Complex MAGIC Simulation Specification Document

Stav Belogolovsky & Amnon Wahle

VERSION 1.0 5 May 2018

Contents

1. Introduction

Complex MAGIC simulation is designed to validate and prove the work and efficiency of different algorithms for complex numbers operations within MAGIC memory.

For better understanding of concepts in this document, the reader should be familiar with basic knowledge in the following subjects:

- memristors functional concepts in general
- MAGIC NOR functional operation
- MAGIC memory array logic concepts of work

The simulation was written in Python 2.7 to serve as an open source base for future work of many more MAGIC memory simulations and applications proof of concept.

The simulation imitates the actions needed for executing operations inside MAGIC memory without any data traffic in or out of the memory, thus acting as the memory controller. While doing so, the simulation updates data outputs and counters from the memory and compare it to the expected, calculated result. The simulation is implemented so it can easily plot graphs of these values for visually presenting results.

Running files were added to ease the execution of each application individually.

Python libraries needed:

- Numpy
- matplotlib
- BitVector
- random

The simulator was developed by Amnon Wahle and Stav Belogolovsky as a part of their B.Sc. project in the Electrical Engineering faculty, at the Technion Israel Institute of Technology, under the supervision of Ameer Haj-Ali.

2. Overall structure and description

2.1. description

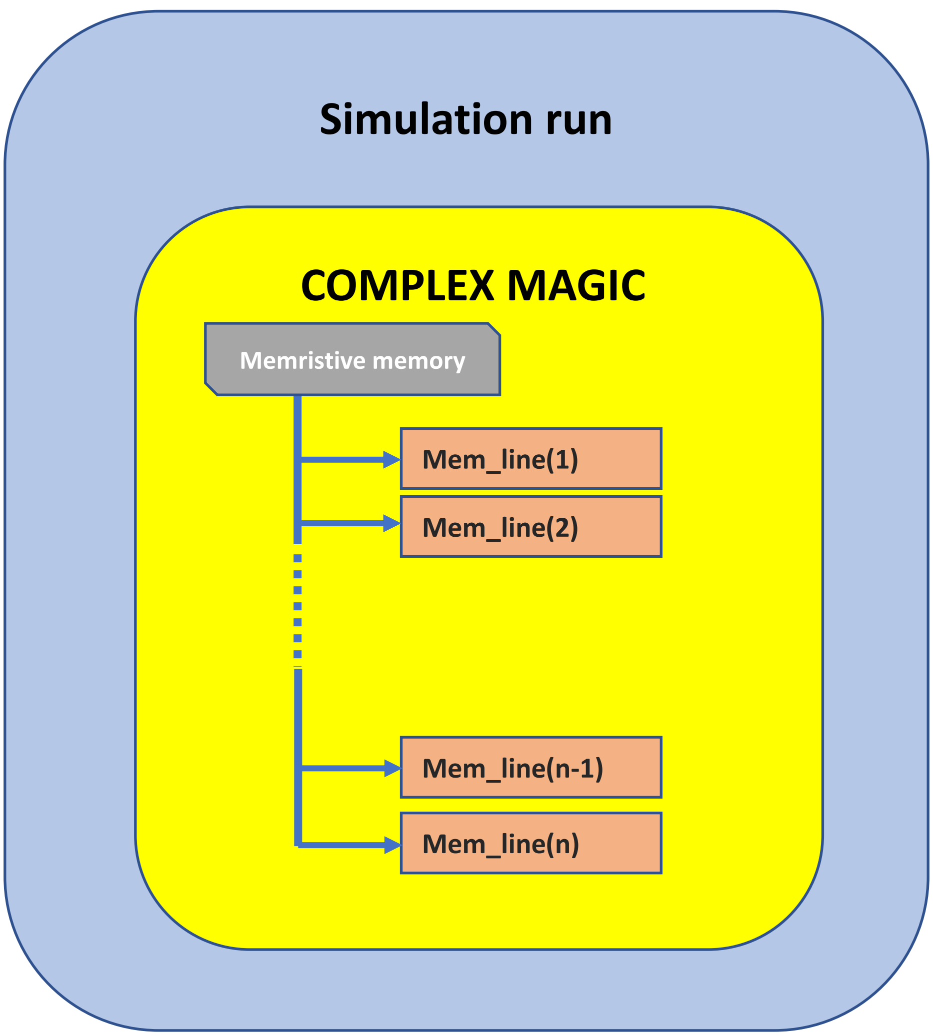
The simulation is built to simulate the work of Memristive memory using MAGIC in bits and addresses level. Working with numbers within memory is in 2's complement representation with fixed point operations only. The simulation functions as if it is memory controller running those bits, ordering the memory which function to execute on what addresses.

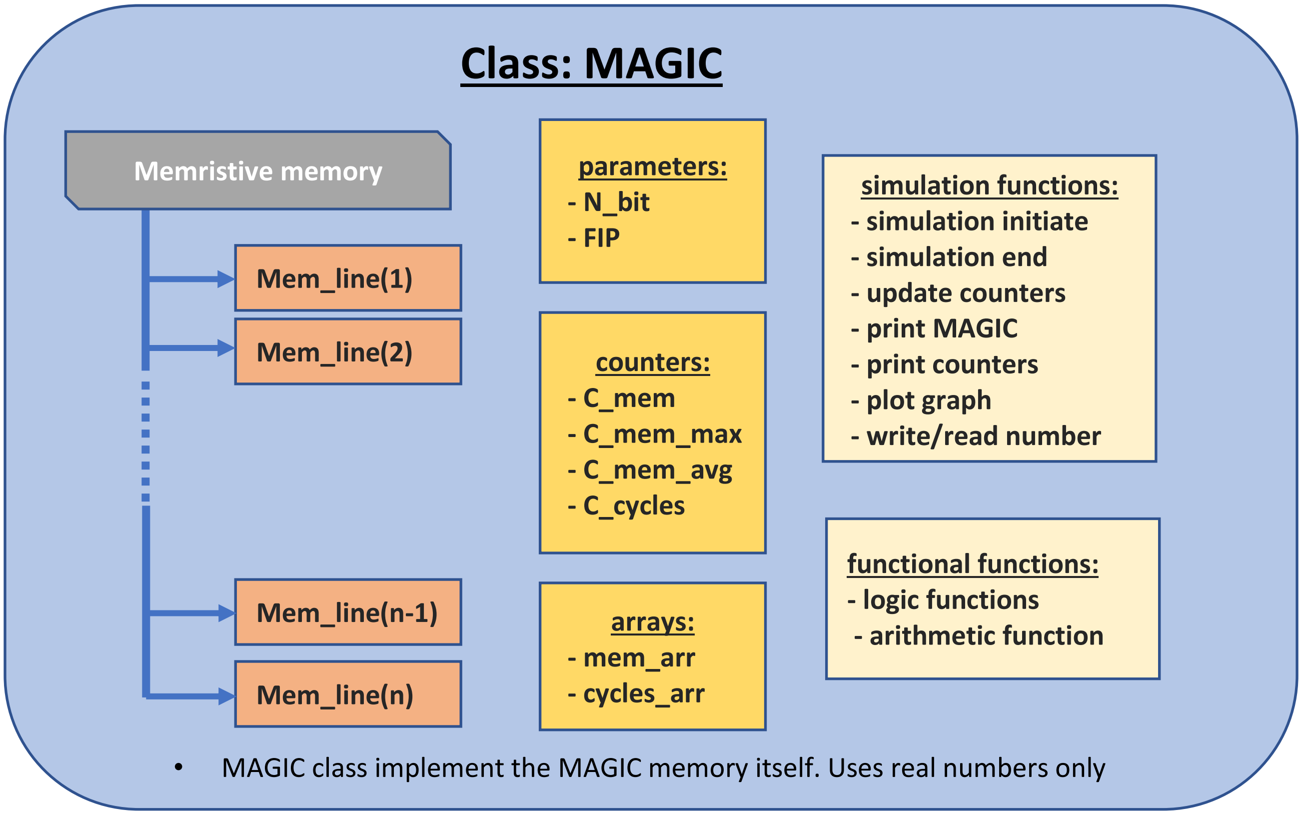
Every simulation is done by creating a COMPLEX MAGIC object, calling initiation function and then run any wanted function. At the end a "simulation end" function has to be called for getting the parameters out of the simulation run.

