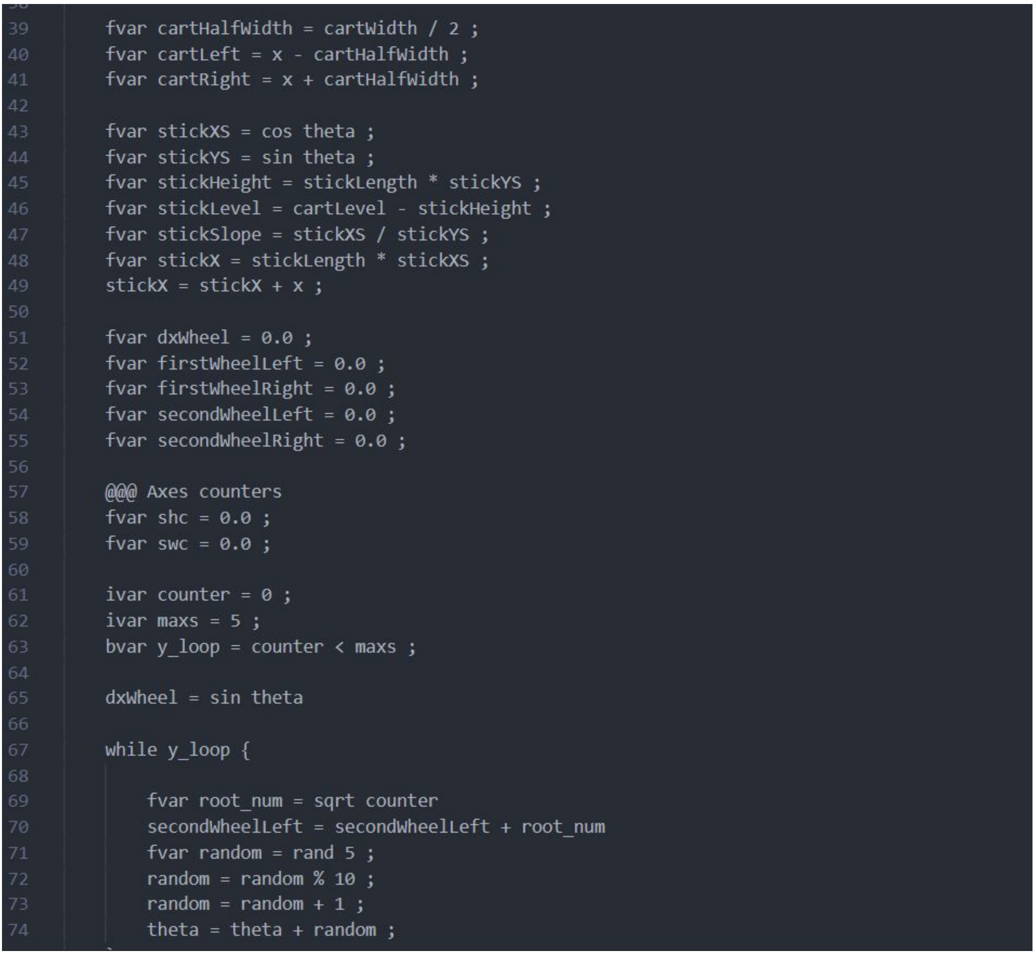
# BEMO Instruction Set

03.April.2020



```
fvar cartHalfWidth = cartWidth / 2 ;
fvar cartLeft = x - cartHalfWidth ;
fvar cartRight = x + cartHalfWidth ;

fvar stickXS = cos theta ;
fvar stickYS = sin theta ;
fvar stickHeight = stickLength * stickYS ;
fvar stickLevel = cartLevel - stickHeight ;
fvar stickSlope = stickXS / stickYS ;
fvar stickX = stickLength * stickXS ;
stickX = stickX + x ;

fvar dxWheel = 0.0 ;
fvar firstWheelLeft = 0.0 ;
fvar firstWheelRight = 0.0 ;
fvar secondWheelLeft = 0.0 ;
fvar secondWheelRight = 0.0 ;

@@@ Axes counters
fvar shc = 0.0 ;
fvar swc = 0.0 ;

ivar counter = 0 ;
ivar maxs = 5 ;
bvar y_loop = counter < maxs ;

dxWheel = sin theta

while y_loop {

    fvar root_num = sqrt counter
    secondWheelLeft = secondWheelLeft + root_num
    fvar random = rand 5 ;
    random = random % 10 ;
    random = random + 1 ;
    theta = theta + random ;
```

Prepared by: Ahmed Salah Eldin Mohamed

# TABLE OF CONTENTS

# OVERVIEW

To be able to control our architecture in an easy way, and to implement the game, and to make the AI, and to control the flow of the program, An instruction set was crucial to perform these tasks. That’s why BEMO has been created.

