Report

1. Introduction of Unum computing

We now have several different IEEE standard binary floating point precisions: 16-bit(half), 32-bit(single), 64-bit(double) and 128-bit(quad). Currently, programmers have to choose one of the precision they think is right for their programs. However, if they choose too little, it will lead to low resulting accuracy and if they choose too much precision, the resulting program will consume unnecessary storage space. Therefore, it’s time to re-examine this floating point format and see if we can make something better. The “universal number” (or unum) is able to vary the precision and dynamic range to the optimum number of bits, and also record whether the number is exact or lies within a range, instead of rounding the number. A unum is a bit string of variable length that has six sub-fields: sign bit, exponent, fraction, uncertainty-bit (ubit), exponent size (es), and fraction size (fs).

As a quick review, here is what the unum format looks like:

<---------es bits--------><------------------------------fs bits----------------------------> <--esize size bits--><----fsize size bits---->

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| s | e | f | u | es-1 | fs-1 |

sign exponent fraction ubit exponent size fraction size

To represent IEEE single precision:

<----------8 bits--------><------------------------------23 bits---------------------------->

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| s | e | f | u | 111 | 10110 |

The last two fields indicate the number of bits in the exponent and fraction respectively, allowing both to change with every calculation. The number of bits for the fields that describes the exponent size and fraction size are set in the environment (set the esize size and fsize size). For example, a (2,3) environment indicates the esize size is 2 and fsize size is 3. Therefore, in this environment, we can have up to 2^2 = 4 exponent bits (esize) and 2^3 =8 fraction bits (fsize).

The left three fields are almost the same to IEEE floating point, except better rules about Not-a-Number (NaN) and Infinity and a better way to handle overflow, underflow and rounding cases with the ubit. (Note: all the examples below are based on (2,3) environment)

* Infinity: With all the bits set to 1 except the ubit with the maximum exponent size and fraction size.

0 1111 11111111 0 11 111 is positive infinity

1 1111 11111111 0 11 111 is negative infinity

0 1 1 0 00 000 is not infinity because it does not use the maximum esize and fsize

* NaN: With all the bits set to 1 with the maximum exponent size and fraction size

0 1111 11111111 1 11 111 is quiet NaN (beyond positive infinity)

1 1111 11111111 1 11 111 is signaling NaN (beyond negative infinity)

* Rounding: If the ubit is set, the unum is inexact, which represents the open interval between two exact unums (interval: (exact number, exact number + ULP)). Otherwise, the unum is exact and represents a single number.

Note: A ULP is the difference between exact values represented by bit string that differ by one Unit in the Last Place, the last bit of the fraction.

0 1 0 0 00 000 represents the number 2,

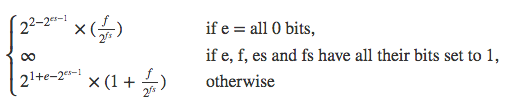
0 1 1 0 00 000 represents the number 2.2

0 1 0 1 00 000 is the interval (0 1 0 0 00 000, 0 1 1 0 00 000) = (2, 2.2)

* Overflow/Underflow: The ubit also marks the case of a value being between the largest representable real number and infinity or the smallest magnitude representable number and zero.

Overflow: 0 1 1 1 00 000 is the interval (0 1 1 0 00 000, +Inf)

Underflow: 0 0 0 1 00 000 is the interval (0, 0 0 1 0 00 000)

 The following formula shows how to convert exact unums to real numbers:

There are two import concepts in the unum format, that is, the u-layer and the g-layer:

1. Unum layer (u-layer)

The u-layer is the level of computer arithmetic where all the operands are unums and data structures made from unums, like ubounds. A ubound is a single unum or a pair of unums that represent a mathematical interval of the real line. Closed endpoints are represented by exact unums, and open endpoints are represented by inexact unums.

Suppose an exact unum *u* represents the exact value 2 and another inexact unum *v* represents the interval (3, 3.5). Thus. The pair {u, v} is a ubound, representing the interval [2, 3.5). A ubound where both endpoints are the same represents the same number or range as the unum endpoints.

1. General layer (g-layer)

The g-layer is the scratchpad where results are computed such that they are always correct to the smallest representable uncertainty when they are returned to the u-layer. A gbound is the data structure used for temporary calculations at higher precision than in the unum environment. Basically, a gbound is made up of a pair of gnums, representing the endpoints of an interval.