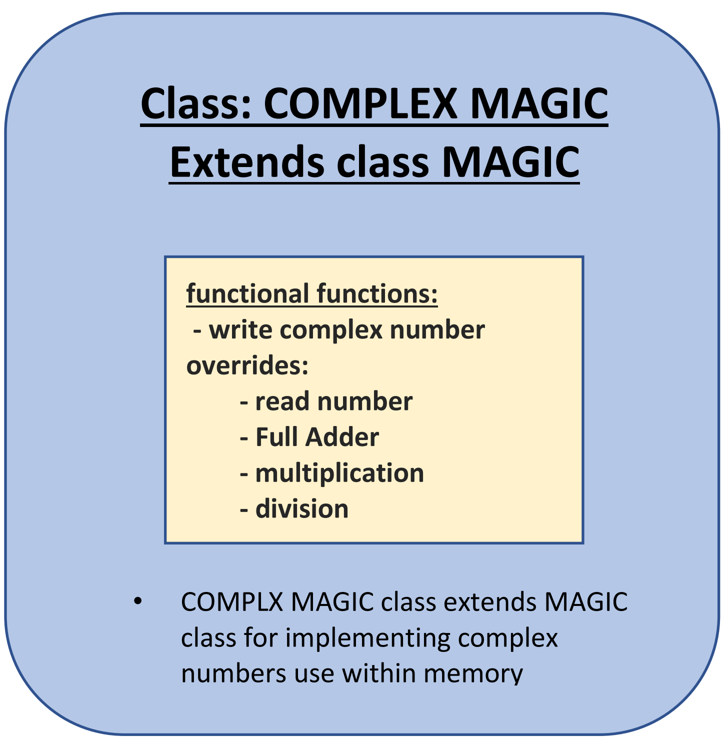
Simulation uses 3 classes: COMPLEX MAGIC, MAGIC, MEM_LINE and some global functions.

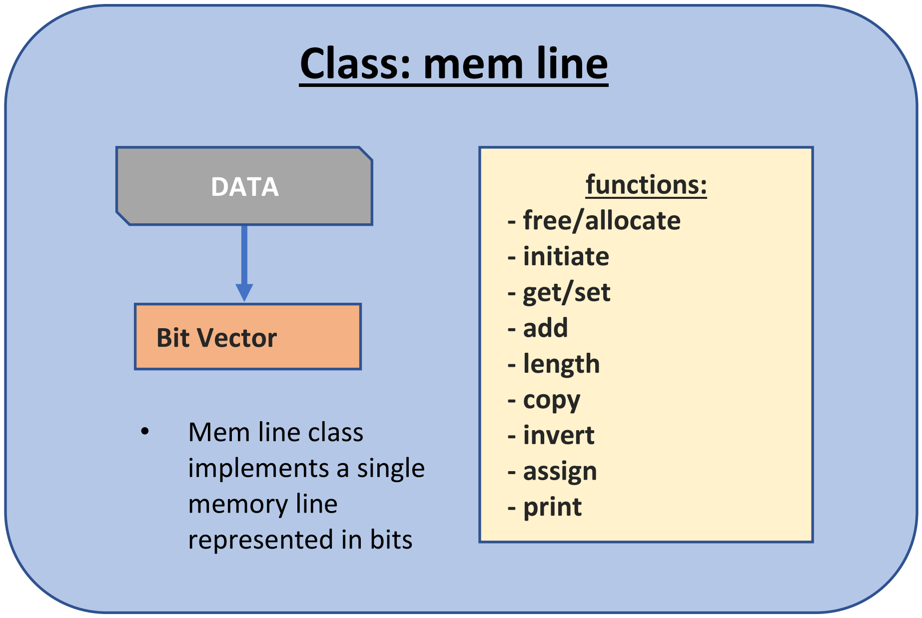
The hierarchy of simulation can be shown in the diagrams below.

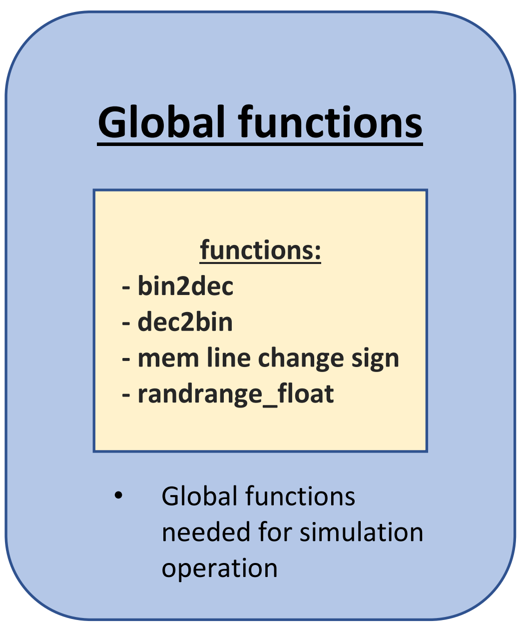
2.2. diagrams




Class: MAGIC
Memristive memory
Mem_line(1)
Mem_line(2)
Mem_line(n-1)
Mem_line(n)
parameters:
- N_bit
- FIP
counters:
- C_mem
- C_mem_max
- C_mem_avg
- C_cycles
arrays:
- mem_arr
- cycles_arr
simulation functions:
- simulation initiate
- simulation end
- update counters
- print MAGIC
- print counters
- plot graph
- write/read number
functional functions:
- logic functions
- arithmetic function
• MAGIC class implement the MAGIC memory itself. Uses real numbers only


Class: COMPLEX MAGIC
Extends class MAGIC
functional functions:
- write complex number
overrides:
- read number
- Full Adder
- multiplication
- division
• COMPLX MAGIC class extends MAGIC class for implementing complex numbers use within memory


Class: mem line
DATA
Bit Vector
Mem line class implements a single memory line represented in bits
functions:
- free/allocate
- initiate
- get/set
- add
- length
- copy
- invert
- assign
- print


Global functions
functions:
- bin2dec
- dec2bin
- mem line change sign
- randrange_float
Global functions needed for simulation operation

2.3. Assumptions

To verify the correctness of the algorithms some assumptions had to be made:

- 2.3.1. infinite memory size
- 2.3.2. 2's complement, fixed point number representation
- 2.3.3. constant number of bits representing numbers (N – number of bit, p – working resolution)
- 2.3.4. inputs/outputs aligned to the left
- 2.3.5. reading/writing from/to memory is done in zero time
- 2.3.6. constant '1' and '0' are stored in known addresses at each line
- 2.3.7. all memristor lines are the same length
- 2.3.8. only a single functional operation can be processed in an individual simulation iteration
- 2.3.9. each line can execute a single operation in a single cycle
- 2.3.10. same operations can be done on different memory lines in parallel only if done in same columns

2.4. Simulation inputs

See detailed parameters in section 4.1

- 2.4.1. parameters for controller
- 2.4.2. parameters for simulation
- 2.4.3. arguments for desired computation
- 2.4.4. what function to run

2.5. Simulation outputs

- 2.5.1. computation result
- 2.5.2. latency in NOC (number of cycles)
- 2.5.3. max area use (number of max memristors used in single cycle)
- 2.5.4. average area use (for optimization during algorithm run)
- 2.5.5. graph: memristor used as function of time (optional)

3. Functional lists

3.1. User function supported

These functions can be operated from outside, as if a controller is using them

3.1.1. complex/real write number to memory
3.1.2. complex/real read number from memory
3.1.3. complex/real numbers full-adder
3.1.4. complex/real numbers fixed point multiplication
3.1.5. complex/real numbers fixed point division with remain
3.1.6. complex/real numbers fixed point division with approximation (no remain)
3.1.7. logic functions (NOR, NOT, OR, AND, XOR)
3.1.8. dec2bin
3.1.9. bin2dec
3.1.10. randRange_float
3.1.11. real numbers absolute value
3.1.12. numbers inversion according to sign (‘1’ or ‘0’)