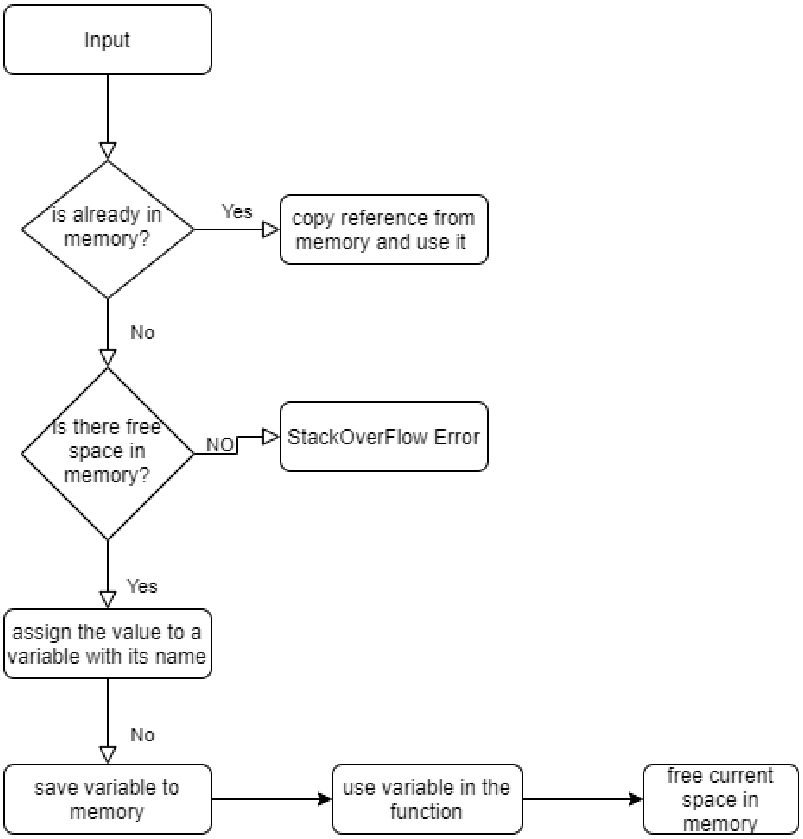
# PARTS AND THEIR FUNCTIONS

BEMO consists of several dependencies (parts) and they are as follows:

## 1. Memory management

The memory management part is very important to manage our limited memory space. The following diagram represents the memory management of the software.

The following diagram shows a flowchart indicating the memory management operation.



Input
is already in memory?
Yes
copy reference from memory and use it
No
Is there free space in memory?
NO
StackOverFlow Error
Yes
assign the value to a variable with its name
No
save variable to memory
use variable in the function
free current space in memory

Click here to see the code of the memory management class.

## 2. Compiler

Compiler reads a text file where the BEMO code is written. The compiler works by reading each line in the text file and storing it inside an array, then the run function loops over each line in the array and compiles it into assembly code detecting the keywords and the commands reserved in BEMO instruction set, and then compiling it into its matching assembly statement. Then the compiler stores the assembly code into another array.

Storing the compiler into another array allows us to best use this array to create notations like while and if, where the compiler will recursively compile the statements in scope until an end condition is met.

The whole Process is repeated until all the lines are compiled. After the whole process is finished, a text file is generated which has the assembly code translated by the compiler.

You can click here to see the code of the compiler.

## 3. Assembler

The Assembler works in a way close to that of the compiler. Commands in the assembly code like **Add $1 $2 $3 ,** or **save $1 $2 $3 ,** or….etc, are saved with it's binary values determined (that is shared by the control unit.)

