indeed solve the puzzle. In the end I could run a recursive Fibonacci calculation. On the way I had to develop a big stack for return addresses and squeeze it into only a few registers. And later on I made a data stack specifically for traversing the Fibonacci tree.

There are emulators for the HP34C that provide the same instruction set and architecture (e.g. RPN-34 CE from Cuvee Software) with more memory and more registers. Then another, easier attack vector could have been chosen. The same goes for the HP15CE calculator: more code space, more registers. Nevertheless I opted for making it work on the original programmable HP34C calculator. The program I made can provide well over 100 levels deep of subroutine nesting. With none or a few minor adjustments this X-GSB Handler will work on the other machines as well and will provide even more levels of nesting.

So here it is, the story of my journey, from scratch to a real recursive program running on the original HP34C. As I said, it's of no real use in today's world, but maybe, just maybe, someone out there, finds it fun to read. I know I had lots of fun making it!

Harry de Groot

Hart Nibbrigkade 17
2597 XN Den Haag
Netherlands
harry@hdg.nu
+31 622421128

- -

If you want to buy, repair or replace an HP calculator, you can find info below. There are options to change the old processor for a faster chip that emulates the HP34C code. Also, there are very fast software emulations for iPhone and laptop. See also page 17 for performance differences on the Fibonacci program.

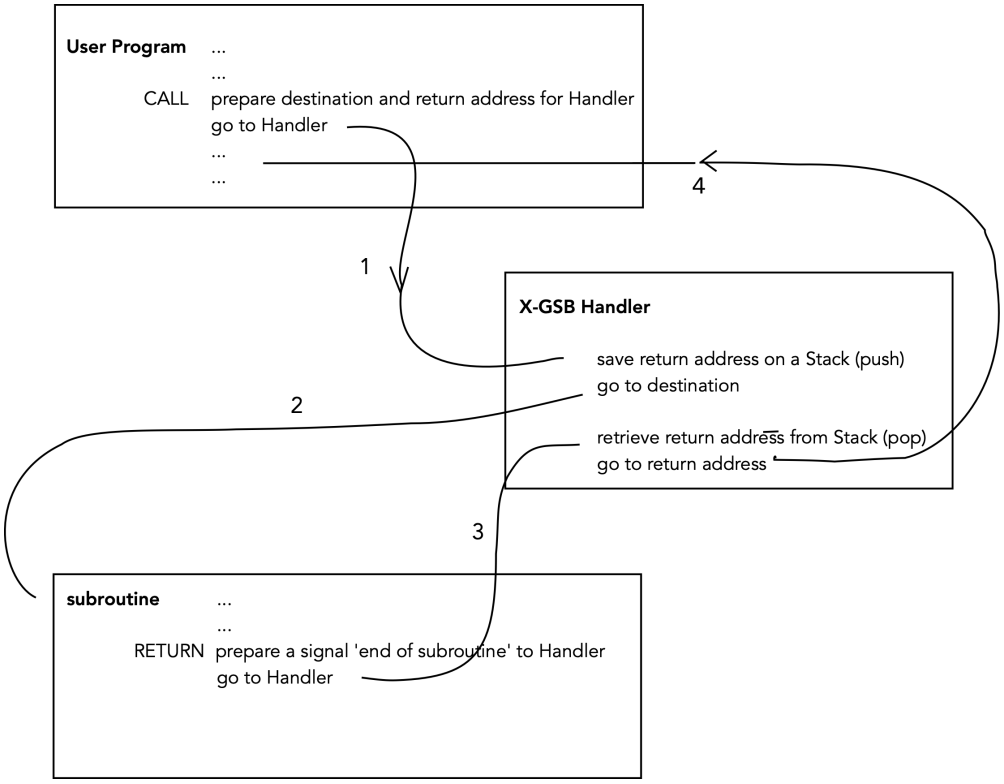
www.thecalculatorstore.com
www.cuveesoft.ch/
www.panamatik.de/html/hp_low_power.html

Limitations of the HP34C

The HP34C can hold (only) up to six subroutine levels using GSB and RTN. For programs that use recursive routines (a routine that calls itself multiple times) this is almost always too little. To allow for more levels a new method is designed for subroutine calling and returning from a subroutine. This method is implemented as a HP34C program called X-GSB Handler and takes care of the calls and returns, bypassing GSB and RTN. The Handler has to be included as part of your user program. Depending upon the size of the user program and the number of registers it uses, the Handler can provide well over 100 levels deep of nesting.

Method of calling and returning

When a program wants to go to a subroutine, it passes the requested subroutine address and it's return address to the Handler. The Handler pushes the return address onto a Stack. For returns it pops the return address from the Stack.This Stack must not be confused with the common XYZT stack of the HP34C; we therefore write Stack with a capital S.



The following matters must be given attention:

- The destination address (the address of the subroutine) in the HP34C is usually indicated by a label (GTO label or GSB label). The return address is implied to be the next instruction after GSB. The HP34 works out the return address internally; the address is not available to the programmer as an absolute address or label. If we do not use the regular GSB command then we must devise another way of determining the return address.

- If each return address is stored in a regular register, the Stack will soon be full. There are only 20 regular registers. And these are shared with the program space. Therefore we must pack more return addresses into a register.

- HP34C can use labels or line numbers for addresses. There are only 12 labels. 0..9, A and B. A linenumber can only be used with the GTO **I** command. If the number in **I** is negative, it will be interpreted as a positive line number.

- The handler is called by the user program and must be able to determine if it is called to go to a subroutine or if is is called to return from a subroutine