3.2. Shortcuts list

3.2.1. NOC – number of cycles
3.2.2. FIP – fixed point
3.2.3. FA – full-adder
3.2.4. MUL – multiplication
3.2.5. DIV – division
3.2.6. Cmplx – complex

3.3. Files list

File name	Content and description
MAGIC	The class MAGIC which implements the memory itself, implemented as a list of mem_lines Containing all its user and inner function along with variables and parameters In charge of the simulation counters for memory and latency use Working with real numbers only
MAGIC_CMPLX	Implements the class of complex Memristive memory by extending MAGIC class and overriding needed functions.
Mem_line	A class implementing data structure for a single memory line. Hides low level dealing with the bit vectors arrays. Uses bitVector library
Functions	Holds global functions needed to run the simulation
Complexity_param	Holds lambda function of the latency and memristors use complexity of different function used in simulation without implementation. For example – full-adder is implemented in simulation and counters are updated according to given FA

	parameters from this file. Any change in parameters should be done from this file
Running_2num_logic_func	A simulation run of all logic functions implemented in MAGIC
Running_abs_func	A simulation run of absolute value function implemented in MAGIC
Running_ADD1bit_func	A simulation run of ADD1 function implemented in MAGIC. The function adds a given bit to a N bit number using HA instead of FA and therefore save latency (not saving memristors)
Running_CMPLX_DIV_remain	A simulation run of complex division function with remain implemented in CMPLX_MAGIC
Running_CMPLX_DIV_approx	A simulation run of complex division function with approximation implemented in CMPLX_MAGIC
Running_CMPLX_FA_func	A simulation run of complex full-adder function implemented in CMPLX_MAGIC
Running_CMPLX_MUL	A simulation run of complex multiplication function implemented in CMPLX_MAGIC
Running_CMPLX_N_simulation	A simulation run of selected complex function with different N bits representation to simulate selected function dependency of N
Running_CMPLX_P_simulation	A simulation run of selected complex function with different P (fixed point location) representation to simulate selected function dependency of P
Running_DIV_remain_func	A simulation run of real numbers division function with strict number representation and resolution implemented in MAGIC The function output is with result and remain. Having the result and remain together, the output is accurate.
Running_DIV_approx_func	A simulation run of real numbers division function with flexible result representation implemented in MAGIC. Result will be less accurate according to given N & p values given by user.
Running_FA_func	A simulation run of real numbers full-adder function implemented in MAGIC
Running_invert_according2sign	A simulation run of invert according2sign function implemented in MAGIC. The function will invert a given number according to a given sign (‘0’ or ‘1’) using 2’s complement method
Running_MUL_func	A simulation run of real numbers multiplication function implemented in MAGIC
Running_N_simulation	A simulation run of selected function with different N bits representation to simulate selected function dependency of N
Running_P_simulation	A simulation run of selected function with different P (fixed point location) representation to simulate selected function dependency of P
Running_write_read_func	A simulation run of real numbers read and write functions implemented in MAGIC

4. Detailed information

4.1. Parameters table

#	Name	Description
1	FIP	Fixed point location – number of bits to the right of decimal point
2	N_bit	Number of bits representing each number (include all)
3	Mem_FA(N)	Full adder area efficiency - a lambda function depended on N
4	Cycles_FA(N)	Full adder latency (in NOC) - a lambda function depended on N
5	Mem_MUL(N)	Multiplication area efficiency- a lambda function depended on N
6	Cycles_MUL(N)	Multiplication latency (in NOC) - a lambda function depended on N
7	Mem_ADD1bit(N)	Add1 function area efficiency- a lambda function depended on N
8	Cycles_ADD1bit(N)	Add1 function latency (in NOC) - a lambda function depended on N

4.2. Counters table

Each operation updates the counters according to its use

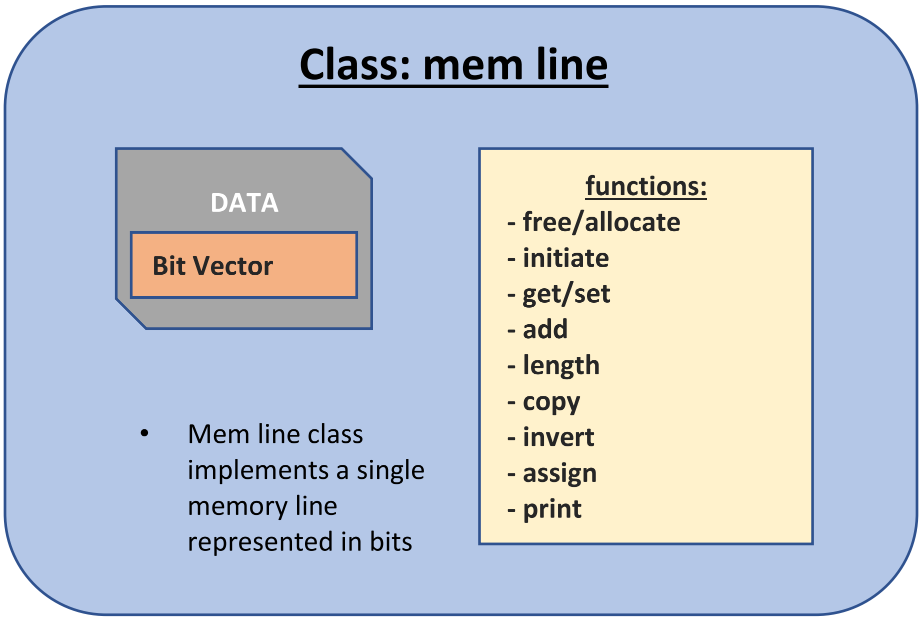
#	Name	Description
1	C_cycles	Cycles counter
2	C_mem	Memristor used counter in single line (assuming all line have the same length)
3	C_mem_max	Maximum memristors used in a single line
4	C_mem_avg	Weighted average memristors use in a single line. Weight is according to number of cycles the memristors were used
5	Mem_arr	Array holding the memristor use. Each cell [i] holds memristor use in time cycles_arr[i]
6	Cycles_arr	Array holding the number of cycles at every start/end of operation. The array is being built during the progress of the simulation such that it is being continuously updated.

4.3. Classes brake down

4.3.1. Mem_line

4.3.1.1. Description

This class is designed to function as a single memory line with the capability of allocating, freeing, reading and writing to this memory line. The class uses and hides a python implemented data structure called BitVector. This class is designed to ease the work with bits in MAGIC simulation.



4.3.1.2. Functions

1. __init__

1.1. Inputs

N(optional) – number of bits representation of the number to initiate to. Default is n=0

1.2. Outputs

non

1.3. Description

Constructor of mem_line object. Initiate a BitVector of size N in its data field

Private function

2. __getitem__

2.1. Inputs

i – index or a slice (start & end index) of the wanted bit/bits.

2.2. Outputs

The value of mem_line[i] or -1 if index/slice is out of bound
If i stands for a slice – the return value is a mem_line object containing only the slice

2.3. Description

Getting an indexed value out of mem_line object easily with “[i]” or return -1 when index is out of bound.

Private function

3. __setitem__

3.1. Inputs

i – index of where to set wanted bit.
value – a bit value to assign – ‘0’ or ‘1’

3.2. Outputs

-1 if I out bound
0 if successful

3.3. Description

Sets ‘value’ to index ‘i’ in mem_line

Private function

4. __add__

4.1. Inputs

rhs – a mem_line object

4.2. Outputs

A new Mem_line object containing the data of the caller mem_line followed by the data of rhs

4.3. Description

Adding 2 mem_lines data into a new mem_line and returning the new mem_line. "this" data followed by "rhs" data

Private function

5. __len__

5.1. Inputs

Non

5.2. Outputs

The length of data - integer

5.3. Description

Returning the length of mem_line data of caller object

6. Deep_copy

6.1. Inputs

Non

6.2. Outputs

A new mem_line object – copy of the caller mem_line object

6.3. Description

Copying mem_line object into a new allocated mem_line

7. Invert

7.1. Inputs

i – start index where to start inversion from

n – length of how many bits from i to invert

7.2. Outputs

A new mem_line – the inversion of caller mem_line[i,i+n-1]

7.3. Description

Inverting bits i to i+n-1 of calling object and returning a copy of the result

8. assign

8.1. Inputs

i – start index where to start assign from
value – the bit value to assign – ‘0’ or ‘1’