The Assembler Begins by rewriting the commands and memory locations into another file after being decoded into their binary values using VHDL syntax to be simulated in the Xilinx.

You can click  here to see the code of the assembler.

## 4. Integration

The execution of BEMO is integrating both the compiler and the assembler to generate their assembly - VHDL files respectively. All that is needed for the BEMO file to be compiled is to be cationt to the directions given in the following section to write BEMO, and to be able to run the (run) command in the main file.

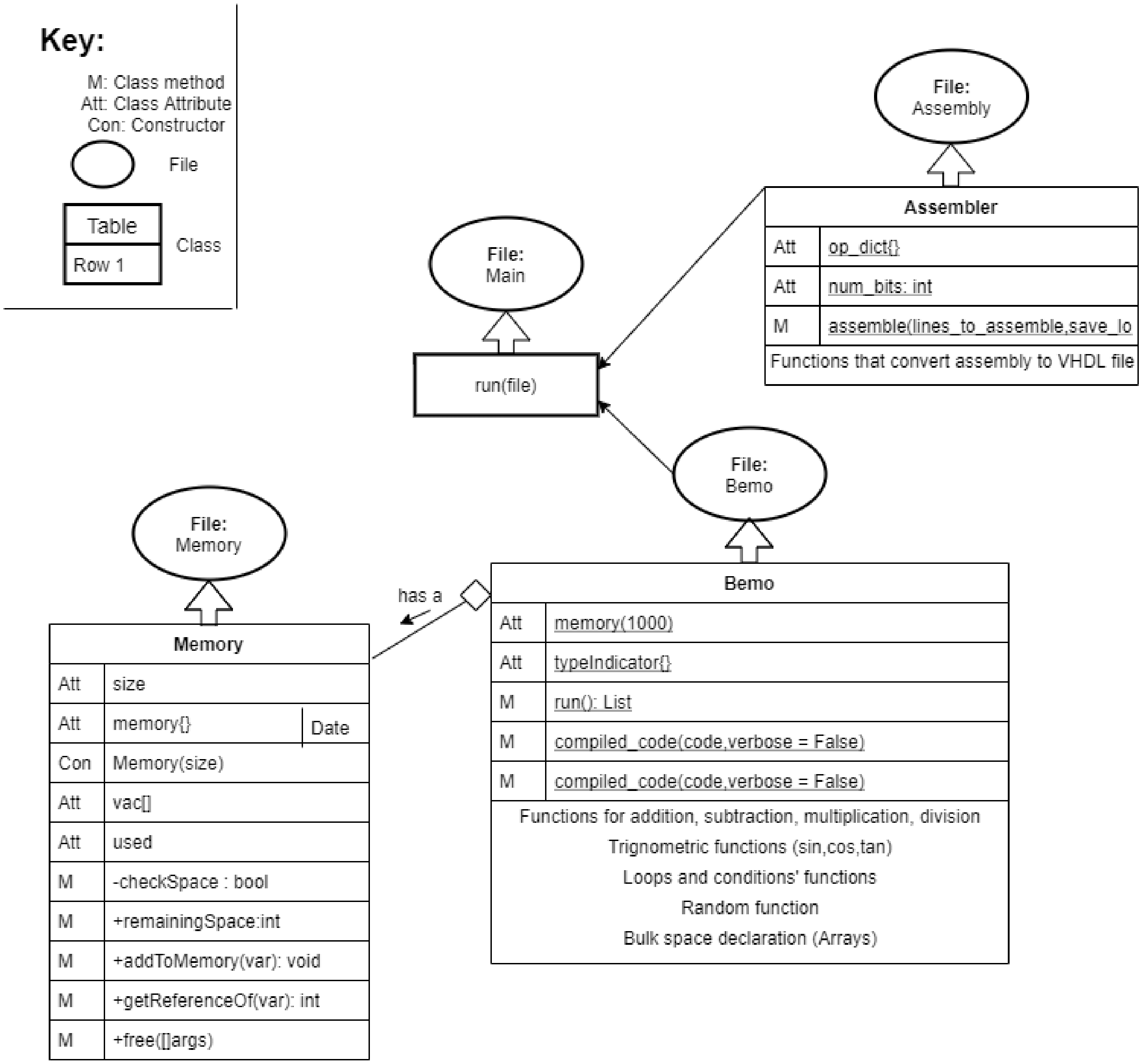
To run the main file, you only need to change the file name below

`if __name__ == "__main__":` to match the file name where BEMO is written.