Design of the Stack for storing return addresses

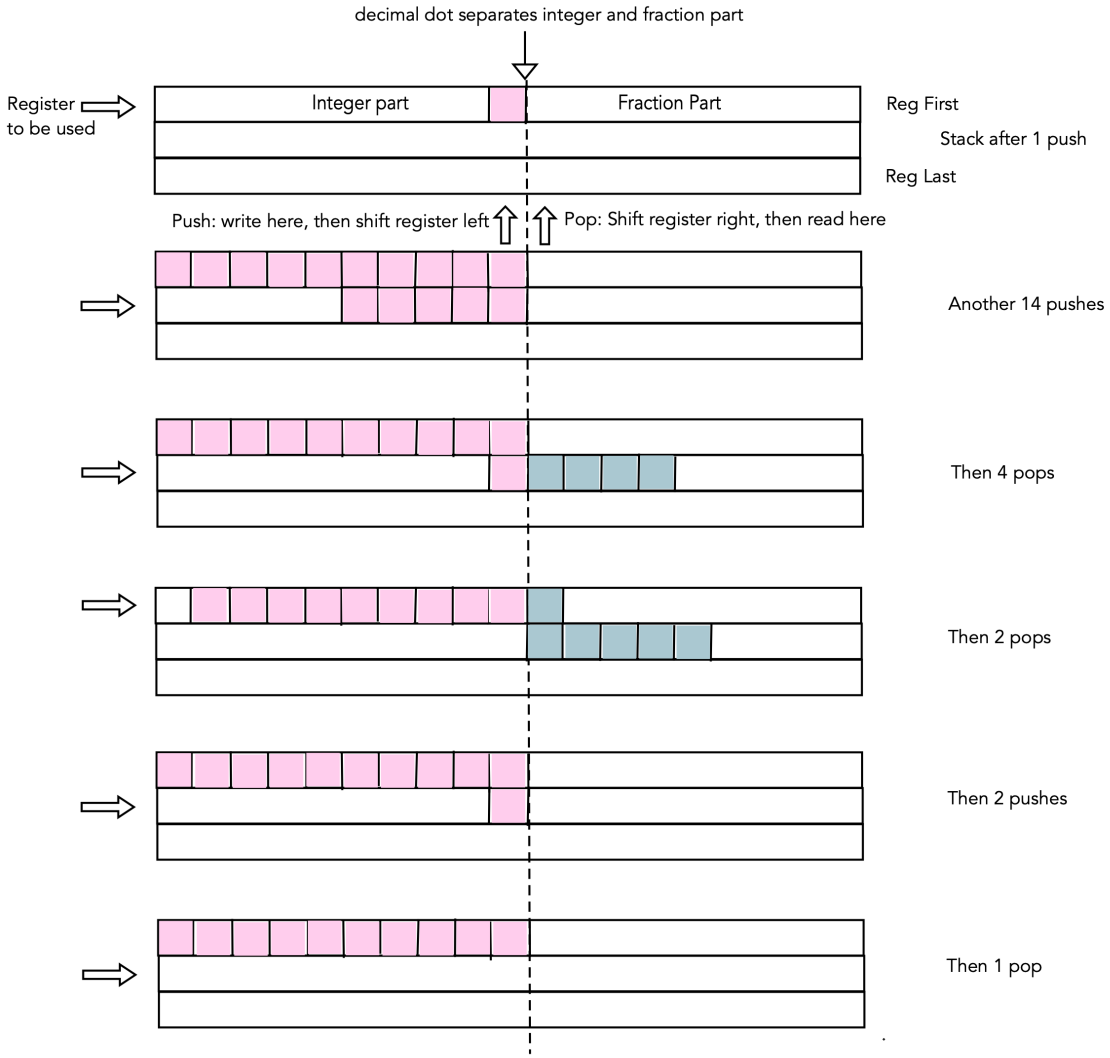
If GSB cannot be used, then we have to find another way of noting the destination and return address for the Handler. We can choose a label for the destination address and also a label for the return address. This would require two labels for each call of a subroutine. We would run out of labels quickly, as there are only 12 labels. Therefore we will instead use the line number for the address of the subroutine. For the return address we could also use a line number, but in order to be able to store many return addresses and for convenience (labels are easier to use) we can use a label as return address. This requires less storage in the Stack. One register can hold 10 one digit label numbers. We will not use label A and B for return addresses because they are two digits (10 and 11).

When a call is made, a register of the Stack is filled (push) with the return address (a label number) by first shifting all digits of the register one place to the left (bij multiplying by 10) to make room for the next labelnumber to be added. This is done by adding (the + operation) the one digit number. to the register. On initialization of the Stack the leftmost digit of the fraction must be zero).

When a return is made, the last previously stored label is retrieved (popped) by shifting the digits one place to the right (divide by 10), so that the labelnumber is now the leftmost digit of the fraction part of the register and can be converted into an integer bij multiplying by 10.

When a register is filed with 10 labels and another push is needed, then the next register must be used. When a pop is needed and the register is already emptied, the previous register must be used.

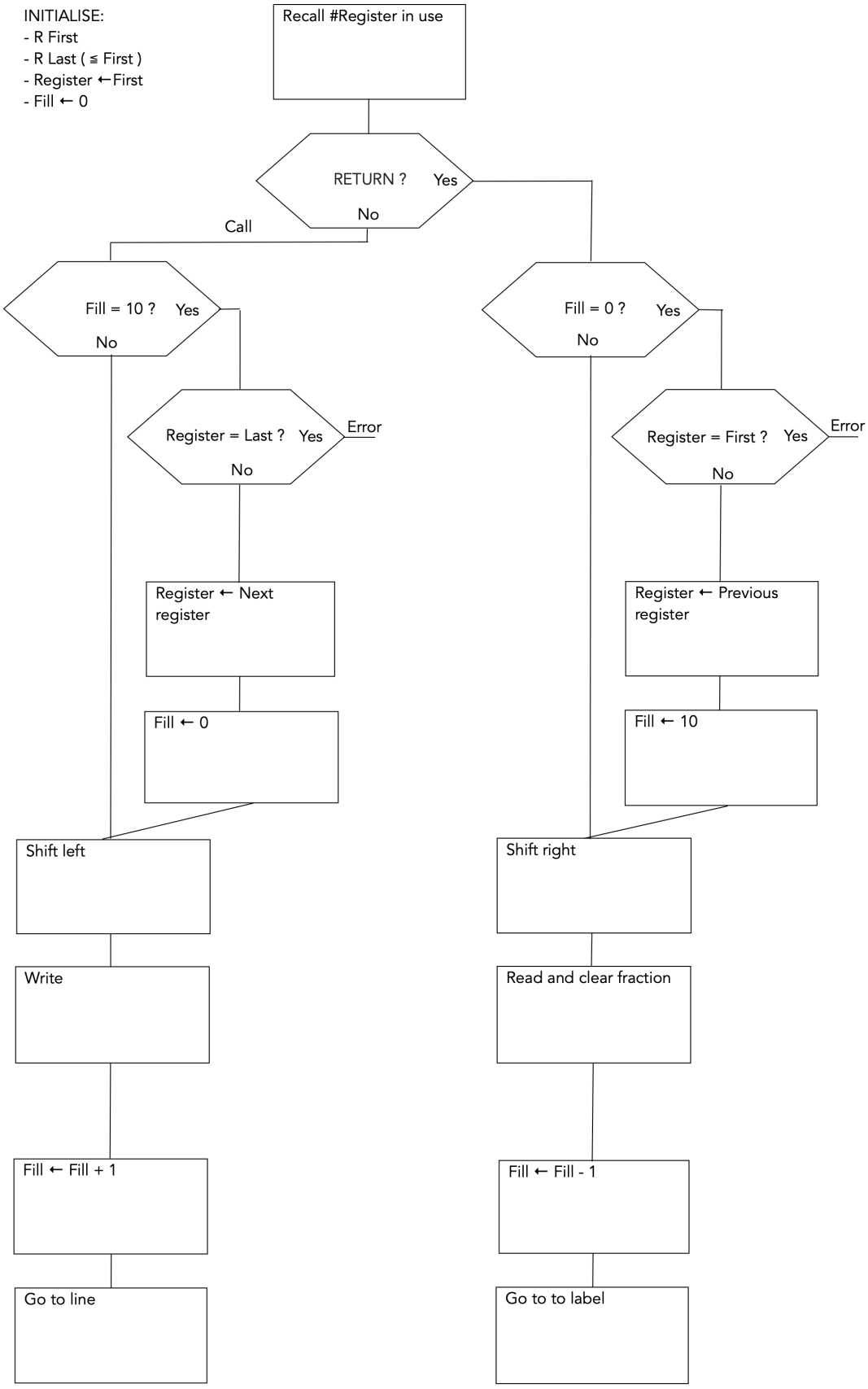
A Stack consisting of three registers



Note that the fraction part (grey colored) might not be 0, but a next push (shift left) requires it to be zero. Therefore the fraction must be cleared after each pop. The last two diagrams show that, if this is implemented, the fraction will always be zero.