8.2. Outputs

-1 if fail duo to i out of bound
0 is all successful

8.3. Description

A user function hiding __setitem__ and assigning a bit value to mem_line

9. allocate

9.1. Inputs

N(optional) – length of how many bits from to allocate. Default is 1

9.2. Outputs

non

9.3. Description

Allocating new n bits to mem_line at the end of the existing line

10. free

10.1. Inputs

n – length of how many bits to free

10.2. Outputs

non

10.3. Description

Freeing n bits from mem_line at the end of the existing line

11. Print_mem

11.1. Inputs

non

11.2. Outputs

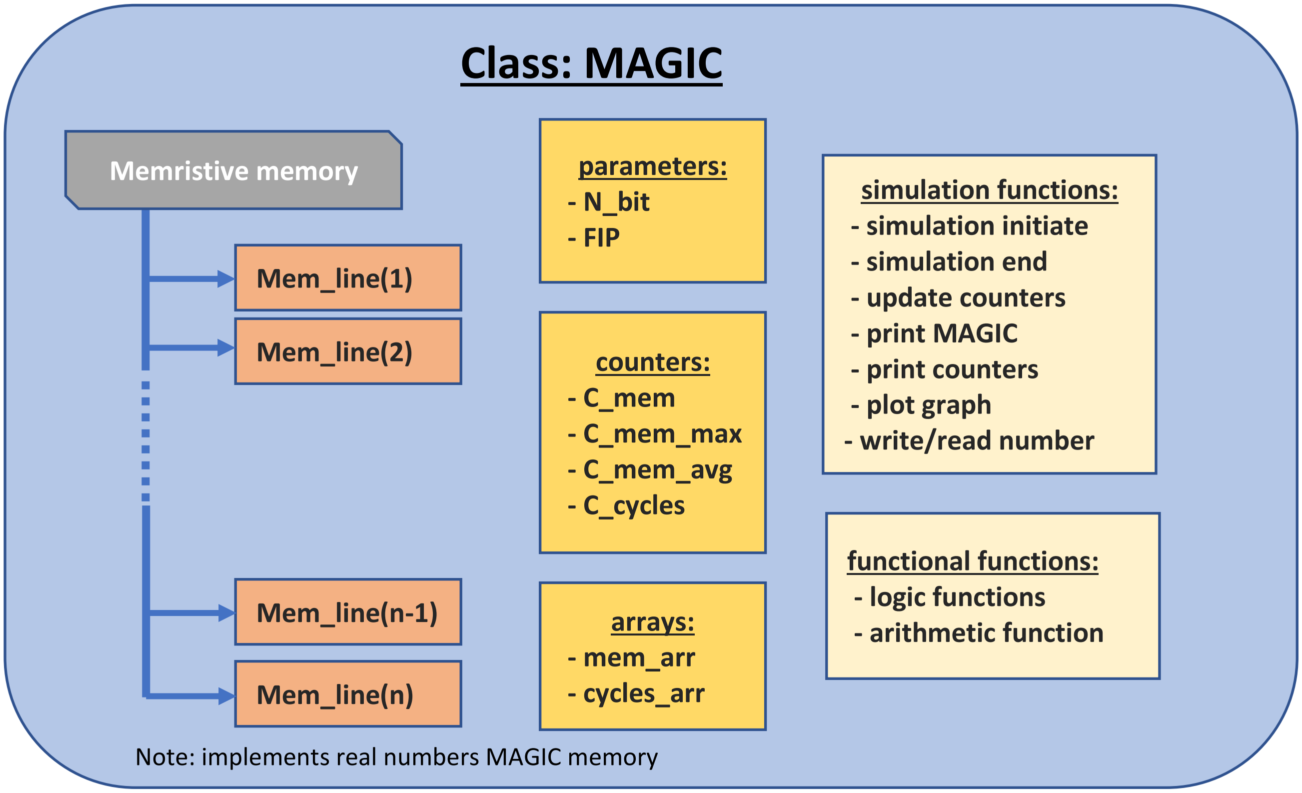
non

11.3. Description

Printing existing mem_line data

4.3.2. MAGIC

4.3.2.1. Description



MAGIC class has a data structure of a list of mem_lines, each functions as a different row (address) in memory. The class is designed to simulate a partial controller executing operations on Memristive memory. It does so by executing logical and arithmetical operations in an address-based functions. For every execution the class updates its counters according to known parameters of the execution. Execution of high-level function is done by imitating low level operations on the given inputs. Thus, by updating simulation time and memristor use, the simulation can output the latency and area needed for the high-level application execution.

MAGIC and counters are global and visible to all class' objects.

4.3.2.2. Simulation functions

1. F_simulation_init

11.4. Inputs

N_bit – number of bits representing the number
FIP – decimal fixed-point location. The number of bits to the right of decimal point.

11.5. Outputs

non

11.6. Description

Initializing important parameters for the simulation. These parameters should be known to memory controller.

Initializing all counters to 0.

Initializing memory to an empty list.

2. F_simulation_end

2.1. Inputs

non

2.2. Outputs

C_mem_max – maximum memristor use in a single line during the simulation
C_mem_avg – average memristor use in a single line during the simulation
C_cycles – number of cycles took to run the simulation

2.3. Description

Printing memory content, plots the graph of the simulation and returning the output values

3. F_mem_graph

3.1. Inputs

non

3.2. Outputs

non

3.3. Description

Plotting memristors continues use, the maximum value and the average value as function of cycles

4. F_print

4.1. Inputs

non

4.2. Outputs

non

4.3. Description

Printing memory content line after line in bits

5. F_printCounters

5.1. Inputs

non

5.2. Outputs

non

5.3. Description

Printing counters values and arrays values

6. F_update_counters

6.1. Inputs

non

6.2. Outputs

non

6.3. Description

Updating all counters:

Updates C_mem according to longest memory line.

Updates arrays by adding current C_mem and C_cycles values to a new cell in existing compatible arrays.

Updates C_mem_avg by calculating new weighted average.

7. F_write_num

7.1. Inputs

Row_address – row address of the number to write into memory
Msb_address – col number of the msb of the number to write into memory
Value – decimal value of the number to write into memory
N (optional) – number of bits representation to write in, positive numbers only. Default value is N_bit of the simulation.
P (optional) – decimal point location of written number, positive numbers only. Default value is FIP of the simulation.

7.2. Outputs

non

7.3. Description

Writing the given number into its memory address.

Writing takes zero time.

Uses dec2bin function, then writing the number bit by bit using __F_write_bit

Writing is always possible. If address does not exist, allocates new memory lines/memristors up to given address.

8. __F_write_bit

8.1. Inputs

Row_address	– row address of the bit to write into memory
bit_address	– col number of the bit to write into memory
Value	– bit value of the number to write into memory

8.2. Outputs

non

8.3. Description

Writing the given bit into its memory address.

Writing takes zero time.

Uses assign function of mem_line.

Writing is always possible. If address does not exist, allocates new memory lines/memristors up to given address.

9. F_read_num

9.1. Inputs

Row_address	– row address of the number to read from memory
Msb_address	– col number of the msb of the number to read from memory
N (optional)	– number of bits to read, positive numbers only. Default value is N_bit of the simulation.
P (optional)	– decimal point location of the number to read, positive numbers only. Default value is FIP of the simulation.

9.2. Outputs

Value	– decimal value of the read number or ERROR throw if reading outside memory

9.3. Description

Reading number from given memory address.

Throws error assertion if reading out of memory

10. F_row_out_of_bound

10.1. Inputs

Row_address	– row address to check if in bound

10.2. Outputs

'1'	– row address out of memory
'0'	– row within memory

10.3. Description

Checks given row to be within memory

11. F_mem_out_of_bound

11.1. Inputs

Row_address – row address to check

Mem_add – memristor address to check if inside the memory given row

11.2. Outputs

‘1’ – mem address out of memory row

‘0’ – mem within memory row

11.3. Description

Checks given memristor address to be within given row in memory

4.2.2.3. Logical functions

1. F_2num_NOR
    1.1. Inputs

    Rows – row addresses to operate on (array or single address)
    In1_msb – in1 number msb address to execute NOR on
    In2_msb – in2 number msb address to execute NOR on
    out_add – output msb address to write result of NOR operation to
    N (optional) – number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

    1.2. Outputs

    non

    1.3. Description

    Executes bitwise NOR on 2 given inputs, writing result to given address and updates counters accordingly.