Then execute the main file to generate 2 files, one containing the assembly code, and the other containing the VHDL code.

You can click here to see the code of the "Main" file.

The following diagram shows the relationship between the entities used in the design of the compiler. and the assembler.


Key:
M: Class method
Att: Class Attribute
Con: Constructor
File
Table
Row 1
Class
File: Assembly
Assembler
Att op_dict{}
Att num_bits: int
M assemble(lines_to_assemble,save_lo
Functions that convert assembly to VHDL file
File: Main
run(file)
File: Bemo
Bemo
Att memory(1000)
Att typeIndicator{}
M run(): List
M compiled_code(code,verbose = False)
M compiled_code(code,verbose = False)
Functions for addition, subtraction, multiplication, division
Trignometric functions (sin,cos,tan)
Loops and conditions' functions
Random function
Bulk space declaration (Arrays)
has a
File: Memory
Memory
Att size
Att memory{}
Date
Con Memory(size)
Att vac[]
Att used
M -checkSpace : bool
M +remainingSpace:int
M +addToMemory(var): void
M +getReferenceOf(var): int
M +free([]args)

# NOTATION

## Important General Notes

1. All statements must be separated with space

2. After each statement, use a semicolon at the end of the line

3. No more than 1 operation can take place at the same line

Example: fvar x = 50/60 ; is permitted but, fvar x = 50*30+60 ; is forbidden.

4. Statements like if and while must have curly braces “{}” to determine its scope.

5. Statements like if and while must have a boolean condition to determine the stop value

Example:

```
ivar start = 0 ;
ivar end = 10 ;
bvar cond = start < end ;
while cond {
show start ;
1
start = start + 1 ;
}
```

6. Code file’s name must follow the following format (Name.txt.bmo)

## Notation

- (ivar) is used to declare an integer variable.

Example: ivar start = 1 ;

- (fvar) is used to declare a float variable.

Example: fvar start = 10.0 ;

- (bvar) is used to declare a boolean variable.

Example: bvar cond = start < end ;

- (@) is used to make a comment

Example: @sdfksmfkl lkef lk

- (a>b) check 'a' is greater than 'b'
- (a<b) check 'a' is less than 'b'
- (a==b) check 'a' equals 'b'
- (intf val) changes an integer value (val) to float
- (fint val) changes a float value (val) to int
- (sin val) calculates the value of sin(x) where (x = val)
- (cos val) calculates the value of cos(x) where (x = val)
- (tan val) calculates the value of tan(x) where (x = val)
- (bvar exp = A and B) Implements the logical AND operation
- (bvar exp = A or B) Implements the logical OR operation
- (bvar exp = not B) Implements the logical NOT operation
- (fvar res = sqrt val) Gets the square root of the value (val) and stores it in res
- (ivar res = absi val) Gets the absolute value of the integer value (val)
- (fvar res = absf val) Gets the absolute value of the float value (val)
- (res = a % b ) Gets 'a' MOD 'b' and stores result in 'res'
- (res = rand seed) generates a random number based on the given seed and stores it in res.

Click here to see a sample file written with BEMO (It is advised to open it using a code editor).

## BEHIND THE CURTAINS

Click here to see the assembly file, and here to see the VHDL written file (It is advised to open them using a code editor).

All trigonometric functions have been implemented using taylor series to a specific accuracy. Click here for more details

Square root function has been basically implemented using Newton's raphsons, with a little modification. Click here to see the formula.

Loops have been implemented recursively to be able to track the changes in the code, the list is updated with the result of the loop and then the decision is checked for the loop stop.

## Useful Links

The following link contains a link to the BEMO master folder containing the whole compiler and assembler files in addition to test files.

Click here to go to the masterfolder of the whole BEMO project.

The following folder contains a documentation to the whole CORONA PU project in addition to a zip file to download the project.

Click here go to the CORONA PU documentation folder.