In the last diagram, the second register is emptied and the pointer on the left still points to this empty register. So, if the next operation is a pop again, then the previous register must be selected.

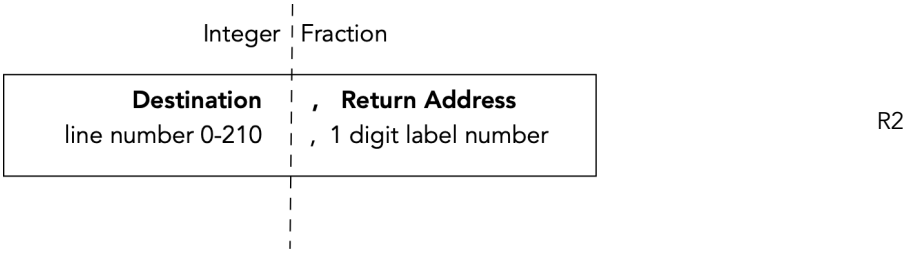
Flow chart of X-GSB Handler



Passing addresses from user program to X-GSB Handler

The user program passes the line number of the subroutine (that it wants to go to) to the Handler. And for the return address it passes a label. It is not difficult for a user to find the line number of the subroutine. However if the program is changed by deleting or inserting commands the user must be aware that the line number of the subroutine may have changed.

To use as few registers as possible, the destination and return address are packed into one register as the integer (line number) and fraction (label number) part. Labels are a one digit number. To keep things simple the Handler assumes a one digit label number and thus the Handler will work for labels 0..9 only and not for A and B.



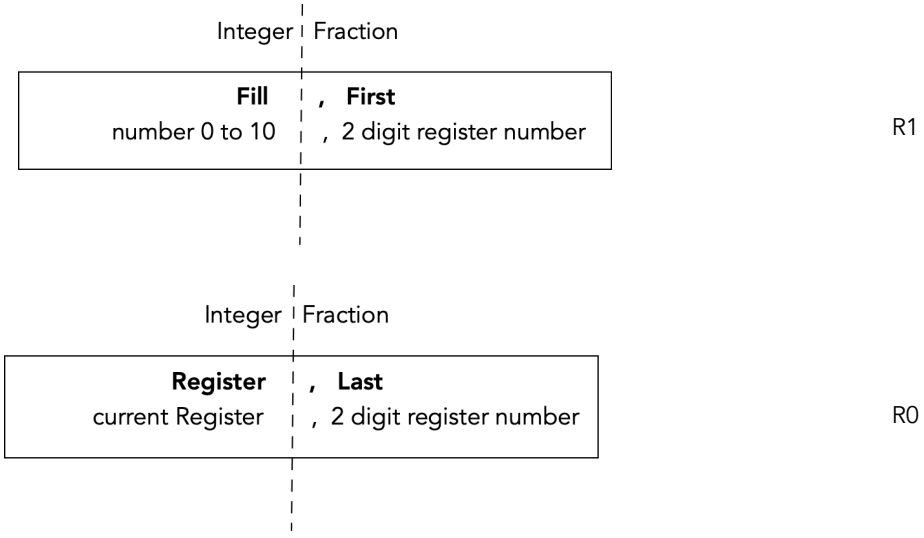
The Handler can extract the line number (1 … 210) by the INT command. And the label number can be extracted by the FRAC command and then multiply by 10 to get an integer from 0..9. In order to signal to the Handler that a return from subroutine must be made the subroutine puts 0 as a destination address. In this case the return address (the fraction part) is not relevant.

Note: The line number is stored in the integer part, otherwise the lines 1 to 210 would have to be coded in the fraction part as always 3 digits e.g. 095 instead of 95. This requires more program lines for coding and decoding and more excecution time.

Registers for control of the Stack

For the Stack we can use as many consecutive free registers as we want. The total Stack size is the number of assigned registers multiplied by 10. The registers are named from First to Last. A counter named Fill is used to check if the current Register is empty (0) or full (10)

Fill and First are stored together as the INT part and FRAC part respectively. First is a two digit number so e.g. 7 should be input as 07. Variables Register and Last are also stored together in one register.



Using CALL and RETURN in your program, a basic approach

Each CALL requires a label for the return address and needs to specify the line number of the subroutine.

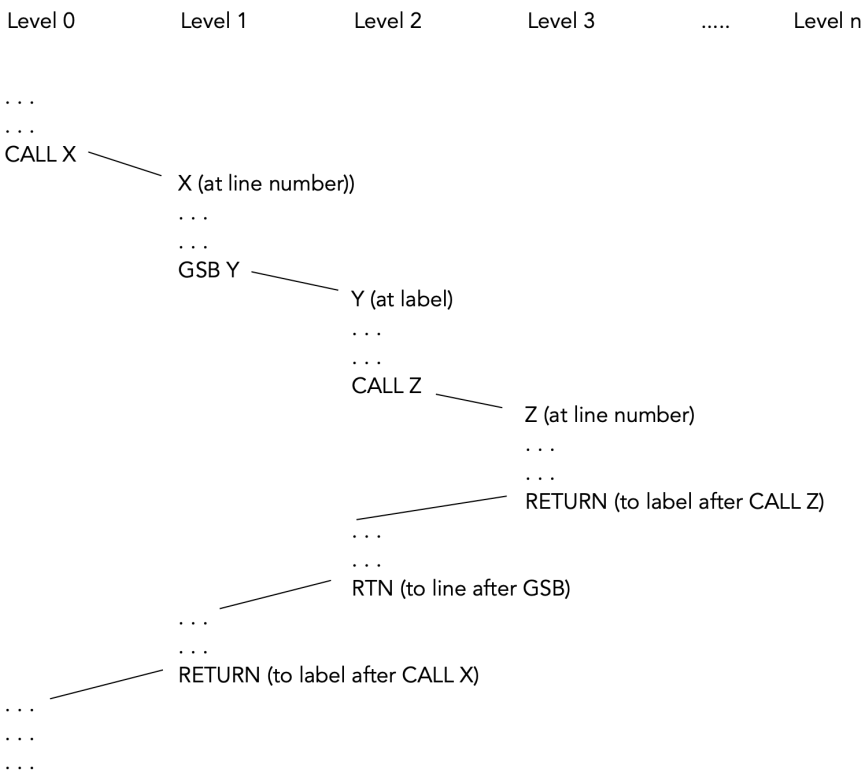
```
          085 …          As an example, let's say your subroutine starts here. It does not need a label.
          086 …          It is called by it's line number 85.
          087 …
          088 …
          089 …
          090 …
          091 …
          092 ..
          093 …
          094 …
RETURN    095 CLX        Put zero as address to indicate a RETURN
          096 GTO 0      Go to X-GSB Handler

          100 LBL A      Let's say your user program starts here
          101 …
          102 …
          103 …
          104 …
          105 8          destination line, return label
          106 5
          107 ,
CALL      108 4          Choose a label e.g. 4 as the return address
          109 GTO 0      Go to X-GSB Handler
          110 LBL 4      Returns here
```

Choose the line number of the subroutine wisely so that it will remain as much as possible at the same address, which is handy when you are debugging your main program.