    Can fail if input address out of bound (throws assertion)

    Need $1 \times n$ cycles and $0 \times n$ intermediate memristors to execute.

    Does not initiate any memristors – initiation I from high level function.

2. F_2num_OR
    2.1. Inputs

    Rows – row addresses to operate on (array or single address)
    In1_msb – in1 number msb address to execute OR on
    In2_msb – in2 number msb address to execute OR on
    out_add – output msb address to write result of OR operation to
    N (optional) – number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

    2.2. Outputs

    non

    2.3. Description

    Executes bitwise OR on 2 given inputs, writing result to given address and updates counters accordingly.

    Can fail if input address out of bound (throws assertion)

    Need $2 \times n$ cycles and $1 \times n$ intermediate memristors to execute.

    Does not initiate any memristors – initiation I from high level function.

3. F_2num_XOR
    3.1. Inputs

    Rows – row addresses to operate on (array or single address)

In1_msb – in1 number msb address to execute XOR on
In2_msb – in2 number msb address to execute XOR on
out_add – output msb address to write result of XOR operation to
N (optional) – number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

3.2. Outputs

non

3.3. Description

Executes bitwise XOR on 2 given inputs, writing result to given address and updates counters accordingly.

Can fail if input address out of bound (throws assertion)

Need $5 \times n$ cycles and $4 \times n$ intermediate memristors to execute.

Does not initiate any memristors – initiation I from high level function.

4. F_2num_AND

4.1. Inputs

Rows – row addresses to operate on (array or single address)
In1_msb – in1 number msb address to execute AND on
In2_msb – in2 number msb address to execute AND on
out_add – output msb address to write result of AND operation to
N (optional) – number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

4.2. Outputs

non

4.3. Description

Executes bitwise AND on 2 given inputs, writing result to given address and updates counters accordingly.

Can fail if input address out of bound (throws assertion)

Need $3 \times n$ cycles and $2 \times n$ intermediate memristors to execute.

Does not initiate any memristors – initiation I from high level function.

5. F_1num_NOT

5.1. Inputs

Rows – row addresses to operate on (array or single address)
In_msb – in number msb address to execute NOT on
out_add – output msb address to write result of NOT operation to
N (optional) – number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

5.2. Outputs

non

5.3. Description

Executes bitwise NOT on the given input, writing result to given address and updates counters accordingly.

Can fail if input address out of bound (throws assertion)

Need $1 \times n$ cycles and $0 \times n$ intermediate memristors to execute (same as NOR only on single num)

Does not initiate any memristors – initiation I from high level function.

4.2.2.4. Arithmetical functions

1. F_2num_FA

1.1. Inputs

Rows	– row addresses to operate on (array or single address)
In1_msb	– in1 number msb address for full-adder
In2_msb	– in2 number msb address for full-adder
out_add	– output msb address to write result of full-adder operation to
carry_in(optional)	– carry in to add to FA. Default is ‘0’. Can be ‘0’ or ‘1’.
carry_out_add(optional)	– address to write carry out to. Default is to not to write it.
N (optional)	– number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

1.2. Outputs

non

1.3. Description

Executes full-adder operation on 2 given inputs, writing result to given address and updates counters accordingly. The function uses F_2bit_FA to execute operation on all lines & bits.

Can fail if input address out of bound (throws assertion)

Number of cycles and memristor needed for the operation is according to cycles_FC(N) & mem_FA(N)

2. F_2bit_FA

2.1. Inputs

Row	– single row addresses to operate on
In1_add	– in1 bit address to execute bit FA on
In2_add	– in2 bit address to execute bit FA on
carry_in(optional)	– carry in to add to FA. Default is ‘0’. Can be ‘0’ or ‘1’.

2.2. Outputs

carry_out	– carry out result. Can be ‘0’ or ‘1’.
Result	– FA bit wise result. Can be ‘0’ or ‘1’

2.3. Description

Executes full-adder operation on 2 given bits. Returns the result and carry values

Can fail if input address out of bound (throws assertion)

Does not update any counters.

3. F_ADD1bit

3.1. Inputs

Rows	– row addresses to operate on (array or single address)
In1_msb	– in1 number msb address for add1bit
Bit_in2	– a given bit to add to in1 number with add1bit

out_add – output msb address to write result of add1bit operation to
carry_out_add(optional) – address to write carry out to. Default is to not to write it.
N (optional) – number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

3.2. Outputs

non

3.3. Description

Executes addition operation between a single bit and a whole n-bits, writing result to given address and updates counters accordingly. The function uses F_2bit_HA to execute operation on all lines & bits.

Can fail if input address out of bound (throws assertion)

Number of cycles and memristor needed for the operation is according to cycles_ADD1BIT(N) & mem_ADD1BIT(N)

4. F_2bit_HA

4.1. Inputs

Row – single row addresses to operate on
In1_add – in1 bit address to execute bit HA on
In2 – in2 bit to execute bit HA on – can be the bit value or its address in line.
value_in(optional) – a flag indicating if in2 is the bot value or its address. Default is '0'. Can be '0' or '1'. '1' stands for value – '0' stands for address

4.2. Outputs

carry_out – carry out result. Can be '0' or '1'.
Result – HA bit wise result. Can be '0' or '1'

4.3. Description

Executes half-adder operation on 2 given bits. Can execute HA with in2 as abit value or as an address. Value_in input indicates which are we using. Returns the result and carry values

Can fail if input address out of bound (throws assertion)

Does not update any counters.

5. F_abs

5.1. Inputs

Rows – row addresses to operate on (array or single address)
In_msb – input number msb address to calculate its absolute value
out_add – output msb address to write result of the operation to
N (optional) – number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

5.2. Outputs

non

5.3. Description

Executes abstract-value operation on the given input on all given rows. Uses F_invert_according2sign to do so.

Does not update any counters.

6. F_invert_according2sign

6.1. Inputs

Rows	– row addresses to operate on (array or single address)
Sign_add	– the address of the sign bit to invert the input number according to
In_msb	– input number msb address to invert
out_add	– output msb address to write result of inversion to
N (optional)	– number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

6.2. Outputs

non

6.3. Description

Executes value inversion of a given number according to another given bit. It does so by operation bitwise XOR on the given number with the sign bit, then adding the sign bit to the result. This operation will do nothing if sign bit ='0', but it will invert the number according to 2's complement representation if sign bit ='1'

Initiate memristors used. Uses f_2num_XOR with n=1, and f_ADD1BIT. (also uses NOT for re-writing sign bit into a full-length number for FA operation)

7. F_2num_MUL

7.1. Inputs

Rows	– row addresses to operate on (array or single address)
In1_msb	– in1 number msb address for multiplication
In2_msb	– in2 number msb address for multiplication
out_add	– output msb address to write result of multiplication operation
N (optional)	– number of bits representing the inputs numbers, positive numbers only. Default value is N_bit of the simulation.
Res_n (optional)	– number of bits representing the result number, positive numbers only. Default value is N_bit of the simulation.
p (optional)	– decimal point location of input numbers, positive numbers only. Default value is FIP of the simulation.
Res_p (optional)	– decimal point location of result numbers, positive numbers only. Default value is FIP of the simulation.

7.2. Outputs

non

7.3. Description

Executes multiplication operation on 2 given inputs, writing result to given address and updates counters accordingly.