1. Unum review
2. Benefits of unum format

* Handling overflow/underflow cases

Instead of overflowing to infinity, underflowing to 0 or rounding to the nearest even, we can use an interval to represent the result by setting the ubit.

* Storage saving

One remark advantage of unum format is the unfixed exponent and fraction size, which enables us to use the shortest binary string to represent exactly the same value as IEEE format. E.g.

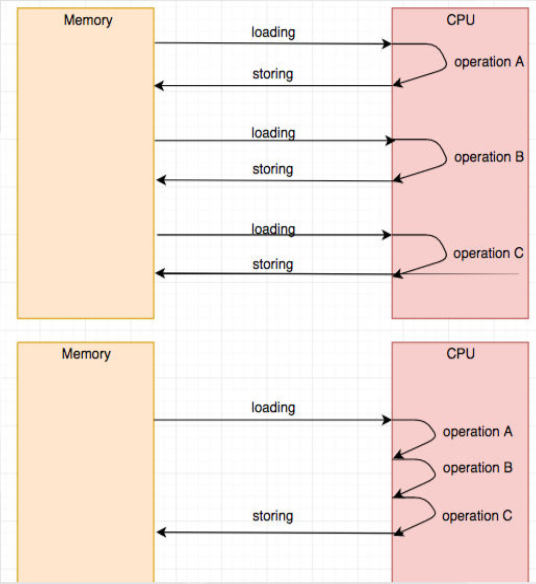
IEEE single: 0 10000000 10000000000000000000000 = 3.0

Unum format with {1,1} environment 0 1 1 0 0 0 = 3.0

Note: With smaller environment, the unum string is shorter, but the range of the values it can represent also gets narrower.

1. Drawbacks of software implementation

First of all, theoretically, the length of each unum is not fixed and therefore, how to represent unum bit string is tricky. One way to do that is to manage memory allocation and de-allocation for each unum. Another way to do that is to use a fixed length integer to represent each unum. In this way, the storage-saving benefit of unum will be sacrificed. However, this is the only method we could adopt to represent unum if we want to design hardware circuits for unum.

Furthermore, one unum operation (e.g. conversion, addition, etc.) consists of a lot of hardware operations (e.g. shifting, bit testing. With software implementation, we need to load from and then write to memory multiple times, making it inefficiency. Note that the operation such as loading and writing are the most expensive ones.

1. Advantage of using FPGA (Vivado)

Firstly, with FPGA, we can design a hardware circuit, acting like a container, which concatenates all the operation in it and in this way, only one time of loading and writing is enough. The image below compares the difference between software and hardware implementation.

Secondly, considering I have no experience with hardware design, it’s pretty tricky for me use HDL (hardware description language) to design hardware circuit directly. A good news is that Vivado provides high level synthesis, which enables me to first use C++ code to describe the hardware behavior and then use Vivado HLS IDE to convert the C++ code automatically to HDL.

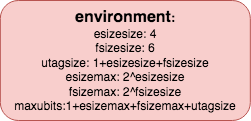
What’s more, Vivado also provides external library to support arbitrary precision integer and fixed-point type, which can be used to represent unum and gnum respectively.

1. Design and Implementation
2. Basic idea

Considering I have no experience with hardware design, it’s pretty tricky for me use HDL (hardware description language) to design hardware circuit on FPGA directly. Fortunately, Vivado Design Suite provides high level synthesis, enabling me to first use C++ code to specify the hardware behavior and then use Vivado HLS IDE to convert the C++ code automatically to HDL(Verilog/VDL). What’s more, Vivado also provides external library to support arbitrary precision integer and fixed-point type, which can be used to represent unum and gnum respectively (see details in the next section).

1. C++ code design
2. Unum environment

Generally, unum environment indicates the esizesize and fsizesize. Note that a (3,5) environment is enough for the IEEE single precision and (4,6) is enough for the double precision.

* esizesize: the number of bits allocated to store the size of exponent
* fsizesize: the number of bits allocated to store the size of fraction
* utagsize: the size of ubit, esizesize, and fsizesize
* esizemax: the maximum size of the exponent
* fsizemax: the maximum size of the fraction
* maxubits: maximum number of bits of a unum

1. Basic data structure

In order to improve efficiency and utilize as less memory space as possible, I decide to use arbitrary precision type for integers and arbitrary precision fixed point type for decimals (both are external libraries provided by Vivado).