Mixed use of GSB, RTN and of CALL, RETURN

Within a CALLED subroutine a regular GSB can be used and vice versa.



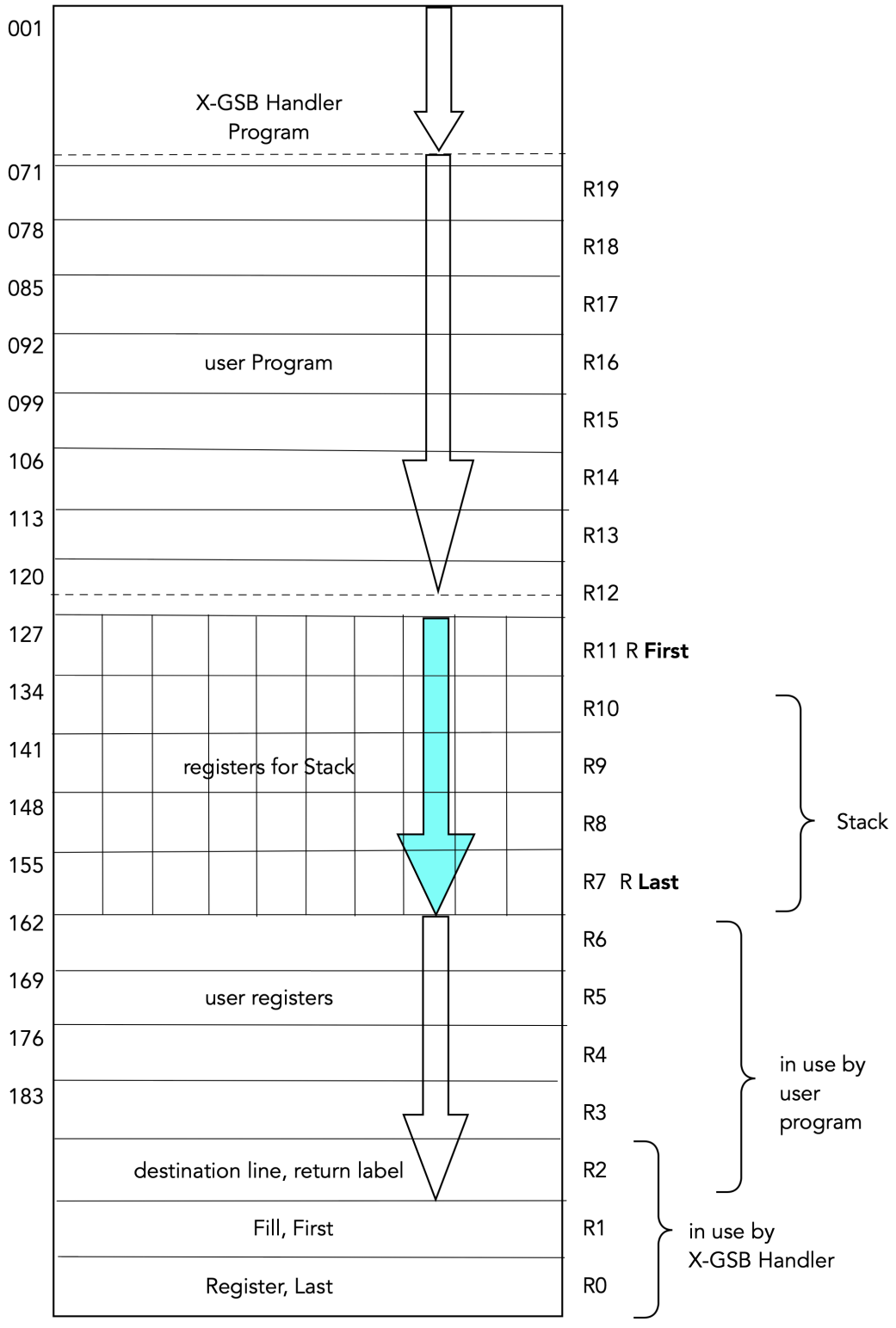
Memory Lay out of the HP 34C when using X-GSB

Each register takes 7 lines of program space. But program lines 001-070 are free program lines that do not affect register space. The user program can be appended after the Handler.