Can fail if input address out of bound (throws assertion)

Number of cycles and memristor needed for the operation is according to MUL_C(N) & MUL_A(N)

8. F_2num_DIV_remain

8.1. Inputs

Rows	– row addresses to operate on (array or single address)
Numerator_add	– msb address of numerator number
Divisor_add	– msb address of divisor number
result_add	– msb address to write result of division to
remain_add	– msb address to write the remain from division operation to

8.2. Outputs

non

8.3. Description

Dividing 2 numbers of size N bits: $\frac{numerator}{divisor}$. The result will be written to the given address.
Result will always be a N bits number with p=0.
Remain will be written to the given remain address.

$$result = sign(num) \times sign(div) \times \left\lfloor \left| \frac{numerator}{divisor} \right| \right\rfloor$$

(the lower whole value of the absolute numbers division, with correct sign)

We receive an exact result in the following manner:

$$numerator = divisor \times result + reamin$$

Division is done in 3 main stages:
1. initiation
2. iterations
3. epilogue

Initiation:

- Allocation of memristors needed for calculations and results
- Calculating needed arguments for next stages:
  - Result & remain signs
  - Absolut values of numerator & divisor
  - NOT(divisor)

Iterations:

- Iterates over the numerator like in long division taught in school, reducing divisor from numerator starting from MSB to LSB

- Note that this is done on positive numbers only

Epilogue:

- Adjusting result and remain signs according to previous calculations
- Writing the results into their place
- Freeing unnecessary previously allocated memory

9. F_2num_DIV_approx

9.1. Inputs

Rows	– row addresses to operate on (array or single address)
Numerator_add	– msb address of numerator number
Divisor_add	– msb address of divisor number
result_add	– msb address to write result of division to
N (optional)	– number of bits representing for result number, positive numbers only. Default value is N_bit of the simulation.
p (optional)	– decimal point location for result number, positive numbers only. Default value is FIP of the simulation.

9.2. Outputs

non

9.3. Description

Dividing 2 numbers of size N bits: $\frac{numerator}{divisor}$. The result will be written to the given address.
Result's representation parameters (N,p) will be given by user (or default) to the given address

In this function the result will not necessarily be accurate. Accuracy depends on N & p.
To prevent overflow $N_{res} \geq N_{simulation} + p_{simulation}$
The result will be accurate up to $2^{-p_{res}}$

Division is done in 3 main stages:
1. initiation
2. iterations
3. epilogue

Initiation:

- Allocation of memristors needed for calculations and results
- Calculating needed arguments for next stages:
  - Result & remain signs
  - Absolut values of numerator & divisor
  - NOT(divisor)

Iterations:

- Iterates over the numerator like in long division taught in school, reducing divisor from numerator starting from MSB to LSB
- Note that this is done on positive numbers only
- Iterations continue for p_{res} iterations in order to get the wanted result resolution

Epilogue:

- Adjusting result sign according to previous calculations
- Writing the result into its place
- Freeing unnecessary previously allocated memory

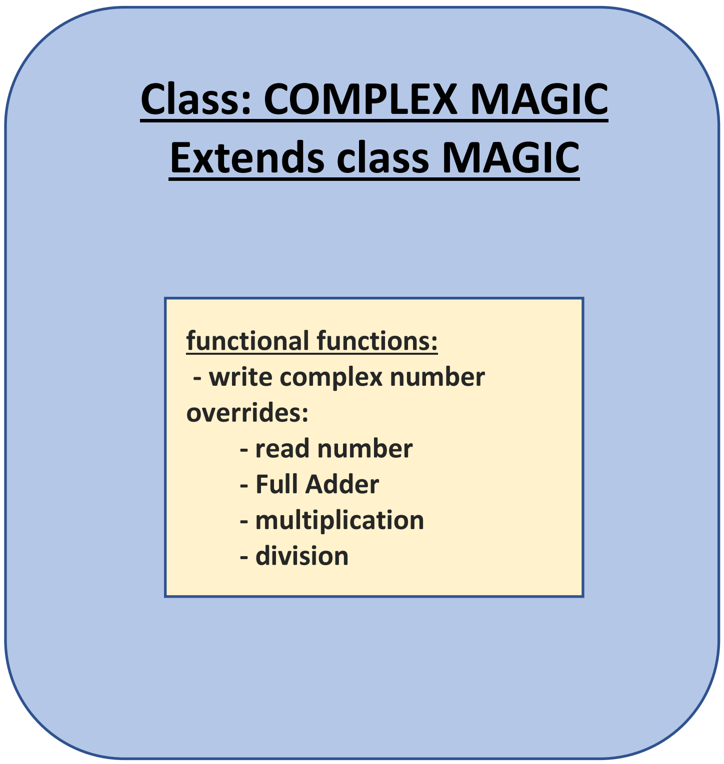
4.2.3. CMPLX_MAGIC

4.2.3.1. Description

CMPLX_MAGIC class is designed for simulating complex numbers calculations within memory and is doing so by extending MAGIC class.

Complex numbers represented in $z = a + jb$ representation where $a = \mathcal{R}eal(z)$ $b = \mathcal{I}m(z)$ and both has the same N & p parameters.

The class implements '*write number*' function and overrides read, FA, MUL & DIV function of MAGIC.



4.2.3.2. Functions

1. F_write_CMPLX_num

1.1. Inputs

Row_address	– row address of the number to write into memory
Msb_address	– col number of the msb of the number to write into memory
Cmplx_num[2]	– 2 cells array of the decimal value of the number to write into memory. $cmplx\_num[0] = \mathcal{Real}(z), cmplx\_num[1] = \mathcal{Im}(z)$
N (optional)	– number of bits representation to write in, positive numbers only. Default value is N_bit of the simulation.
P (optional)	– decimal point location of written number, positive numbers only. Default value is FIP of the simulation.

1.2. Outputs

non

1.3. Description

Writing the given complex number into its memory address.

Writing takes zero time.

Uses MAGIC F_write_num function to write each part of the complex number. Real part followed by imaginary.

Writing is always possible. If address does not exist, allocates new memory lines/memristors up to given address.

2. F_read_num

2.1. Inputs

Row_address	– row address of the number to read from memory
Msb_address	– col number of the msb of the number to read from memory
N (optional)	– number of bits to read, positive numbers only. Default value is N_bit of the simulation.
P (optional)	– decimal point location of the number to read, positive numbers only. Default value is FIP of the simulation.

2.2. Outputs

Cmplx_num	– 2 cells array of decimal value of the read number, $cmplx\_num[0] = \mathcal{Real}(z), cmplx\_num[1] = \mathcal{Im}(z)$ or ERROR throw if reading outside memory

2.3. Description

Overrides F_read_num of MAGIC class

Reading complex number from given memory address. By using MAGIC F_read_num

Throws error assertion if reading out of memory

3. F_2num_FA

3.1. Inputs

Rows	– row addresses to operate on (array or single address)
In1_msb	– in1 number msb address for full-adder
In2_msb	– in2 number msb address for full-adder
out_add	– output msb address to write result of full-adder operation to
carry_in[2](optional)	– 2 cells array of carry in to add to FA. First for real part FA, second for imaginary part FA. Default is ‘0,0’. Each can be ‘0’ or ‘1’.
carry_out_add(optional)	– address to write carry out to. Default is to not to write it.
N (optional)	– number of bits representing the in & out numbers, positive numbers only. Default value is N_bit of the simulation.

3.2. Outputs

non

3.3. Description

Overrides F_2num_FA of class MAGIC.

Executes full-adder operation on 2 given complex inputs, writing result to given address. The function uses F_2num_FA to execute operation.

Execution is done in the following manner:

$$\mathcal{R}eal(result) = \mathcal{R}eal(in1) + \mathcal{R}eal(in2)$$

$$\mathcal{I}m(result) = \mathcal{I}m(in1) + \mathcal{I}m(in2)$$

Can fail if input address out of bound (throws assertion)

4. F_2num_MUL

4.1. Inputs

Rows	– row addresses to operate on (array or single address)
In1_msb	– in1 number msb address for multiplication
In2_msb	– in2 number msb address for multiplication
out_add	– output msb address to write result of multiplication operation
N (optional)	– number of bits representing the inputs numbers (each part of the complex number), positive numbers only. Default value is N_bit of the simulation.
Res_n (optional)	– number of bits representing the result number (each part of the complex number), positive numbers only. Default value is N_bit of the simulation.
p (optional)	– decimal point location of input numbers, positive numbers only. Default value is FIP of the simulation.
Res_p (optional)	– decimal point location of result numbers, positive numbers only. Default value is FIP of the simulation.

4.2. Outputs

non

4.3. Description

Overrides F_2num_MUL of class MAGIC.

Executes multiplication operation on 2 given complex inputs, writing result to given address. Uses F_2num_MUL of MAGIC to do so.