* unum

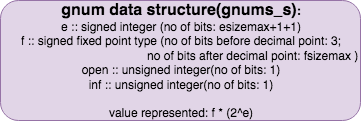
data type (unum\_s): maxubits-bit unsigned integer.

* ubound

p: a single-bit flag, indicating whether the ubnd is a single unum or a pair of unums

l: the left end point

r: the right end point

* gnum

e: exponent value

data type: (esizemax+1+1)-bit signed integer (1 for sign and 1 for potential overflow)

f: fraction

data type: signed fixed-point type (before decimal point: 3 bits, 1 for hidden bit of mantissa, 1 for potential mantissa overflow and 1 for sign; after decimal point: fsizemax bits)

open: a single-bit flag, indicating the interval is open or closed

inf: a single-bit flag, indicating the gnum is infinity or not (if this flag is set, f=1 means positive infinity and f=-1 means negative infinity)

Note: An alternative way is to store the value f \* 2e in gnum. However, in my design, I choose to separate e and f. There are several advantages of doing that. The first reason is storage saving. Given an environment, the maximum exponent value of a unum is 2esizemax-1 and therefore, the value f \* 2e will need up to 2esizemax-1+3 bits and fsizemax bits before or after the decimal point respectively. Nevertheless, in my design, only esizemax+2 bits for e and fsizemax bits for f are enough. (See the table below)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| esize size | esizemax | Store e and f separately | Store the value f \* 2e | difference |
| 1 | 2 | 4 + fsizemax | 22-1+3+fsizemax = 5+fsizemax | 1 |
| 2 | 4 | 6 + fsizemax | 24-1+3+fsizemax = 11+fsizemax | 5 |
| 3 | 8 | 10 + fsizemax | 28-1+3+fsizemax = 131+fsizemax | 121 |
| 4 | 16 | 18 + fsizemax | 216-1+3+fsizemax = 32771+fsizemax | 32753 |

Secondly, if we store the value f \* 2e, when converting gnum back to unum, we still need to find f and e of the value, which is extremely complicated. By contrast, that procedure is no longer necessary if we store e and f directly.

* gbound

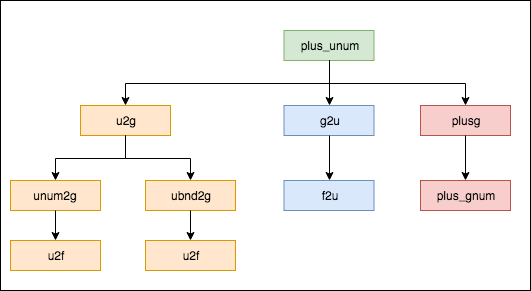
nan: a single-bit flat, indicating the gbnd is NaN or not

l: the left end point

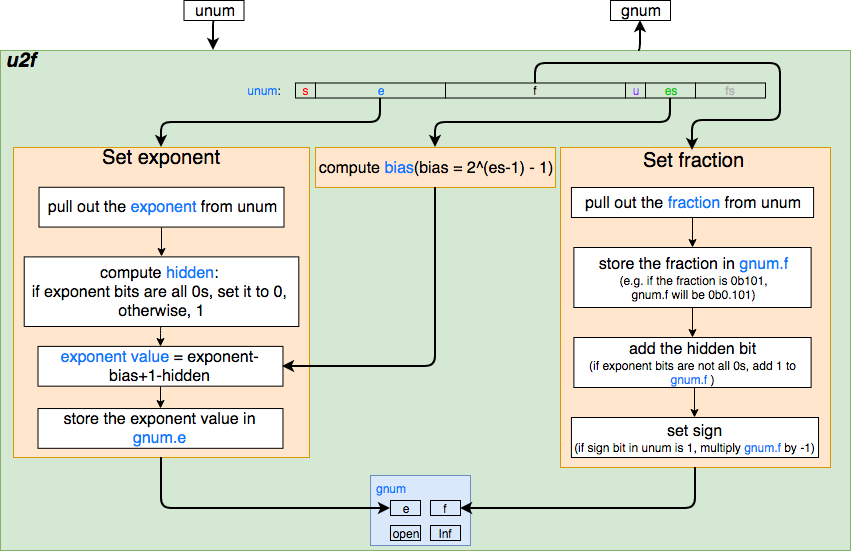
r: the right end point

1. Functions

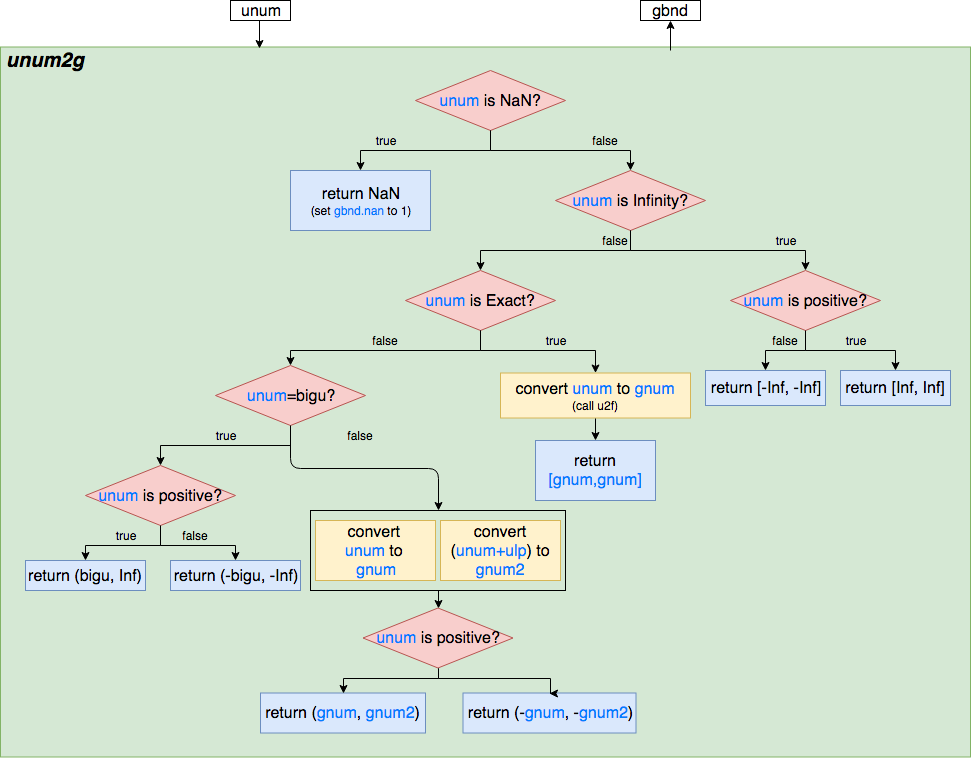
The picture below shows the hierarchy of the functions:



* u2f: convert unum to gnum



* unum2g: convert unum to general interval

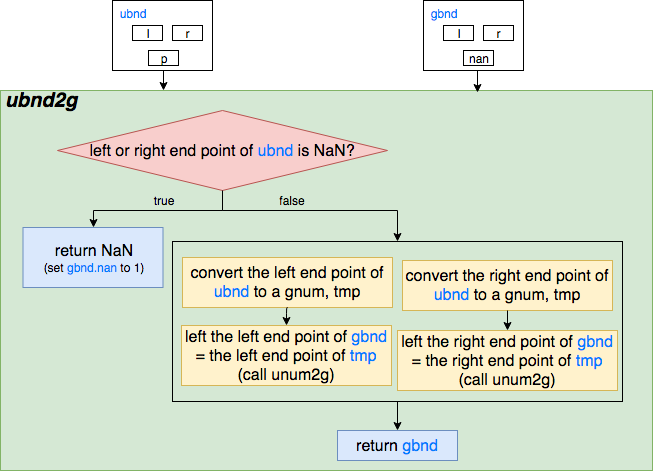


Note: bigu is the unum representing the biggest exact real expressible with the same utag (e and f are all 1 bits). E.g. *u* = 0 1 11 0 00 001. When converting

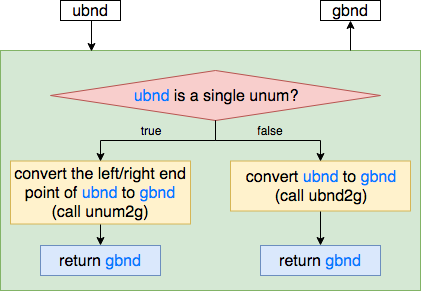
0 1 11 1 00 001, it is the overflow case and therefore we should return *(u, +Inf)*

where *u* is the gnum converted from *u*.

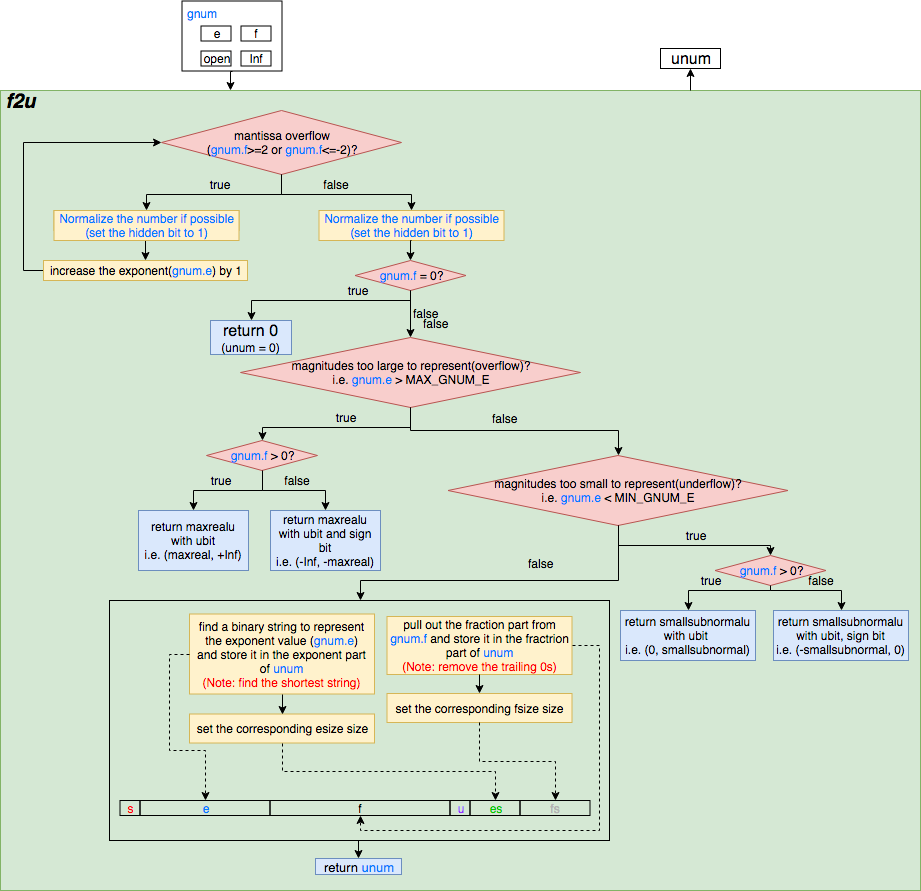
* ubnd2g: convert ubnd to a general interval



* u2g: convert u-layer (either unum or ubnd) to gbnd



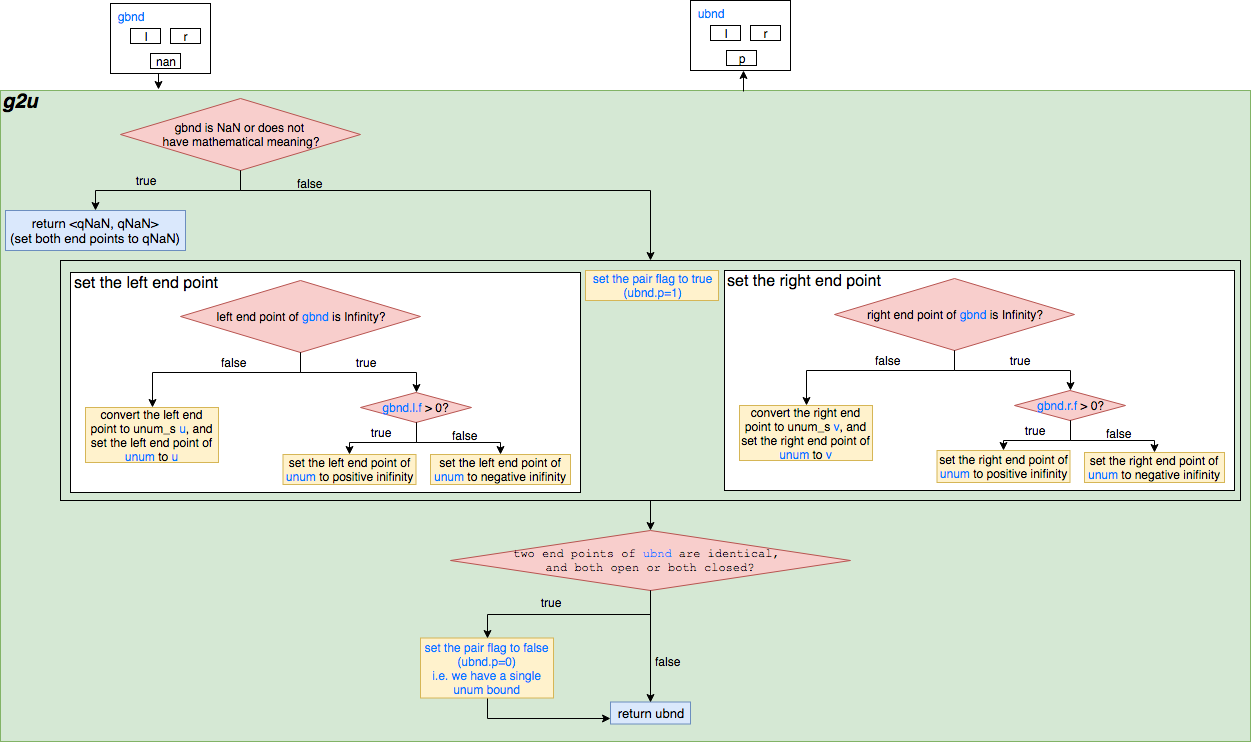
* f2u: convert gnum to unum



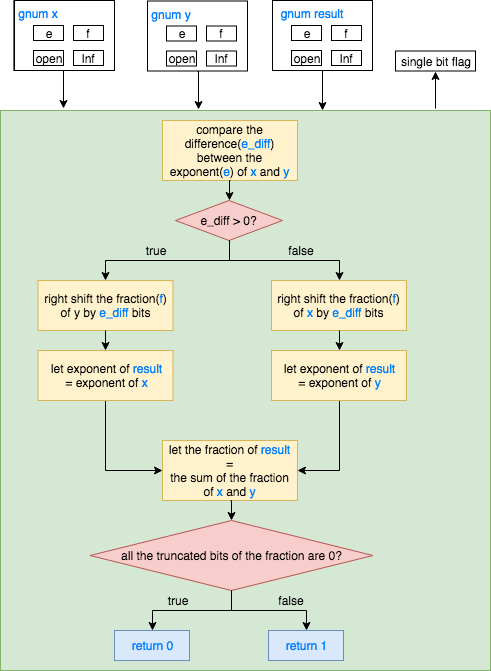
Note: (the following examples are based on (2,2) environment)

1. *MAX\_GNUM\_E*: the maximum exponent value in the current environment (2esizemax-1)
2. *MIN\_GNUM\_E*: the maximum exponent value in the current environment (2-2esizemax-1)
3. When pulling out the fraction from the *gnum.f,* we need to remove all the trailing 0s in order to find the shortest unum representation. E.g. if *gnum.f* = 1.1000, instead of storing 1000 in the fraction part and setting fsize to 4, we can simply store only 1 in the fractioin and set fsize to 1.
4. When using a binary string to represent the exponent value, we also need to find the shortest representation. E.g. if *gnum.e* = 1, instead of storing 10 in the exponent part and setting esize to 2(bias=1, exponent value=2-1 = 1), we can simply store 1 in the exponent part and set esize to 1 (bias=0, exponent value=1-0 = 1).

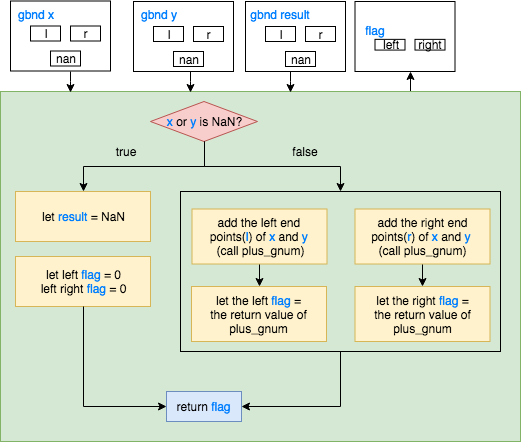
* g2u: convert gbnd to ubnd



* plus\_gnum: add two gnums



* plusg: add two gbnds



1. Evaluation
2. Timeline
3. Future work
4. Summary