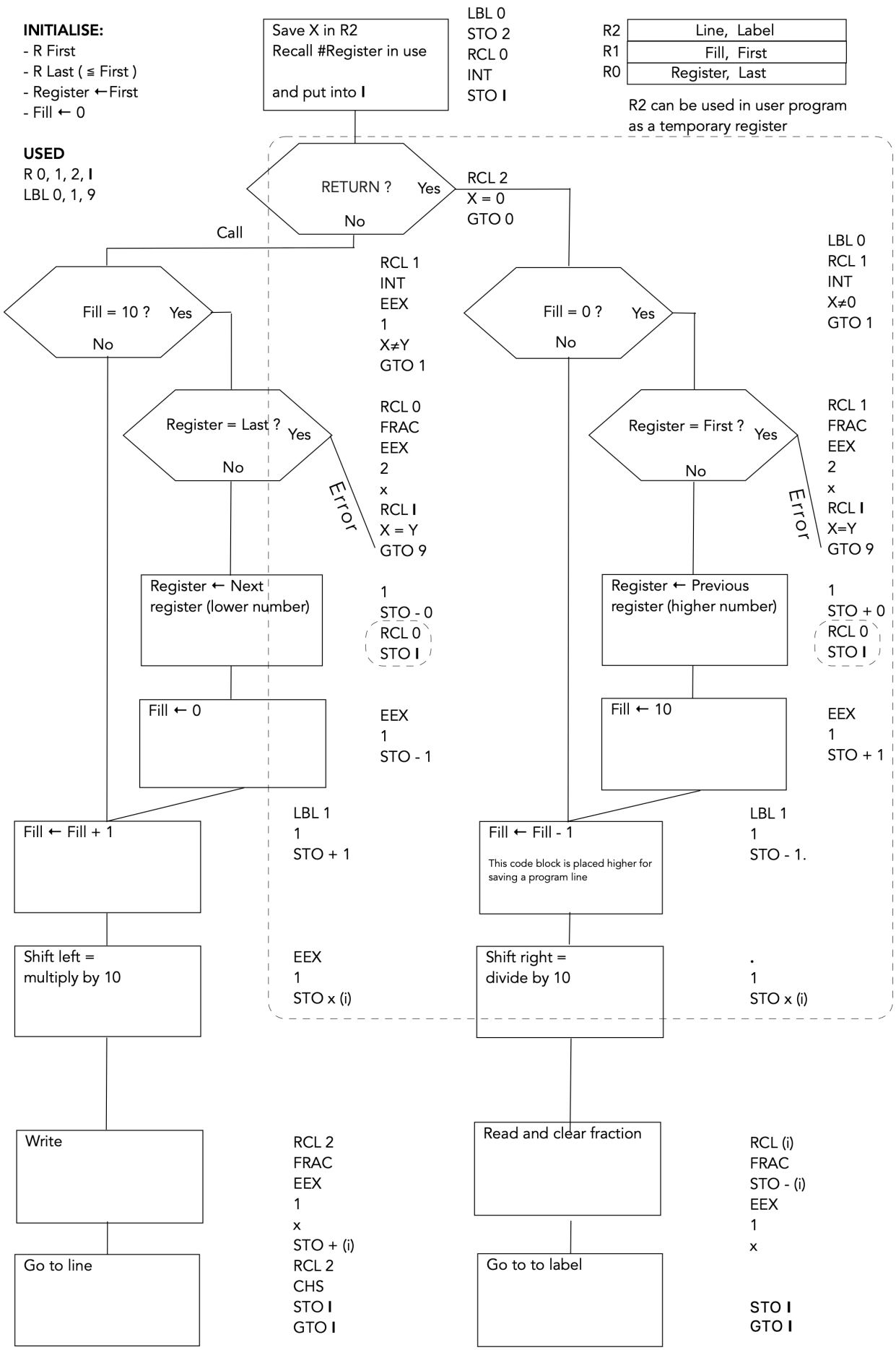
The start of the stack is R First and is the first free register after the end of the user program. The last register of the stack is R Last and is the register just above the highest user register. Thus the last stack register must always be R3 or higher.

The Address_Register R2 is a temporary register and can also be used by the programmer, as a temporary register.

Example



BASIC CODE VERSION for HP 34C



BASIC CODE VERSION for HP34C

```
001 LBL 0
002 STO 2       Store passed values of address and label, given by the CALL
003 RCL 0       Recall the number of the current Stack Register
004 INT
005 STO I       Use I for indirect addressing

006 RCL 2       CALL or RETURN?
007 X = 0
008 GTO 0

                CALL Part                       046 LBL 0       RETURN Part
009 RCL 1       Fill = 10?                      046 RCL 1       Fill = 0?
010 INT                                         048 INT
011 EEX
012 1
013 X ≠ Y                                       049 X ≠ 0
014 GTO 1                                       050 GTO 1

015 RCL 0       Register = Last?                051 RCL 1       Register = First?
016 FRAC                                        052 FRAC
017 EEX                                         053 EEX
018 2                                           054 2
019 x                                           055 x
020 RCL I                                       056 RCL I
021 X = Y                                       057 X = Y
022 GTO 9                                       058 GTO 9

023 1           Use next Register               059 1           Use previous Register
023 STO - 0                                     060 STO + 0
023 RCL 0                                       061 RCL 0
023 STO I                                       062 STO I

027 EEX         Fill is set to zero             063 EEX         Set Fill to 10
027 1                                           064 1
029 STO -1                                      065 STO + 1

030 LBL1        Increase Fill by 1              066 LBL 1       Decrease Fill by 1
031 1                                           067 1
032 STO + 1                                     068 STO - 1

033 EEX         Shift left                      069 .           Shift right
034 1                                           070 1
035 STO x (i)                                   071 STO x (i)

036 RCL 2       Write                           072 RCL (i)     Read and clear fraction
036 FRAC                                        073 FRAC
038 EEX                                         074 STO -(i)
039 1
040 x
041 STO + (i)
                                                075 EEX         Go to label
042 RCL 2       Go to line                      076 1
043 CHS                                         076 x
044 STO I                                       078 STO I
045 GTO I                                       079 GTO I
```

Remarks on the Basic Code version for the X-GSB Handler

1. The large encircled code part can be made shorter by putting it in a GSB subroutine and/or by making use of a flag to combine, but still differentiate between, CALL and RETURN code (e.g. switch between a times ten multiplication and a division by ten). This is done in the code on the next page.

2. An unused label (9) is used to force a halt and an error message (stack overflow or underflow).

3. Re-use of label 0 is possible, because labels are searched forward during execution.

4. When checking for overflow or underflow, it might be slightly better to check for ≥ and ≤ respectively. This may have a better chance of catching errors in the user program.

5. Tricks: EEX 1 is faster than 10. Multiply by **.**1 is faster than divide by 10.

6. Content of **I** is allowed to have a fraction part as GTO **I** only considers the integer part.

7. STO - **I** and STO + **I** are not possible on the HP34C, but are possible on the RPN-34 CE from Cuvee Software. And also on the HP15 CE. See the two small encircled code parts on page 9, that can be converted from two to one line.

8. In the Cuvee RPN-34 CE the program memory and registers are not shared. And it has more registers. Therefore you could use separate registers for variables R First, R Last, Fill and Register. This will make the code for the Handler smaller and faster.

9. Save register space: Maybe R2 is not needed and the value in the X register can be stored/retrieved in the XYZT stack.

10. (Of course) a user program may have to save **I** before a CALL, if he uses I in the user program (for instance with ISG or DSE commands)

11. Test for overflow and underflow could be omitted, but you have to be *very very very* sure that your user program is 100% correct and thus

    1. never exceeds the capacity of the stack or makes too many returns
    2. never changes R0 or R1, either directly or indirectly
    3. never changes Flag 0 apart from using it in a CALL or RETURN
    4. never uses label 9, either directly or indirectly
    5. never jumps to an address in the Handler code (by using a wrong label or address in the **I** register)

IMPROVED CODE VERSION OF X-GSB FOR HP34C

To save space we can combine code of CALL and RETURN (according to remark 1). Let's use a flag to indicate if a CALL or RETURN is asked by the user program. This code uses one label less. The protocol for CALL and RETURN in user program changes slightly. See example below..

```
001 LBL 0
002 STO 2        Store passed value
003 RCL 0        Recall Register in use
004 INT
005 STO I
006 EEX          Put 10 and 0 into Y and X
007 1
008 ENTER
009 0
010 F 0?
011 X ≥ Y        10 when CALL // 0 when RETURN
012 RCL 1        Fill, First
013 INT          get the Fill part
014 X ≠ Y
015 GTO 0

016 RCL 1        Fill, First
017 F 0?
018 RCL 0        Register, Last
019 FRAC         get Last  //  get First
020 EEX
021 2
022 x
023 RCL I
024 X = Y        overflow // underflow
025 GTO 9
026 1
027 F 0 ?
028 CHS
029 STO + 0      move Register to lower one  //  higher
030 RCL 0
031 STO I
032 EEX
033 1
034 F 0?
035 CHS
036 STO + 1      set Fill to 0  //  set to 10

037 LBL 0
038 1
039 F 0 ?
040 CHS
041 STO - 1      increase Fill  //  decrease
042 .
043 1
044 F 0 ?
045 1/x          10 x (shift left)  //  .1 x (shift right)
046 STO x (i)
047 F 0 ?
048 GTO 0
049 RCL (i)      - // read label number, clear fraction
050 FRAC
051 STO - (i)
052 EEX
053 1
054 x
055 STO I
056 GTO I        go to label

057 LBL 0        write label number // -
058 RCL 2
059 FRAC
060 EEX
061 1
062 x
063 STO + (i)
064 RCL 2
065 CHS
066 STO I
067 GTO I        go to line number
```

Registers used: 0, 1, 2, I

- R2 can be used by user as temporary register
- User may have to save I before a call if he uses I in the user program

Labels used: 9, 0. 0 Can be re-used in user program

Flags used: 0

Example of user program:

```
068 …            Let's say your subroutine starts here.
070 …
071 …
…
078 …
079 CF 0         RETURN
080 GTO 0        Go to X-GSB Handler
. . .
100 LBL A        Let's say your user program starts here
101 …
102 …
103 …
104 …
105 6            Address of subroutine
106 8            user passes this info in X
107 ,
108 4            Return address e.g. label 4
109 SF 0         CALL
110 GTO 0        Go to X-GSB Handler
111 LBL 4
```

IMPROVED CODE VERSION OF X-GSB FOR HP34C ***But without checks for overflow and underflow***

```
001 LBL 0
002 STO 2          Store passed value
003 RCL 0          Recall Register in use
004 INT
005 STO I
006 EEX            Put 10 and 0 into Y and X
007 1
008 ENTER
009 0
010 F 0?
011 X ≥ Y          10 when CALL // 0 when RETURN
012 RCL 1          Fill, First
013 INT            get the Fill part
014 X ≠ Y
015 GTO 0

016 RCL 1          Fill, First
017 F 0?
018 RCL 0          Register, Last
019 FRAC           get Last  //  get First
020 EEX
021 2
022 x
023 RCL I
024 X = Y          Overflow or underflow
025 GTO 9
016 1
017 F 0 ?
018 CHS
019 STO + 0        move Register to lower one  //  higher
020 RCL 0
021 STO I
022 EEX
023 1
024 F 0?
025 CHS
026 STO + 1        set Fill to 0  //  set to 10

027 LBL 0
028 1
029 F 0 ?
030 CHS
031 STO - 1        increase Fill  //  decrease
032 .
033 1
034 F 0 ?
035 1/x            10 x (shift left)  //  .1 x (shift right)
036 STO x (i)
037 F 0 ?
038 GTO 0

039 RCL (i)        - // read label number, clear fraction
040 FRAC
041 STO - (i)
042 EEX
043 1
044 x
045 STO I
046 GTO I          go to label

047 LBL 0          write label number // -
048 RCL 2
049 FRAC
050 EEX
051 1
052 x
053 STO + (i)
054 RCL 2
055 CHS
056 STO I
057 GTO I          go to line number
```

Test for overflow and underflow are omitted, thus do not use this version when debugging your program. You have to be *very very very* sure that your program is 100% correct and thus:

- never exceeds the capacity of the stack or makes too many returns
- never changes R0 or R1, either directly or indirectly
- never changes Flag 0 (except for a CALL or RETURN)
- never uses label 9
- never jumps to an address in the Handler code (by using a wrong label or address in the **I** register

One may consider another flag protocol: The default flag is set to CALL. So a calling program never needs to set the flag before calling the handler. A RETURN in the main program resets the flag before going to the Handler. In the Handler the flag is set after the code part for return has been executed. This protocol requires initiating/setting the flag before the user program is started, which might be forgotten. But it will save some program lines when the function is called from multiple locations in the program.

A small program to manually test the X-GSB Handler for push and pop actions

Using keys A and B one can execute a CALL (push) or a RETURN (pop) and check how the stack expands and contracts (with for example label number 8).

Determine your free registers R First and R Last. **Can be R 17 and R3**

Initialize

Clear R First	
Store 0, R First	into R1 (use 2 digits for R First) e.g. **0,17 STO 1**
Store R First, R Last	into R0 (use 2 digits for R Last) e.g. **17,03 STO 0**
001	**start of X-GSB Handler**
. . .	In the case of a CALL: push onto stack, goto line number
	In the case of a RETURN: pop from stack and go to label
067	**End of Handler**
0**68** GTO 8	A sneaky test subroutine that is called
	and immediately returns
069 LBL B	start of **RETURN** (make a pop)
070 CF 0	
071 GTO 0	go to Handler at label 0 (line number 001)
072 LBL A	start of **CALL** (make a push)
073 **6**	prepare addressinfo as a number with integer
074 **8**	and fraction part: line#,label#
075 ,	
076 **8**	
077 SF 0	
078 GTO 0	go to Handler at label 0 (line number 001)
079 LBL **8**	program returns here after execution of subroutine
080 RCL **.**7	show the contents of a register of your choice (e.g. R First)
081 RTN	Normal end

A small program to test the X-GSB Handler with a simple recursive function

The subroutine is called recursively a number (n) of times. And then the same number of returns is made. A counter t keeps track of the total of number of calls + number of returns and thus t should be twice n once the program is finished. In pseudo code:

```
user must first   initialize Handler

                  initialize program

program           a ← 1  counts up to n and then down from n to 1
                  t ← 1
                  CALL test
                  label 2
                  get t
                  stop

test              if a < n
                        label 3
                        a ← a+1
                        t ← t+1
                        CALL test
                        label 4
                  else
                        if a > 0
                              label 5
                              a ← a -1
                              t ← t +1
                              RETURN
                        else
                              label 6
                              RETURN
```

Initialize Handler:
- Clear all registers used by Stack
- Set R1 to Fill=0 , First (2 digits)
- Set R0 to Register=First, Last (2 digits)

R12 can be R First
R06 can be R Last

Thus n can be 70 max

Initialize test program:
- Put n into X
- Start program A

R3 used for a
R4 used for t
R5 used for n

LBL 0	used by handler
LBL 1	not used
LBL 2	not used
LBL 3	used
LBL 4	used
LBL 5	used
LBL 6	used
LBL 9	used by Handler

Flag 0	used by Handler

```
001               Handler
067

068 RCL 3         a          here starts test
069 RCL 5         n
070 x > y
071 GTO 3
072 RCL 3         a
073 x > 0
074 GTO 5
075 GTO 6

076 LBL 3
077 1
078 STO +3        increase a
079 STO + 4       increase t
080 6             CALL test
081 8
082 ,
083 4
084 SF 0
085 GTO 0
086 LBL 4         return label after the CALL is finished
087 LBL 5
088 1
089 STO - 3       decrease a
090 STO + 4       increase t

091 LBL 6
092 CF 0          RETURN
093 GTO 0

094 LBL A
095 STO 5         n
096 1
097 STO 3         a
098 STO 4         t
099 6             CALL test
100 8             line number of test
101 ,
102 2             return label after the CALL is finished
103 SF 0
104 GTO 0         go to Handler
105 LBL 2
106 RCL 4         recall t
107 RTN           stop
```

The Fibonacci function using the X-GSB Handler and a data stack

A familiar example of a recursive function is the Fibonacci number sequence F(n) = F(n − 1) + F(n − 2).
This function is recursively defined in terms of itself and is reducible to non-recursively defined values, the so called base cases 0 and 1.

F(0) = 0 and F(1) = 1.

n	0	1	2	3	4	5	6	7	8	9	10	11
F(n)	0	1	1	2	3	5	8	13	21	34	55	89

The general form of this function may be written recursively. In pseudo code:

```
F(n) {
        if      n = 0 return  0
        if      n = 1 return  1
        else  return F(n-1) + F(n-2)
    }
```

The Fibonacci calculations can be visualized as a binary tree. It starts with the root and has nodes below. Each node is a new instance of the function. The function is called just so many times until it reaches the lowest node with n=2 which has a leaf of value 1, the base case value.

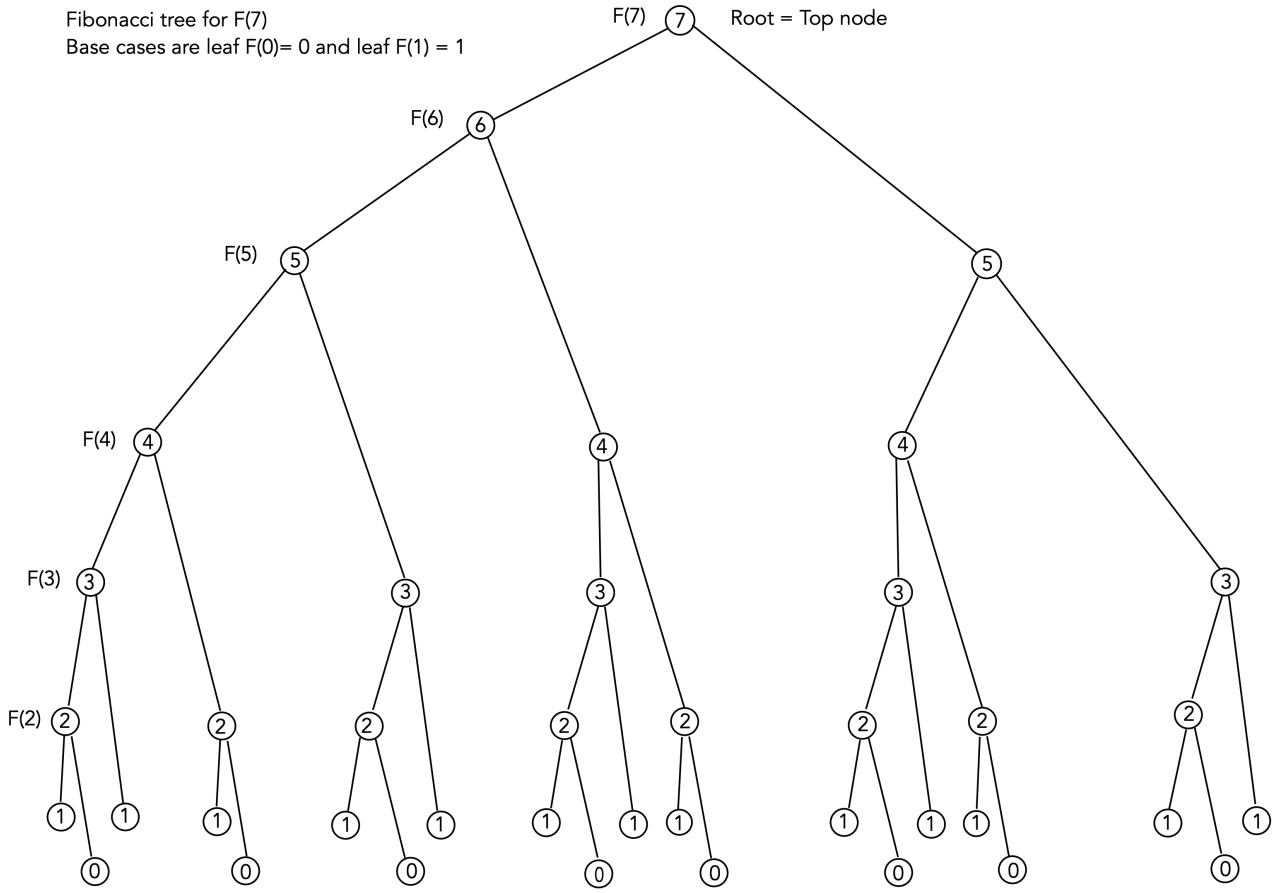
The tree is first traversed on the left side all the way down (until it finds 2 as a node and then finds a "1" by calculating F(n-1). Then works it way to the right and upwards (and to the left and downwards sveral times, when needed) until it hits the root and then starts with the right part of the tree. The final value of the Fibonacci function can also be viewed as the number of leafs with value 1. We will make use of this in our program. A counter t is increased every time a 1 in the tree is encountered.

On each call downwards n is decreased. On each return upwards it must increase n again. The address stack keeps track of the proper return addresses, but we also need to save information about the position within the tree. Think of it as a maze. If the tree gets larger than about 4 deep it is impossible to determine our position within the tree or to determine the next node to visit, because we only have a global n and no history to tell us if we visisted a node before. A global value of n cannot work, the program would make double counts or keeps looping endlessly (until the Stack overflows). Therefore the local value of n that the calling program uses must be saved on a stack. Thus we need a data stack.

As the HP34C with the X-GSB Handler and the Fibonacci program does not have many program lines and registers left and this program is only intended as a demonstrator, I have implemented only a small datastack: one register for 10 (single digit) values of n. Thus one could think that the program works for n = 0 up to n = 9. However…

However, with a little trick we can make the single register data stack also accomodate the two digit numbers n =10 and n = 11. This is because the datastack is not used for the leafs 0 and 1. As explained above, the program will search downwards until it hits a node 2. Thereafter a leaf 1 is found. Thus the data stack will, just before that, have pushed a 2. The datastack will never contain a number lower than 2. And the highest number will be n. This means that, with n = 2 …9, the data stack will only use 8 positions of the 10 digits available. If we transform the number sequence 11… 2 into 9 … 0 we will make use of every digit of the 10 digit register, and thus be able to store 10 single digit node values. A simple subtract by 2 before we push and an addition of 2 after a pop will do the job. This results in the program being able to calculate F(n) for n = 0 up to n = 11. See the remarks in red in the program code. The Stack depth needed for return addresses is 11, because Fibo is called the very first time from program A. Fibo itself never goes deeper than 10.

Please note: The algorithm used is basic and can be improved. For instance: It isn't necessary to visit the base case 0, as it does not contribute to the total counts of . Going down the tree until you hit a two and then increase the count will save a lot of steps. And of course, a Fibonacci number can absolutely be more easily calculated with a straightforward for loop, but I have used Fibonacci only to demonstrate the recursive function capabilities of the HP34C with the X-GSB Handler.



Performance

The program has been tested on the original HP34C and some other HP machines. It is clear, and totally expected, that the 45 year old HP34C is laughably slow compared to machines or programs with today's technology. But it did the job!

	n = 7	n = 11	
HP 34C	5 min 45 s	40 min 35 s	Original
HP-34 E SLP	2 min 25 s	16 min 50 s	Other processor chip
HP 15CE	3,5 s	20 s	Relatively new handheld calculator
RPN-34 CE		3,3 s	Emulation on iPhone. 750 times faster.

Pseudo code

```
F(n) {
        if      n = 0 return  0
        if      n = 1 return  1
        else  return F(n-1) + F(n-2)
    }
```

F(n) in the form of pseudo code for HP34C:

```
INITIALIZE      registers for the address stack and data stack

USER            put n into X
                start A

LBL A           put X into Arg          Arg is the argument that is given to the FIBO function
                Initialize t to 0       t is the number of 1's found and is equal to F(n)
        CALL FIBO (Arg)
        label 5                         return address after FIBO is finished
                RCL t                   Show value of t = F(n)
                RTN                     normal RTN to make calculator stop

Address FIBO    RCL Arg                 Put Arg (the current n) into X register
                 X=0
                GTO RETURN
                1
                X=Y
                GTO INCREASE

                RCL Arg                 Else: Arg ← Arg -1 and try again
                PUSH onto data stack    trick: first subtract 2 before storage into data stack
                1
                STO - Arg
        CALL FIBO                       with Arg - 1
        LBL 3                           return address after FIBO (Arg-1) is finished

                READ previous Arg from data stack and now calculate  FIBO(Arg-2)
                                        trick: add 2 after reading

                2
                -
                STO Argument
        CALL FIBO                       with Arg - 2
        LBL 4                           return address after FIBO (Arg - 2) is finished

                POP                     Pop previous Arg from data stack
                                        trick: add 2 after pop
                STO Arg
                GTO RETURN

LBL INCREASE    STO + t

LBL RETURN      CF0
                GTO 0                   Go to Handler
```

```
000          X-GSB Handler
....
067
068 RCL 3    Recall Arg
069 X = 0
070 GTO 1    Go to RETURN
071 1
072 X = Y
073 GTO 2    Go to INCREASE
074 RCL 3

075 EEX      PUSH onto data stack (R5)
076 1
077 STO x 5  shift left
078 X ≷ Y
079 2        trick. subtract 2
080 -

081 STO + 5  write into most right digit
082 1        Arg is now n - 1
083 STO - 3
084 3        Prepare a CALL to FIBO
085 GTO 6
086 LBL 3

087 RCL 5    READ previous Arg
088 .
089 1
090 x
091 FRAC
092 EEX
093 1
094 x
             trick. 2+ and 2 -  = do nothing
095 STO 3    Arg is now n -2
096 4        Prepare a CALL to FIBO
097 GTO 6
098 LBL 4

099 .        POP from data stack and clear fraction
100 1
101 STO x 5
102 RCL 5
103 FRAC
104 STO - 5
105 EEX
106 1
107 x
108 2        trick. add 2
109 +
110 STO 3    Put previous Arg into Arg
111 GTO 1    Go to RETURN

112 LBL 2    INCREASE
113 STO + 4  result ← result + 1

114 LBL 1    RETURN
115 CF 0
116 GTO 0

117 LBL 6    Common code for all CALL'S
118 .
119 1
120 x
121 6
122 8
123 +
124 SF 0
125 GTO 0    Go to Handler for a call

126 LBL A
127 STO 3    Store X in Arg
128 0
129 STO 4    Reset result
130 5        Prepare a CALL to FIBO
131 GTO 6
132 LBL 5
133 RCL 4    Show result
134 RTN
```

INITIALIZE

```
CLR REG
0,09 STO 1   Set Fill to 0, R First to R09
9,08 STO 0   Set Register to 9, R Last to R08
             Use one address register
0 STO 5      Use one data register R5
```

USER — type n into X (n must be a positive integer with maximum 11)
start A

OVERVIEW

R0, R1, R2	used by Handler
R3	Argument
R4	result
R5	data register
R6, R7	FREE
RI	used by Handler
LBL 0, 9	used by Handler
LBL 1	RETURN
LBL 2	INCREASE
LBL 3	Return address
LBL 4	Return Address
LBL 5	Return Address
LBL 6	Common code for CALL's
LBL 7, 8	FREE
Flag 0	used by Handler

A few remarks on reducing the code

The program can be made 10 lines shorter by using the Handler without the error checking for overflow and underflow. And another 9 lines by omitting the program of LBL A (then we would start the program by filling in the parameters, setting the program at line 001 and press R/S. Then we would also not need R8 for the Stack (as the CALL from A takes 1 push onto the address Stack and makes the required Stack depth 11, while FIBO only needs 10)

Some considerations for a bigger data stack

If we wanted to have a datastack for n > 11 then we must store two digit numbers in the data stack. This means that one register can hold 5 numbers. So for 20 numbers we would need 4 registers. But with this particular Fibonacci function we will then unfortunately very likely run out of code space.

The stack method of the Handler for multiple registers can also be used for a data stack. Even for a data stack of two digit numbers. Then one simply stores the number as two consecutive single digits. Some code to pack and unpack the number is of course necessary.

The X-GSB Handler uses multiple registers of which only one is the current 'stack'. Another way of building a stack mechanism is given below.

We can make an alternative generic design for a single long stack for two digit numbers and make use of the ISG and DSE instructions. Let's say we have four registers R5 … R8 and we make them into one long 40 digit stack. This differs from the stack mechanism of the X-GSB Handler, which would logically also look like a single 40 digit stack, but under the hood it are four 10 digit stacks.

initialize:

8,004
STO **I**

PUSH

First shift *all* registers two places to the left
(this is different from the X-GSB Handler mechanism, that only shifts the current register)

```
LBL START
EEX          shift left and start with the highest register
2
STO x (i)
DSE          just decrement I
.            dot can be omitted in some cases
RCL (i)
EEX
8
CHS
x
INT
ISG          just increment I
.            dot can be omitted in some cases
STO + (i)
DSE          do for 4 registers
GTO START
```

Take care: R5 now has as it lowest two digits the highest two digits of R4, which should be 0

Then add a two digit number to R5
etc…