Execution is done in the following manner:

$$\mathcal{R}eal(result) = \mathcal{R}eal(in1) \times \mathcal{R}eal(in2) - \mathcal{I}m(in1) \times \mathcal{I}m(in2)$$

$$\mathcal{I}m(result) = \mathcal{R}eal(in1) \times \mathcal{I}m(in2) + \mathcal{I}m(in1) \times \mathcal{R}eal(in2)$$

Real part of result calculated first as follows:

$$\mathcal{R}eal(result) = FA\left[\overline{\mathcal{I}m(in1) \times \mathcal{I}m(in2)} + \mathcal{R}eal(in1) \times \mathcal{R}eal(in2) + carry = 1\right]$$

Can fail if input address out of bound (throws assertion)

5. F_2num_DIV_remain

5.1. Inputs

Rows	– row addresses to operate on (array or single address)
Numerator_add	– msb address of complex numerator number
Divisor_add	– msb address of complex divisor number
result_add	– msb address to write result of division to
remain_add	– msb address to write the remain from division operation to
N (optional)	– number of bits representing the inputs numbers, positive numbers only. Default value is N_bit of the simulation
Res_n (optional)	– number of bits representing the result number, positive numbers only. Default value is N_bit of the simulation
p (optional)	– decimal point location of input numbers, positive numbers only. Default value is FIP of the simulation
Res_p (optional)	– decimal point location of result numbers, positive numbers only. Default value is FIP of the simulation

5.2. Outputs

non

5.3. Description

Executing complex numbers division with remain. The division is done by calculating the conjugate number of divisor, then multiplying both numerator and divisor with it, thus creating a real number in divisor. Then, using real number division, dividing real and imaginary parts of the numerator (after the multiplication) with the new, real divisor, using F_2num_DIV_remain function from MAGIC

$$\frac{z_1}{z_2} = \frac{z_1 \times \overline{z_2}}{z_2 \times \overline{z_2}} = \frac{z_1 \times \overline{z_2}}{\left|\left|z_2\right|\right|^2}$$

6. F_2num_DIV_aprprox

Same as F_2num_DIV_remain, just using MAGIC's F_2num_DIV_approx instead of F_2num_DIV_remain

6.1. Inputs

Rows	– row addresses to operate on (array or single address)
Numerator_add	– msb address of complex numerator number
Divisor_add	– msb address of complex divisor number
result_add	– msb address to write result of division to
remain_add	– msb address to write the remain from division operation to
N (optional)	– number of bits representing the inputs numbers, positive numbers only. Default value is N_bit of the simulation
Res_n (optional)	– number of bits representing the result number, positive numbers only. Default value is N_bit of the simulation
p (optional)	– decimal point location of input numbers, positive numbers only. Default value is FIP of the simulation
Res_p (optional)	– decimal point location of result numbers, positive numbers only. Default value is FIP of the simulation

6.2. Outputs

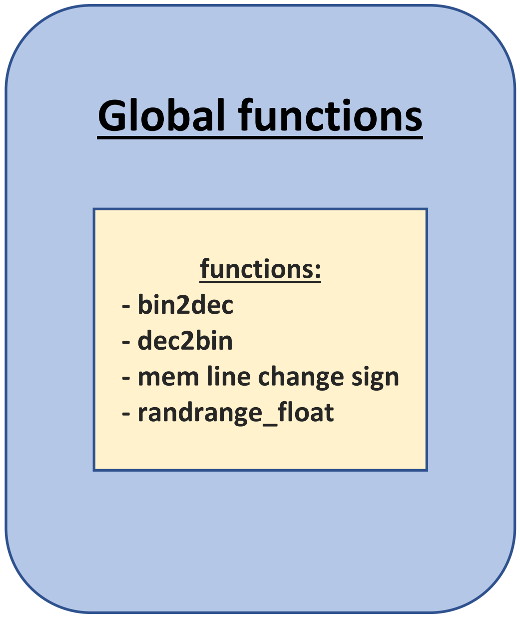
non

6.3. Description

Executing complex numbers division with remain. The division is done by calculating the conjugate number of divisor, then multiplying both numerator and divisor with it, thus creating a real number in divisor. Then, using real number division, dividing real and imaginary parts of the numerator (after the multiplication) with the new, real divisor, using F_2num_DIV_approx function from MAGIC

$$\frac{z_1}{z_2} = \frac{z_1 \times \overline{z_2}}{z_2 \times \overline{z_2}} = \frac{z_1 \times \overline{z_2}}{\left||z_2|\right|^2}$$

4.4. Global functions



1. Bin2dec

1.1. Inputs

P – working resolution of the inverted number

mem – mem_line object of the number to invert to decimal

1.2. Outputs

Result – decimal value of the inverted number

1.3. Description

Inverting binary number saved in a mem_line object to decimal and retuning its value

2. Dec2bin

2.1. Inputs

n – working N bits representation of the inverted number

P – working resolution of the inverted number

num – decimal value of a single number to invert into binary 2’s complement representation

2.2. Outputs

Mem_line – binary value of the given input packed in a mem_line object

2.3. Description

Inverting decimal number to binary, pack it in mem_line object and return the mem_line object

3. Change_sign

3.1. Inputs

mem – mem_line object to change sign to

3.2. Outputs

Mem_line – a copied object with inverted sign of the

3.3. Description

Changing sign in a 2’s complement way of a given mem_line object and returning a copy of it

4. Randrange_float

4.1. Inputs

start – lower bound value of the range in which to rand a value from
end – upper bound value of the range in which to rand a value from
step – step size for legal number resolution in which to randomize the number

4.2. Outputs

Num – the number randomized in between bounds, in the given resolution

4.3. Description

Randomizing numbers with given resolution in given range

4.5. Running functions & files

For acutely simulating all operations on MAGIC complex and regular memory we use running files and function. All running functions and file have the same structure:

1. Randomize different parameters of simulation
2. Initiate several arrays to store inputs and outputs for later comparison of expected vs. real outputs
3. Write randomized, in range values to memory – all aligned to the left and store it also in inputs arrays
4. Execute a single, wanted operation on inputs
5. Read to outputs arrays the result of the operation from memory
6. Calculate expected result according to inputs array using regular, float numbers operation
7. Plot graph and print to screen expected vs. output result graphs

Running files list can be seen in section 3.3

4.6. Using complexity parameters file

In file complexity_param you will find all lambda function for changing high level parameters of simulation for function that are used but not simulated. The functions were implemented in past project/articles and their cost parameters are known for different method of work.

Known parameters were taken from past work and are shown in the table below:

	HA latency	HA memristors used	FA latency	FA memristors used	MUL latency	MUL memristors used
Area optimized	7N	5	15N	5	$13N^2 - 14N + 6$	20N-5
Latency optimized			12N+1	11N-1		
Another known method 1			15N+1	22N-3	Limited precision $6.5N^2 - 7.5N - 2$	Limited precision $19N - 19$
Another known method 1			10N+3	13N-3		

N – number of bits representation

5. Examples

5.1. Running complex FA

Actions needed to run the complex FA simulation:

1. Randomize legal N & p
   1.1. Legal N is a positive integer
   1.2. Legal p is an integer between 0 to N-1
2. Start a new CMPLX_MAGIC object
3. Initiate simulation with N & p from step 1
4. Declare all needed arrays for saving inputs, outputs and expected results
5. Define number of memory lines for this simulation
6. Loop over all lines and write to each line a legal random complex number in place 0
   6.1. Legal numbers randomization is done with randrange_float function
7. Repeat step 6 with writing in msb address aligned to left (msb address = 2N)
8. Define "rows" array to hold all rows acting it the simulation
9. Call object's FA function on all rows, with correct msb addresses of in1, in2 & out = (0,2N,4N)
10. Loop over all rows reading the output values from correct memory address (4N)
    10.1. Push all reading result into predefined arrays (step 4)
    10.2. Read inputs and calculate expected result then push it into predefined arrays (step 4)
11. Print and plot all wanted graphs

Code:

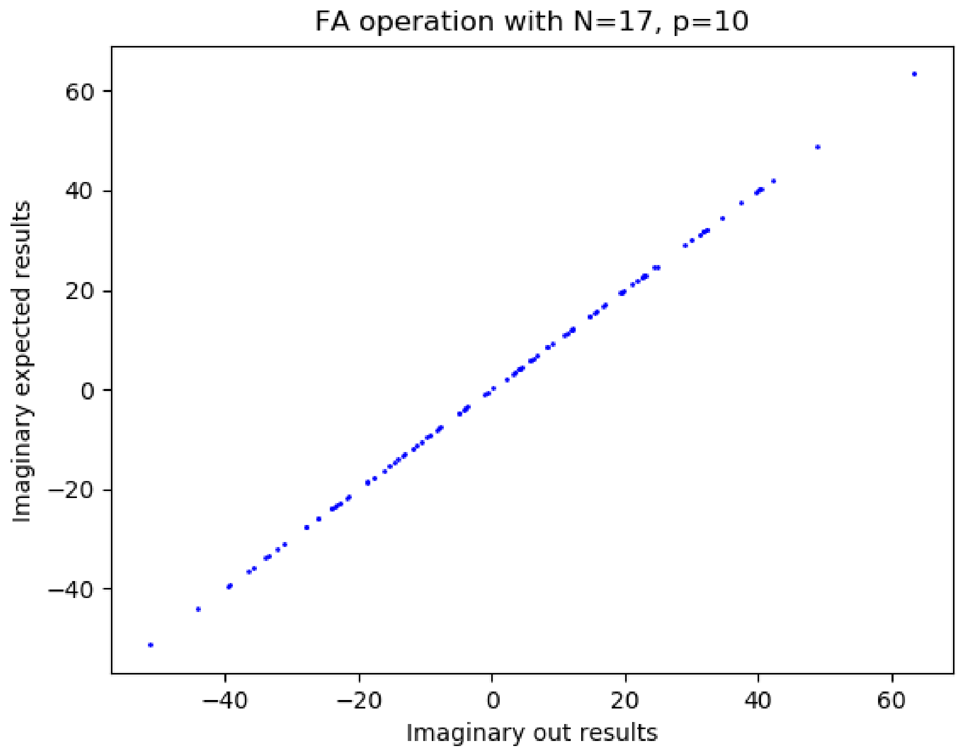
```
####################################################
# running complex FA functions
####################################################

# initating simulation
N = rand.randint(10,20)
p = rand.randint(0,N-1)

myMAGIC = MAGIC_CMPLX()
myMAGIC.F_simulation_init(N,p)
in1_arr_Re = np.array([])
in1_arr_Im = np.array([])
in2_arr_Re = np.array([])
in2_arr_Im = np.array([])
out_arr_Re = np.array([])
out_arr_Im = np.array([])
exp_arr_Re = np.array([])
exp_arr_Im = np.array([])
lines = 1000
```

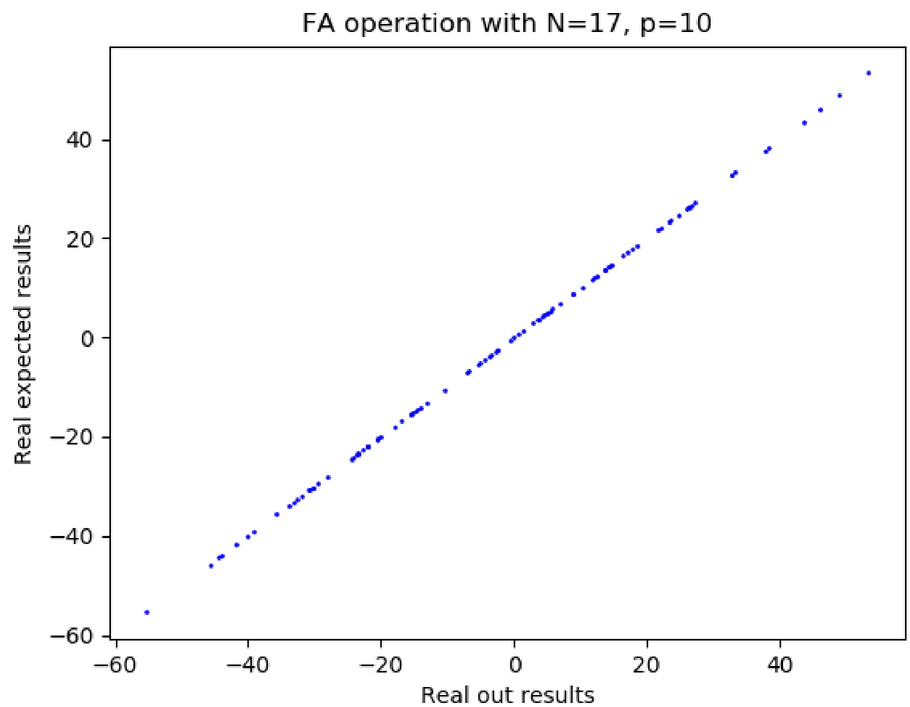
```
# writing some numbers to MAGIC. msb in 0 column
for i in range (0,lines):
    Real = randrange_float(start=-(2 ** (N - p - 2)), stop=(2 ** (N - p - 2)) - (2
** (-p)), step=2 ** (-p))
    Imaginary = randrange_float(start=-(2 ** (N - p - 2)), stop=(2 ** (N - p - 2))
- (2 ** (-p)), step=2 ** (-p))
    in1_arr_Re = np.r_[in1_arr_Re, Real]
    in1_arr_Im = np.r_[in1_arr_Im, Imaginary]
    row_add = i
    msb_add = 0
    myMAGIC.F_write_CMPLX_num(row_add, msb_add, [Real,Imaginary])

# writing second set of numbers to MAGIC. msb in N column
for i in range (0,len(myMAGIC)):
    Real = randrange_float(start=-(2 ** (N - p - 2)), stop=(2 ** (N - p - 2)) - (2
** (-p)), step=2 ** (-p))
    Imaginary = randrange_float(start=-(2 ** (N - p - 2)), stop=(2 ** (N - p - 2))
- (2 ** (-p)), step=2 ** (-p))
    in2_arr_Re = np.r_[in2_arr_Re, Real]
    in2_arr_Im = np.r_[in2_arr_Im, Imaginary]
    row_add = i
    msb_add = 2*N
    myMAGIC.F_write_CMPLX_num(row_add, msb_add, [Real,Imaginary])

# checking full adder on 2 bits with & without carry in
rows = np.arange(0,len(myMAGIC))
myMAGIC.F_2num_FA(rows, in1_msb=0, in2_msb=2*N, out_add=4*N)
for row in rows:
    exp_arr_Re = np.r_[exp_arr_Re, in1_arr_Re[row] + in2_arr_Re[row]]
    exp_arr_Im = np.r_[exp_arr_Im, in1_arr_Im[row] + in2_arr_Im[row]]
    result = myMAGIC.F_read_num(row_address=row, msb_address=4*N)
    out_arr_Re = np.r_[out_arr_Re, result[0]]
    out_arr_Im = np.r_[out_arr_Im, result[1]]
```

```
# ending simulation with prints and graphs
myMAGIC.F_simulation_end()

print "in1_arr_Re = ",in1_arr_Re
print "in1_arr_Im = ",in1_arr_Im
print "in2_arr_Re = ",in2_arr_Re
print "in2_arr_Im = ",in2_arr_Im
print "out_arr_Re = ",out_arr_Re
print "out_arr_Im = ",out_arr_Im
print "exp_arr_Re = ",exp_arr_Re
print "exp_arr_Im = ",exp_arr_Im

plt.figure(2)
plt.plot(out_arr_Re,exp_arr_Re, 'bo', ms=1)
plt.xlabel('Real out results')
plt.ylabel('Real expected results')
plt.title('FA operation with N=%d, p=%d' %(N,p))

plt.figure(3)
plt.plot(out_arr_Im,exp_arr_Im, 'bo', ms=1)
plt.xlabel('Imaginary out results')
plt.ylabel('Imaginary expected results')
plt.title('FA operation with N=%d, p=%d' %(N,p))

plt.show()
```

Output results:

On the two graphs below shown the results of complex FA in comparison to to the expected results:





On the graph below shown the memristor usage during complex FA, working with 17-bit numbers where 10 bits representing the FIP: