ScoreBoard Implementation

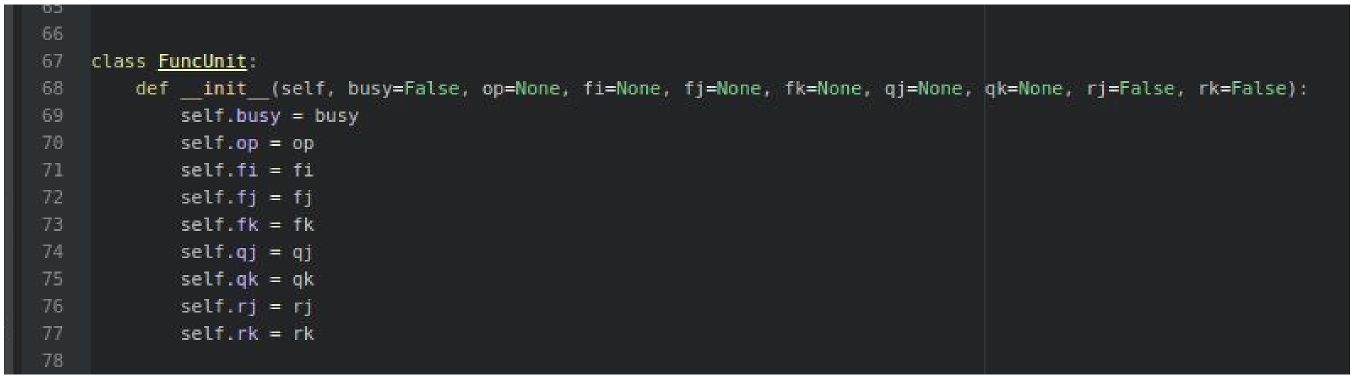
-CED16I002

Implementation Language : `Python3`
IDE used : `PyCharm`

- The main directory contains 2 .py files namely,
  * **Main_Func_Scoreboard.py** (say, file1)
  * **verilog_functions.py** (say, file2)
- File1 is the runner function which iterates through each Instruction (by incrementing PC) in the instruction set and schedules them accordingly using ScoreBoard implementation.
- File2 is a dependency module which contains 4 methods of **FPA, FPM, CLA, WTM** which dynamically create test-benches for fetching the outputs from their corresponding Verilog modules.

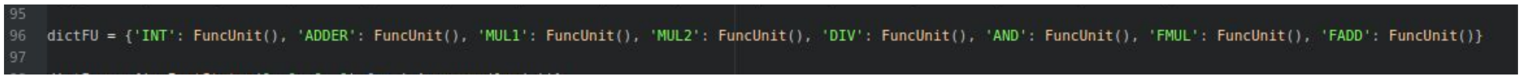
Implementation Idea:

Created a Python class to make instances of Functional Units, InstructionStatus Units, and RegisterStatus Units as shown below.



```python

class FuncUnit:
    def __init__(self, busy=False, op=None, fi=None, fj=None, fk=None, qj=None, qk=None, rj=False, rk=False):
        self.busy = busy
        self.op = op
        self.fi = fi
        self.fj = fj
        self.fk = fk
        self.qj = qj
        self.qk = qk
        self.rj = rj
        self.rk = rk

```

The class **FuncUnit** has 9 variables viz., busy, op, fi, fj, fk, qj, qk, rj, and rk which I have updated accordingly on every **Instruction Fetch** in the instruction set.



```python

dictFU = {'INT': FuncUnit(), 'ADDER': FuncUnit(), 'MUL1': FuncUnit(), 'MUL2': FuncUnit(), 'DIV': FuncUnit(), 'AND': FuncUnit(), 'FMUL': FuncUnit(), 'FADD': FuncUnit()}

```

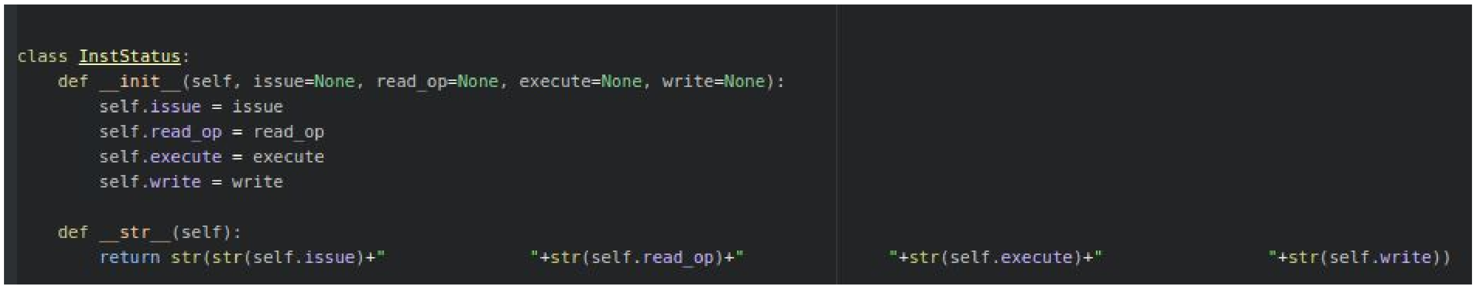
- I've created **8** instances of Functional Units namely,
  - INT
  - ADDER
  - MUL1
  - MUL2
  - DIV
  - AND
  - FMUL
  - FADD

which are updated during every instruction **fetch** and **write_back** stages. As shown in the picture, **dictFU** is the dictionary containing the above told functional units.

InstructionStatus Unit:-

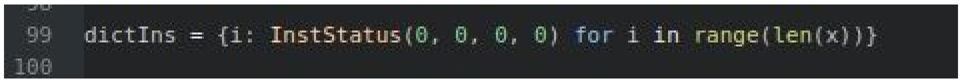
- Another class I've created is **InstStatus** which tracks down the info. about when an instruction is fetched, decoded, executed and written back.
- It has **4** variables namely,
    - issue
    - read_op
    - execute
    - write

  all initialized to "None" at the beginning.



```
class InstStatus:
    def __init__(self, issue=None, read_op=None, execute=None, write=None):
        self.issue = issue
        self.read_op = read_op
        self.execute = execute
        self.write = write

    def __str__(self):
        return str(str(self.issue)+"            "+str(self.read_op)+"              "+str(self.execute)+"              "+str(self.write))
```

- **dictIns** is the dictionary containing instances of InstStatus class for all instructions present in the instruction set.



```
dictIns = {i: InstStatus(0, 0, 0, 0) for i in range(len(x))}
```

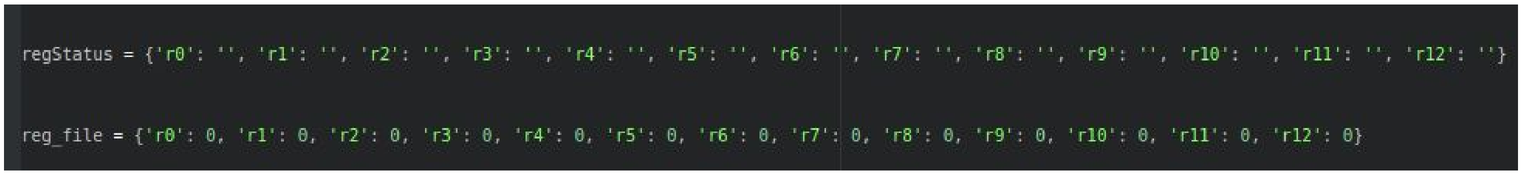
Here, **x** is a Python list containing all the set of instructions in string format.

RegisterStatus Unit:-

- This is implemented using the dictionary, **regStatus,** containing 12 entries, one for each register which indicate status of a register,
    - **'' -> empty string,** meaning no other instruction is writing into this register and is ready to be used.
    - **Non-empty String ->** some other instruction is writing into it at present and the requesting instruction has to wait.

Register File:-

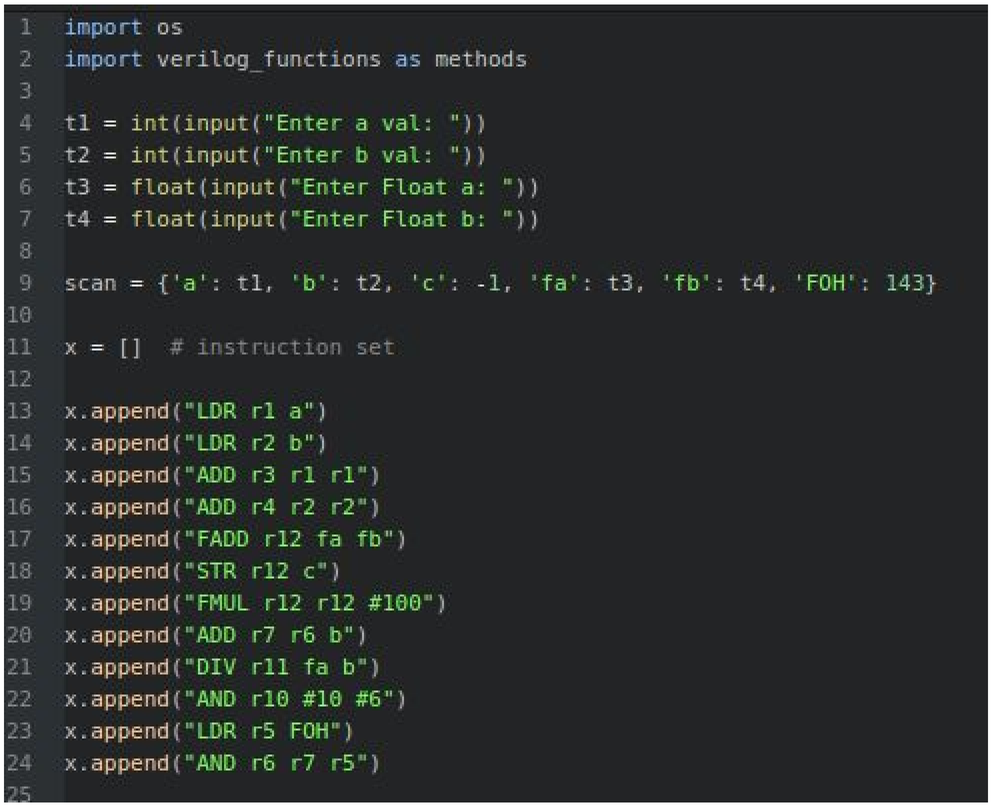
- Done with a dictionary, **reg_file**, where all 12 registers are initialized with zero.



```
regStatus = {'r0': '', 'r1': '', 'r2': '', 'r3': '', 'r4': '', 'r5': '', 'r6': '', 'r7': '', 'r8': '', 'r9': '', 'r10': '', 'r11': '', 'r12': ''}

reg_file = {'r0': 0, 'r1': 0, 'r2': 0, 'r3': 0, 'r4': 0, 'r5': 0, 'r6': 0, 'r7': 0, 'r8': 0, 'r9': 0, 'r10': 0, 'r11': 0, 'r12': 0}
```

Input Formats:-

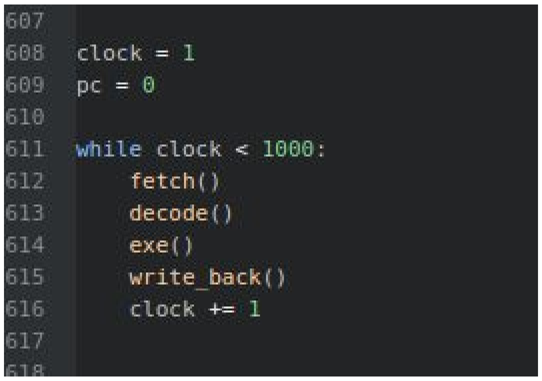
- Integer **a & b** and float **a & b** are inputted by the user and stored in a dictionar, **scan**, for easy access.
- List **x** is the set of all instructions.



```python
import os
import verilog_functions as methods

t1 = int(input("Enter a val: "))
t2 = int(input("Enter b val: "))
t3 = float(input("Enter Float a: "))
t4 = float(input("Enter Float b: "))

scan = {'a': t1, 'b': t2, 'c': -1, 'fa': t3, 'fb': t4, 'FOH': 143}

x = []  # instruction set

x.append("LDR r1 a")
x.append("LDR r2 b")
x.append("ADD r3 r1 r1")
x.append("ADD r4 r2 r2")
x.append("FADD r12 fa fb")
x.append("STR r12 c")
x.append("FMUL r12 r12 #100")
x.append("ADD r7 r6 b")
x.append("DIV r11 fa b")
x.append("AND r10 #10 #6")
x.append("LDR r5 FOH")
x.append("AND r6 r7 r5")

```

Caller Function:-

The main() function has a loop which calls **4 functions** viz., fetch(), decode(), exe(), write_back() repeatedly on every increment of clock until the clock reaches a certain limit(optimally, till the writeback of last instruction happens).



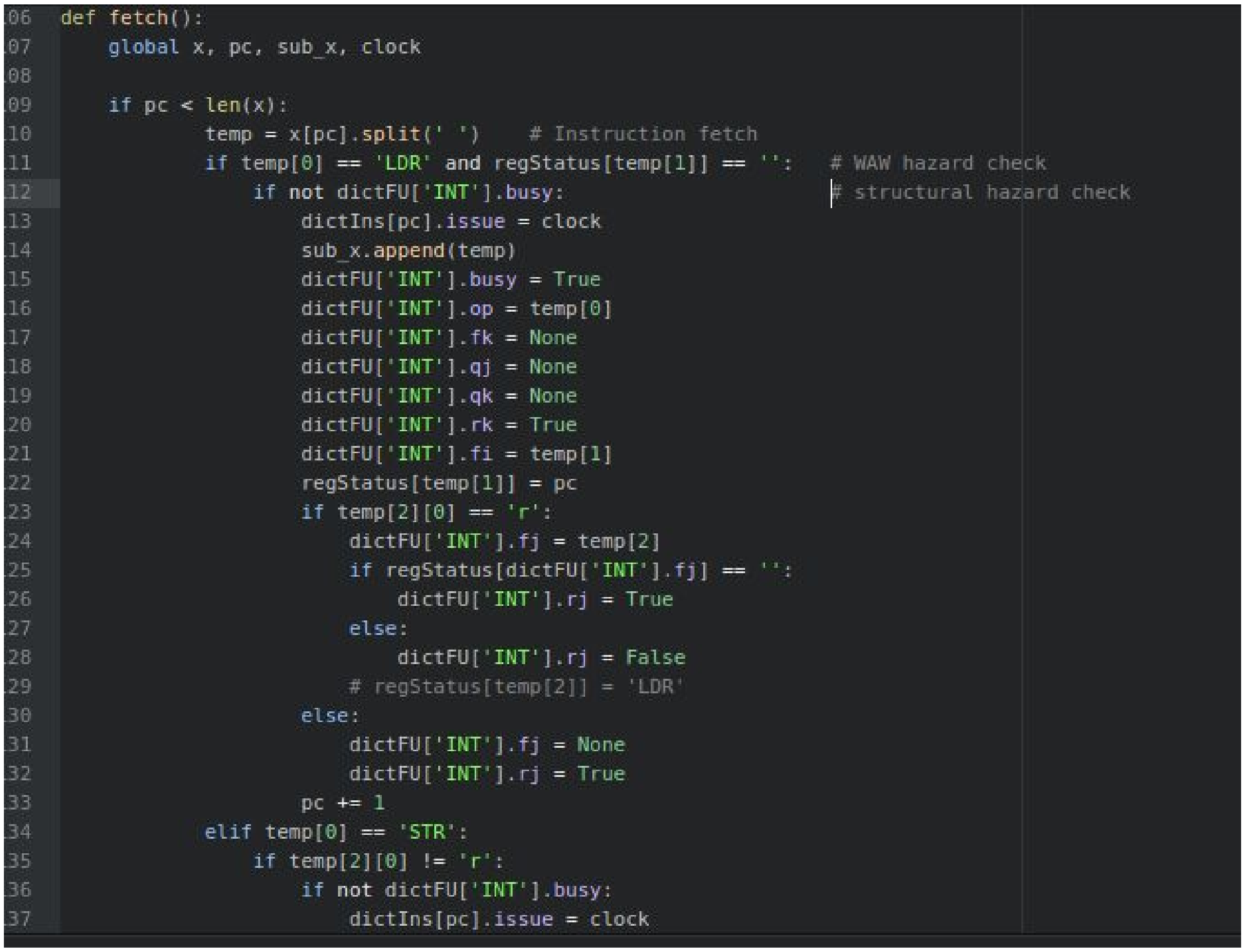
```python

clock = 1
pc = 0

while clock < 1000:
    fetch()
    decode()
    exe()
    write_back()
    clock += 1

```

Fetch() function & resolving WAW & STRUCTURAL Hazards:-

This function fetches each instruction from the instruction set and assigns it to the corresponding Functional Unit provided, it is free (**busy** is an empty string) and then **PC** is incremented. If not it is done in further clock cycles and **PC** remains the same.

This corresponds to **Structural Hazards**.

- split() function is used to extract the sub-strings from each instruction, containing the **op-code, sources & destination** in separate variables.
- Then, the register Status of the destination register is checked for an empty string. If found, the instruction is fetched else not.
    - This corresponds to the resolution of **WAW Hazards**.



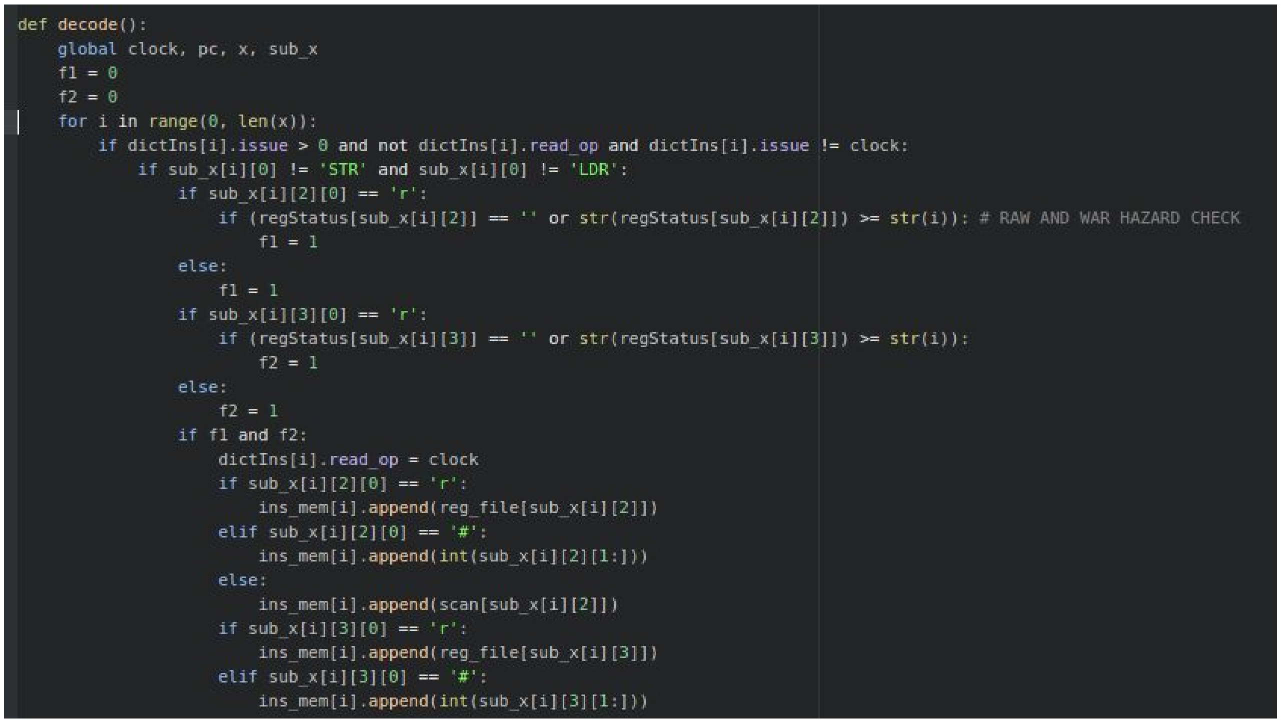
```
def fetch():
    global x, pc, sub_x, clock

    if pc < len(x):
            temp = x[pc].split(' ')    # Instruction fetch
            if temp[0] == 'LDR' and regStatus[temp[1]] == '':   # WAW hazard check
                if not dictFU['INT'].busy:                      # structural hazard check
                    dictIns[pc].issue = clock
                    sub_x.append(temp)
                    dictFU['INT'].busy = True
                    dictFU['INT'].op = temp[0]
                    dictFU['INT'].fk = None
                    dictFU['INT'].qj = None
                    dictFU['INT'].qk = None
                    dictFU['INT'].rk = True
                    dictFU['INT'].fi = temp[1]
                    regStatus[temp[1]] = pc
                    if temp[2][0] == 'r':
                        dictFU['INT'].fj = temp[2]
                        if regStatus[dictFU['INT'].fj] == '':
                            dictFU['INT'].rj = True
                        else:
                            dictFU['INT'].rj = False
                        # regStatus[temp[2]] = 'LDR'
                    else:
                        dictFU['INT'].fj = None
                        dictFU['INT'].rj = True
                    pc += 1
            elif temp[0] == 'STR':
                if temp[2][0] != 'r':
                    if not dictFU['INT'].busy:
                        dictIns[pc].issue = clock
```

Decode() Function and RAW & WAR HAZARDS:-

- On every function call, Decode() iterates through the instruction set and decodes one by one, only if the issue(**Fetch Stage**) of that particular instruction is done.
- It checks if the RegStatus(es) of the source register(s) is(are) empty set(s). If yes, then all previous instructions are done writing into them currently and the data is ready to be read.
- This corresponds to the check of **RAW Hazards**.

- Later, the data is read accordingly and stored in a buffer called (**ins_mem**).

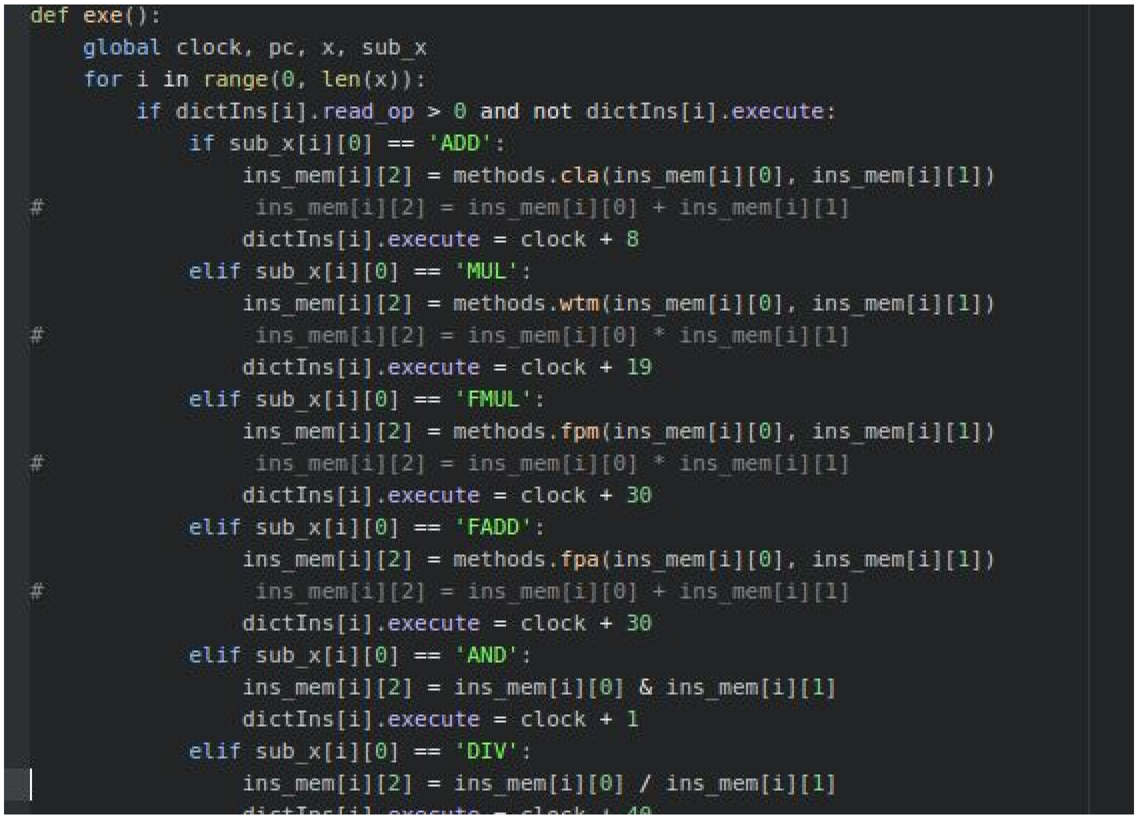
Note: I have answered WAR Hazard solutions in my implementation issues in later pages.



```python
def decode():
    global clock, pc, x, sub_x
    f1 = 0
    f2 = 0
    for i in range(0, len(x)):
        if dictIns[i].issue > 0 and not dictIns[i].read_op and dictIns[i].issue != clock:
            if sub_x[i][0] != 'STR' and sub_x[i][0] != 'LDR':
                if sub_x[i][2][0] == 'r':
                    if (regStatus[sub_x[i][2]] == '' or str(regStatus[sub_x[i][2]]) >= str(i)): # RAW AND WAR HAZARD CHECK
                        f1 = 1
                else:
                    f1 = 1
                if sub_x[i][3][0] == 'r':
                    if (regStatus[sub_x[i][3]] == '' or str(regStatus[sub_x[i][3]]) >= str(i)):
                        f2 = 1
                else:
                    f2 = 1
                if f1 and f2:
                    dictIns[i].read_op = clock
                    if sub_x[i][2][0] == 'r':
                        ins_mem[i].append(reg_file[sub_x[i][2]])
                    elif sub_x[i][2][0] == '#':
                        ins_mem[i].append(int(sub_x[i][2][1:]))
                    else:
                        ins_mem[i].append(scan[sub_x[i][2]])
                    if sub_x[i][3][0] == 'r':
                        ins_mem[i].append(reg_file[sub_x[i][3]])
                    elif sub_x[i][3][0] == '#':
                        ins_mem[i].append(int(sub_x[i][3][1:]))
```

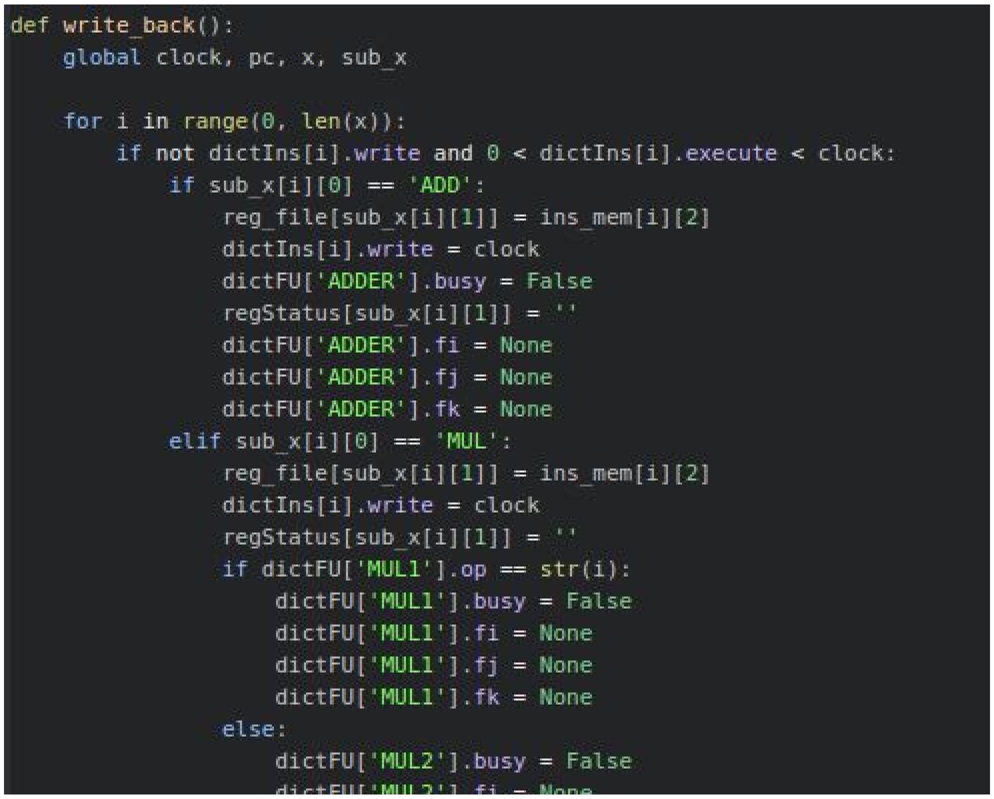
Execute():-

- This function performs a check on the issue and decode timelines and executes the instruction only if both the previous satges are done (**issue & read_op values >** 0).
- It then checks the op-code and does the corresponding Arithmetic invoking the methods from **File2(verilog_functions.py)** and stores back the result in the ins_mem buffer.



```
def exe():
    global clock, pc, x, sub_x
    for i in range(0, len(x)):
        if dictIns[i].read_op > 0 and not dictIns[i].execute:
            if sub_x[i][0] == 'ADD':
                ins_mem[i][2] = methods.cla(ins_mem[i][0], ins_mem[i][1])
#                ins_mem[i][2] = ins_mem[i][0] + ins_mem[i][1]
                dictIns[i].execute = clock + 8
            elif sub_x[i][0] == 'MUL':
                ins_mem[i][2] = methods.wtm(ins_mem[i][0], ins_mem[i][1])
#                ins_mem[i][2] = ins_mem[i][0] * ins_mem[i][1]
                dictIns[i].execute = clock + 19
            elif sub_x[i][0] == 'FMUL':
                ins_mem[i][2] = methods.fpm(ins_mem[i][0], ins_mem[i][1])
#                ins_mem[i][2] = ins_mem[i][0] * ins_mem[i][1]
                dictIns[i].execute = clock + 30
            elif sub_x[i][0] == 'FADD':
                ins_mem[i][2] = methods.fpa(ins_mem[i][0], ins_mem[i][1])
#                ins_mem[i][2] = ins_mem[i][0] + ins_mem[i][1]
                dictIns[i].execute = clock + 30
            elif sub_x[i][0] == 'AND':
                ins_mem[i][2] = ins_mem[i][0] & ins_mem[i][1]
                dictIns[i].execute = clock + 1
            elif sub_x[i][0] == 'DIV':
                ins_mem[i][2] = ins_mem[i][0] / ins_mem[i][1]
```

write_back():-



```
def write_back():
    global clock, pc, x, sub_x

    for i in range(0, len(x)):
        if not dictIns[i].write and 0 < dictIns[i].execute < clock:
            if sub_x[i][0] == 'ADD':
                reg_file[sub_x[i][1]] = ins_mem[i][2]
                dictIns[i].write = clock
                dictFU['ADDER'].busy = False
                regStatus[sub_x[i][1]] = ''
                dictFU['ADDER'].fi = None
                dictFU['ADDER'].fj = None
                dictFU['ADDER'].fk = None
            elif sub_x[i][0] == 'MUL':
                reg_file[sub_x[i][1]] = ins_mem[i][2]
                dictIns[i].write = clock
                regStatus[sub_x[i][1]] = ''
                if dictFU['MUL1'].op == str(i):
                    dictFU['MUL1'].busy = False
                    dictFU['MUL1'].fi = None
                    dictFU['MUL1'].fj = None
                    dictFU['MUL1'].fk = None
                else:
                    dictFU['MUL2'].busy = False
```

- This function iterates through every instruction and writes back the result only if the execute stage of the instruction is done.
- It captures back the result from the **ins_mem** buffer and stores the value into the corresponding destination register.

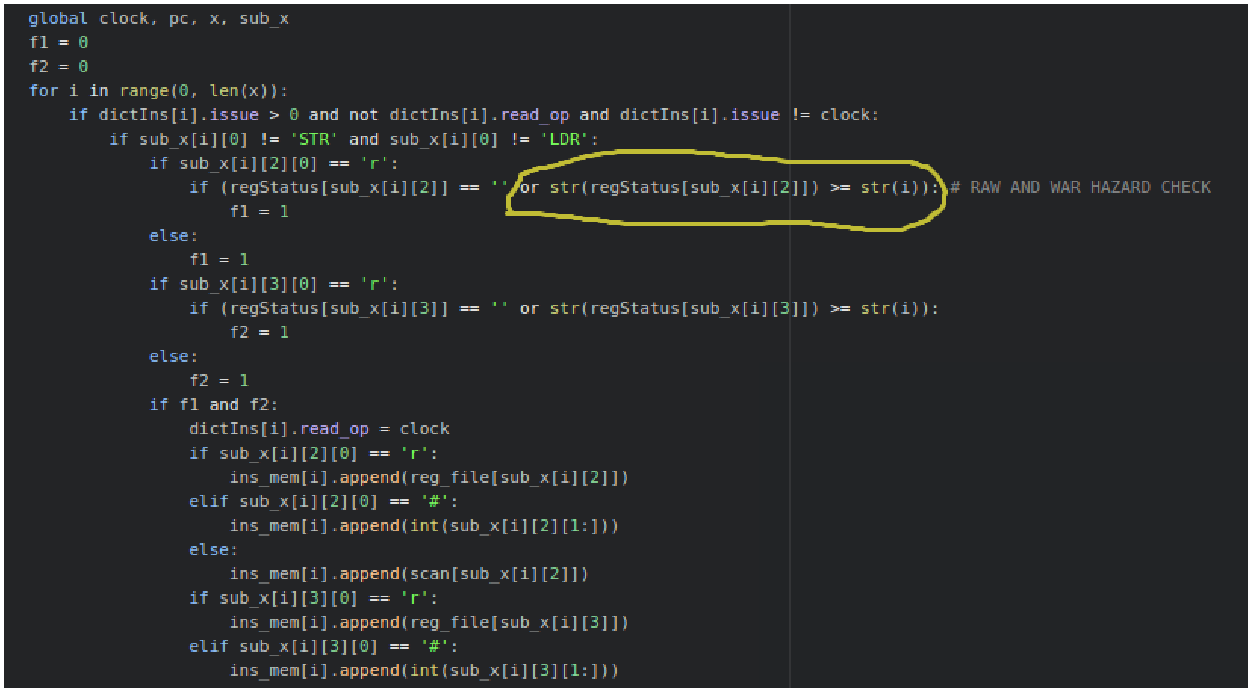
-- In all of these functions, on a successful call, current clock value is captured and the InstrutionStatus table is updated.

My Implementation issues & WAR Hazard check:-

- Since the fetch function asynchronously(in my code) does instruction issue by checking the availability of the corresponding functional Units, it updates the status of the destination registers to not null due to which the decode stage and execute stages may not happen, of some previous instructions which have the same registers as sources.
- As a result, **deadlock occurs** and the process doesn't proceed forward.

The encircled line is added to resolve the deadlock.

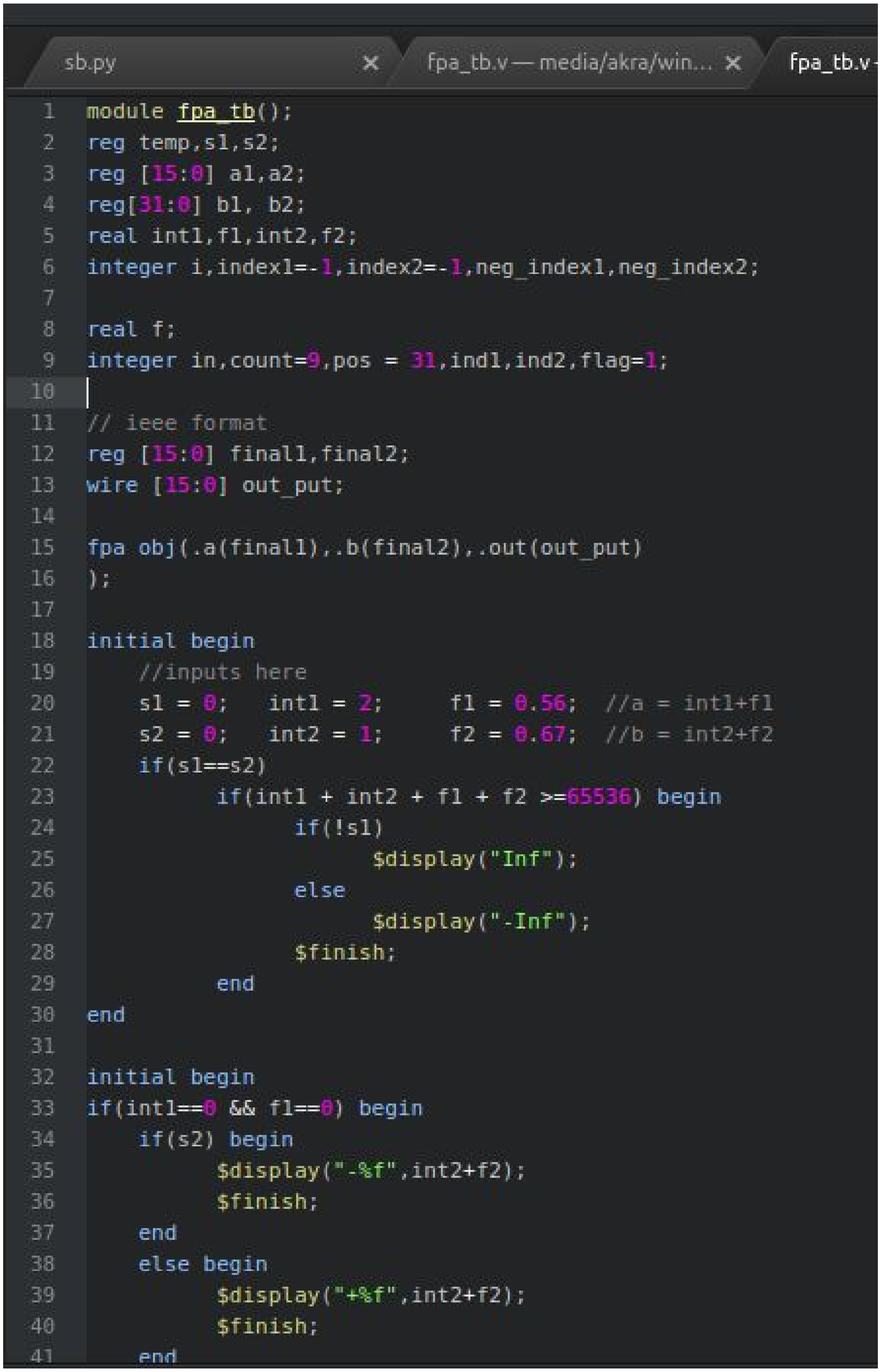
- **This also corresponds to the check of WAR Hazards.**



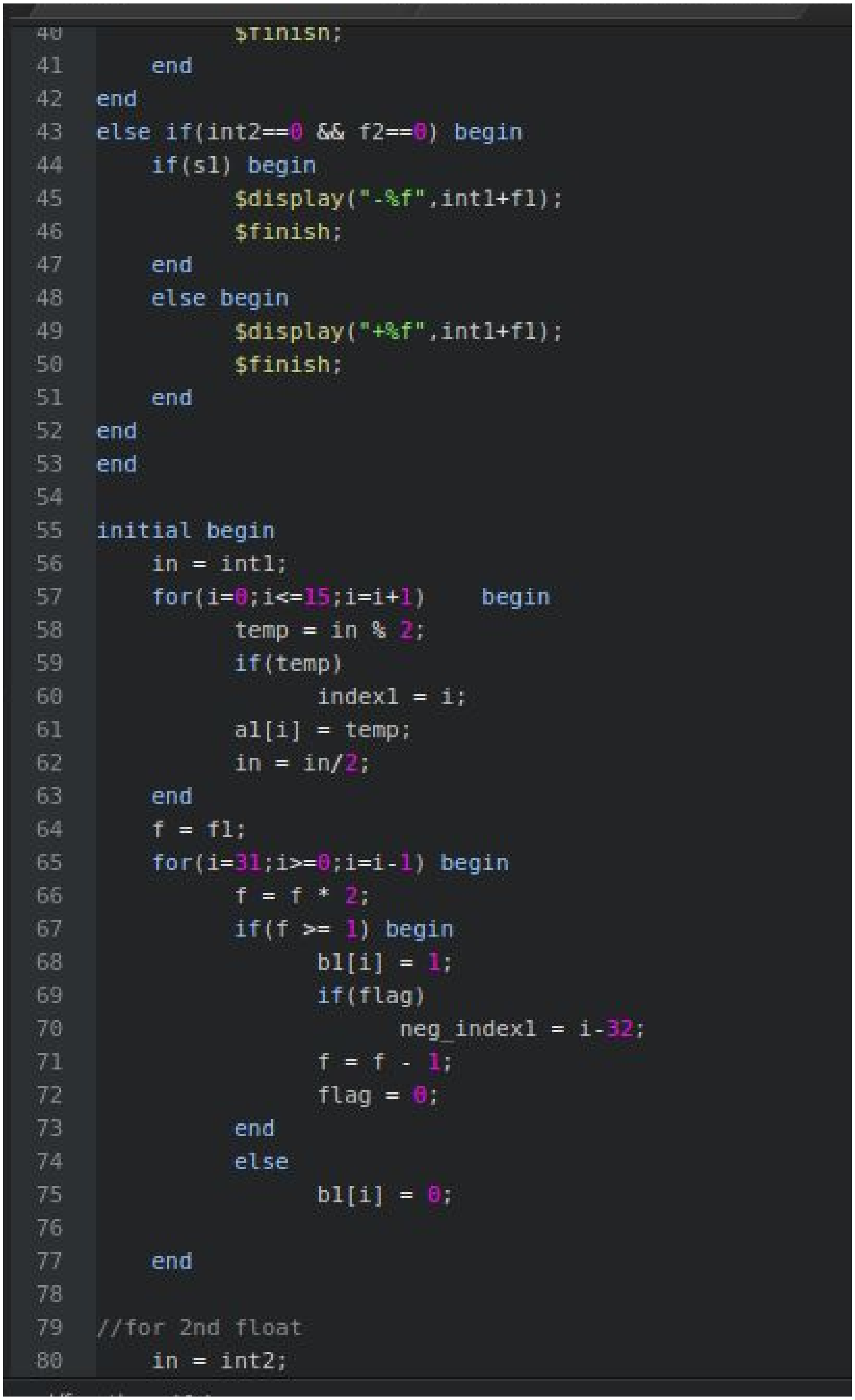
```
global clock, pc, x, sub_x
f1 = 0
f2 = 0
for i in range(0, len(x)):
    if dictIns[i].issue > 0 and not dictIns[i].read_op and dictIns[i].issue != clock:
        if sub_x[i][0] != 'STR' and sub_x[i][0] != 'LDR':
            if sub_x[i][2][0] == 'r':
                if (regStatus[sub_x[i][2]] == '' or str(regStatus[sub_x[i][2]]) >= str(i)): # RAW AND WAR HAZARD CHECK
                    f1 = 1
            else:
                f1 = 1
            if sub_x[i][3][0] == 'r':
                if (regStatus[sub_x[i][3]] == '' or str(regStatus[sub_x[i][3]]) >= str(i)):
                    f2 = 1
            else:
                f2 = 1
            if f1 and f2:
                dictIns[i].read_op = clock
                if sub_x[i][2][0] == 'r':
                    ins_mem[i].append(reg_file[sub_x[i][2]])
                elif sub_x[i][2][0] == '#':
                    ins_mem[i].append(int(sub_x[i][2][1:]))
                else:
                    ins_mem[i].append(scan[sub_x[i][2]])
                if sub_x[i][3][0] == 'r':
                    ins_mem[i].append(reg_file[sub_x[i][3]])
                elif sub_x[i][3][0] == '#':
                    ins_mem[i].append(int(sub_x[i][3][1:]))
```

Verilog Module Updates:-

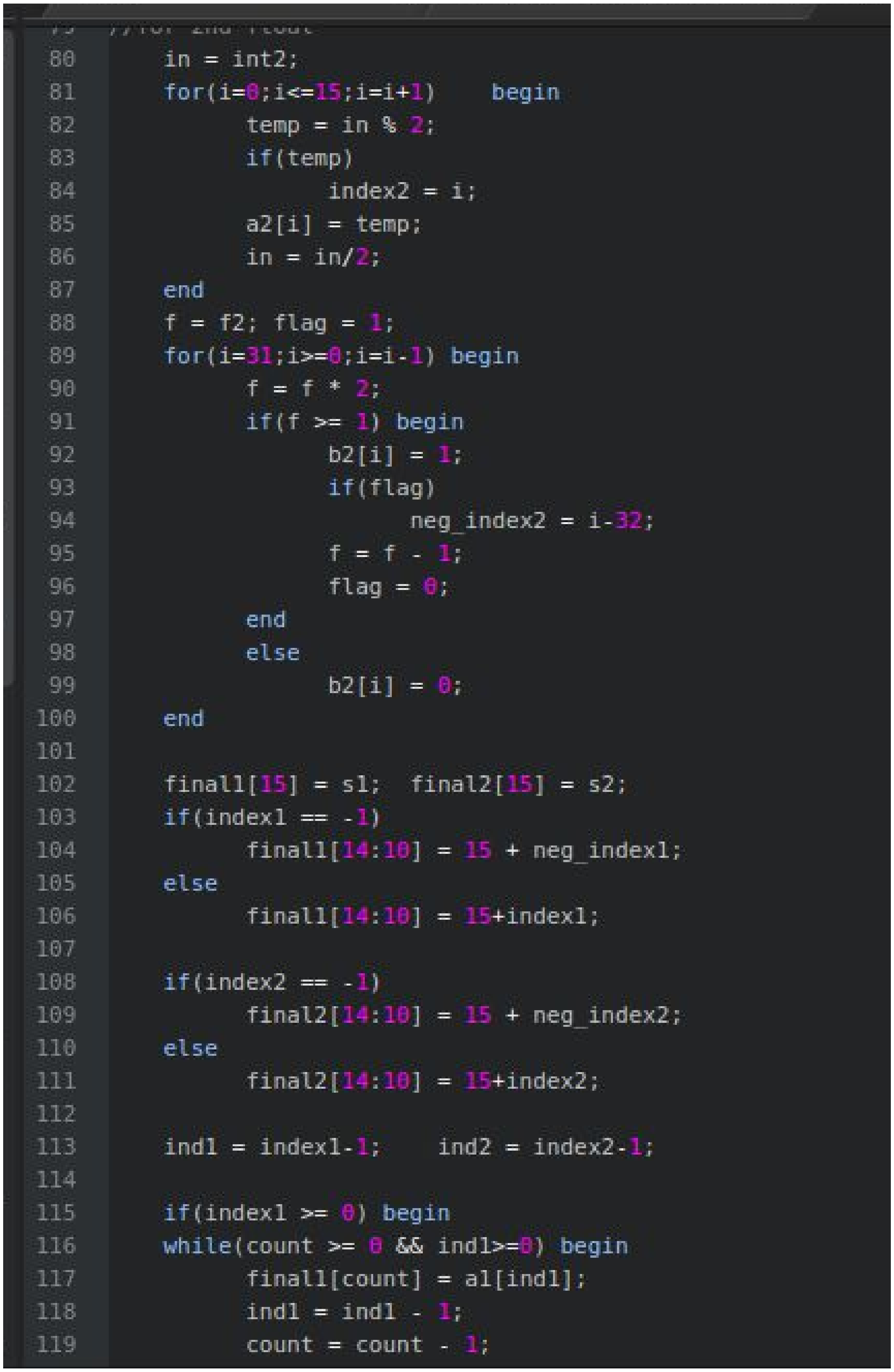
- Inputs are given in Verilog's **real datatype** as shown below where **s1,s2** contains **sign(0 or 1), int1&int2** contains integer part, **f1&f2** has decimal parts.
- These are then converted into 16bit IEEE floating point input as follows:



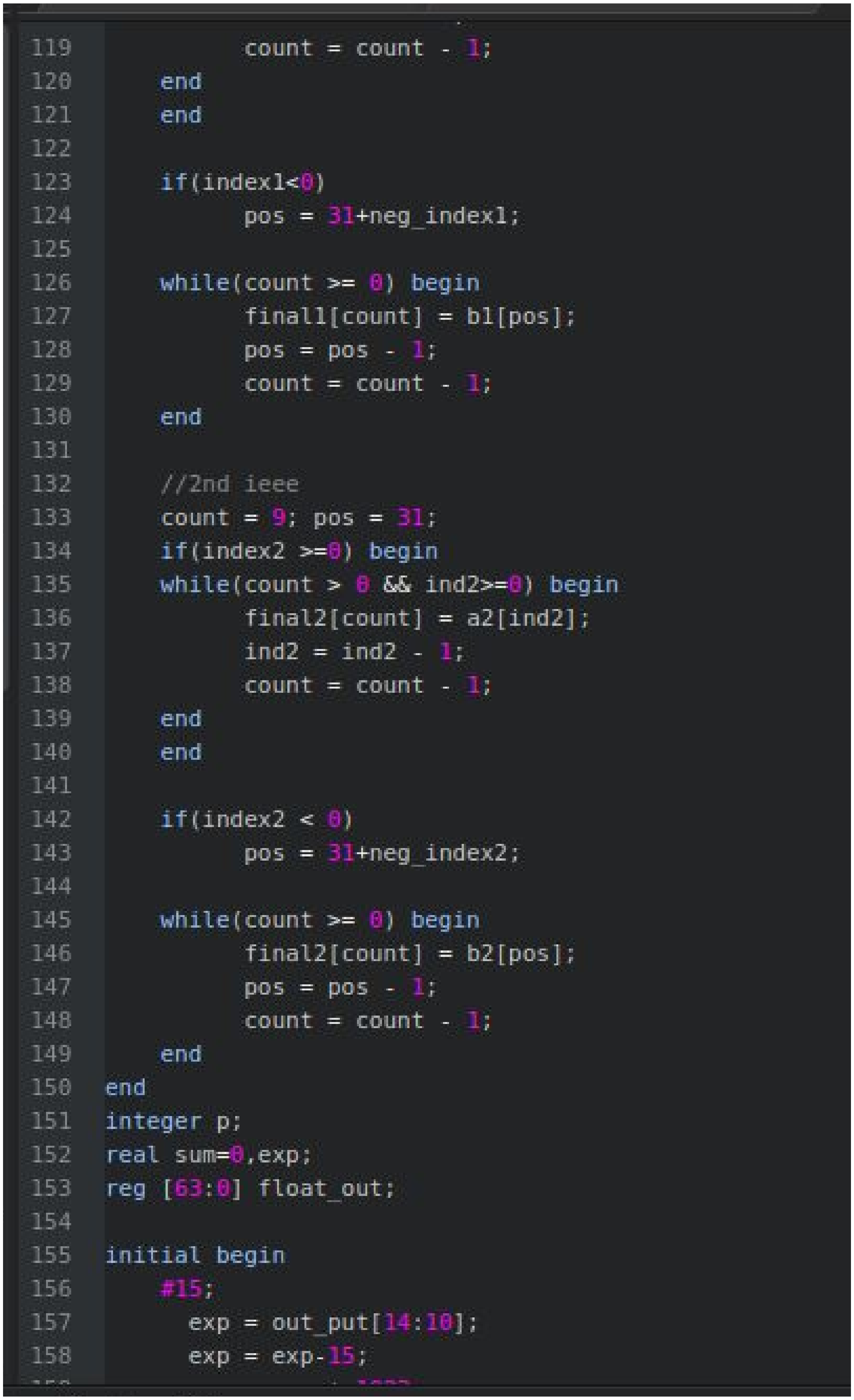
```
module fpa_tb();
reg temp,s1,s2;
reg [15:0] a1,a2;
reg[31:0] b1, b2;
real int1,f1,int2,f2;
integer i,index1=-1,index2=-1,neg_index1,neg_index2;

real f;
integer in,count=9,pos = 31,ind1,ind2,flag=1;

// ieee format
reg [15:0] final1,final2;
wire [15:0] out_put;

fpa obj(.a(final1),.b(final2),.out(out_put)
);

initial begin
    //inputs here
    s1 = 0;   int1 = 2;      f1 = 0.56;  //a = int1+f1
    s2 = 0;   int2 = 1;      f2 = 0.67;  //b = int2+f2
    if(s1==s2)
          if(int1 + int2 + f1 + f2 >=65536) begin
                if(!s1)
                      $display("Inf");
                else
                      $display("-Inf");
                $finish;
          end
end

initial begin
if(int1==0 && f1==0) begin
    if(s2) begin
          $display("-%f",int2+f2);
          $finish;
    end
    else begin
          $display("+%f",int2+f2);
          $finish;
    end
```



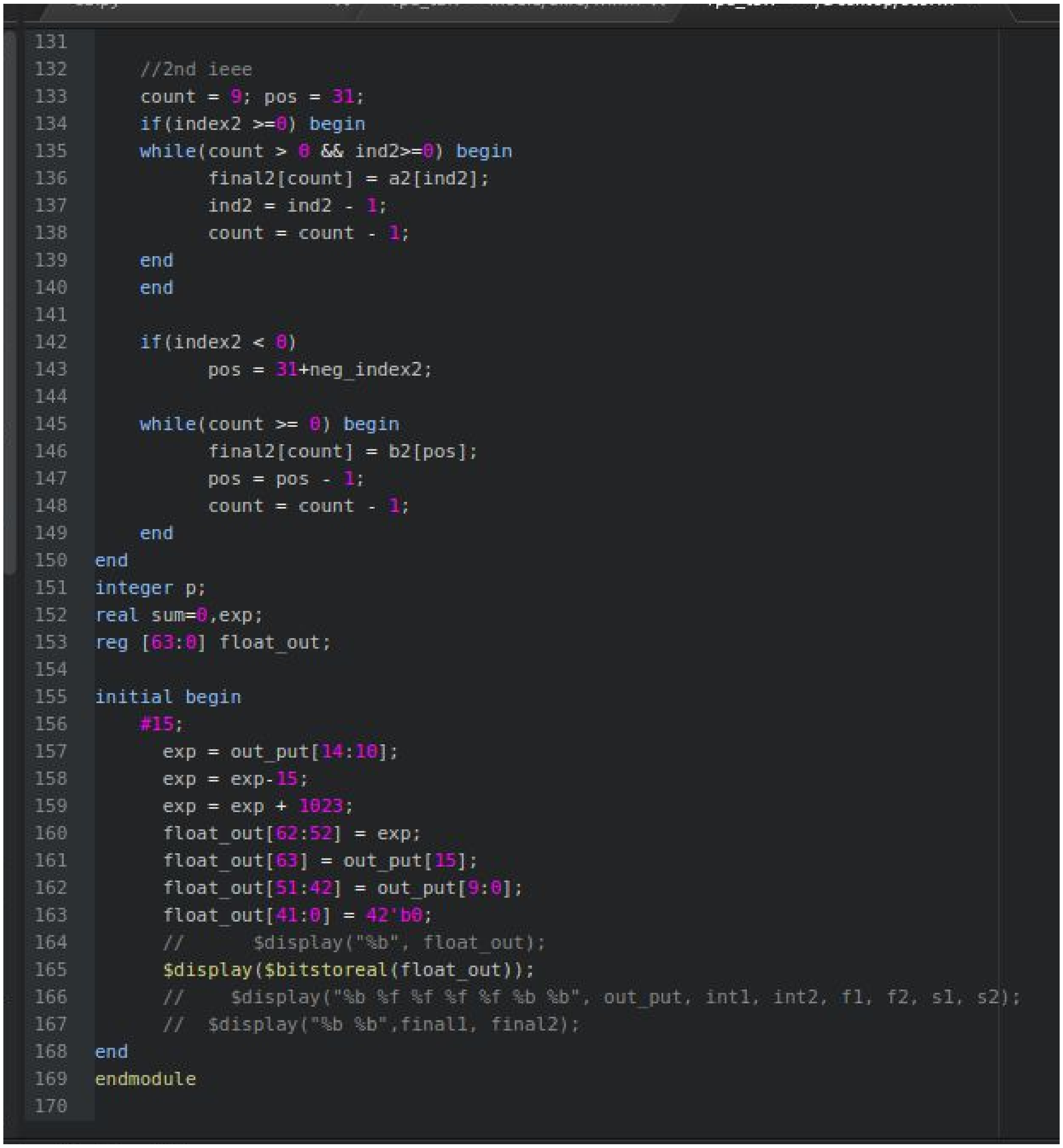
```
          $finish;
    end
end
else if(int2==0 && f2==0) begin
    if(s1) begin
          $display("-%f",int1+f1);
          $finish;
    end
    else begin
          $display("+%f",int1+f1);
          $finish;
    end
end
end

initial begin
    in = int1;
    for(i=0;i<=15;i=i+1)    begin
          temp = in % 2;
          if(temp)
                index1 = i;
          a1[i] = temp;
          in = in/2;
    end
    f = f1;
    for(i=31;i>=0;i=i-1) begin
          f = f * 2;
          if(f >= 1) begin
                b1[i] = 1;
                if(flag)
                      neg_index1 = i-32;
                f = f - 1;
                flag = 0;
          end
          else
                b1[i] = 0;

    end

//for 2nd float
    in = int2;
```



```
    in = int2;
    for(i=0;i<=15;i=i+1)    begin
          temp = in % 2;
          if(temp)
                index2 = i;
          a2[i] = temp;
          in = in/2;
    end
    f = f2; flag = 1;
    for(i=31;i>=0;i=i-1) begin
          f = f * 2;
          if(f >= 1) begin
                b2[i] = 1;
                if(flag)
                      neg_index2 = i-32;
                f = f - 1;
                flag = 0;
          end
          else
                b2[i] = 0;
    end

    final1[15] = s1;  final2[15] = s2;
    if(index1 == -1)
          final1[14:10] = 15 + neg_index1;
    else
          final1[14:10] = 15+index1;

    if(index2 == -1)
          final2[14:10] = 15 + neg_index2;
    else
          final2[14:10] = 15+index2;

    ind1 = index1-1;    ind2 = index2-1;

    if(index1 >= 0) begin
    while(count >= 0 && ind1>=0) begin
          final1[count] = a1[ind1];
          ind1 = ind1 - 1;
          count = count - 1;
```



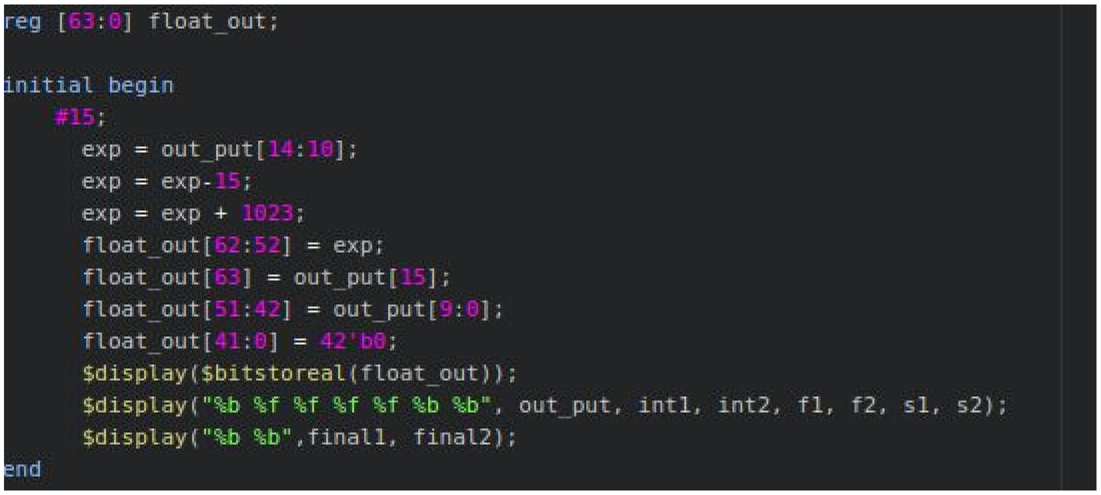
```
          count = count - 1;
    end
    end

    if(index1<0)
          pos = 31+neg_index1;

    while(count >= 0) begin
          final1[count] = b1[pos];
          pos = pos - 1;
          count = count - 1;
    end

    //2nd ieee
    count = 9; pos = 31;
    if(index2 >=0) begin
    while(count > 0 && ind2>=0) begin
          final2[count] = a2[ind2];
          ind2 = ind2 - 1;
          count = count - 1;
    end
    end

    if(index2 < 0)
          pos = 31+neg_index2;

    while(count >= 0) begin
          final2[count] = b2[pos];
          pos = pos - 1;
          count = count - 1;
    end
end
integer p;
real sum=0,exp;
reg [63:0] float_out;

initial begin
    #15;
      exp = out_put[14:10];
      exp = exp-15;
```



```

    //2nd ieee
    count = 9; pos = 31;
    if(index2 >=0) begin
    while(count > 0 && ind2>=0) begin
          final2[count] = a2[ind2];
          ind2 = ind2 - 1;
          count = count - 1;
    end
    end

    if(index2 < 0)
          pos = 31+neg_index2;

    while(count >= 0) begin
          final2[count] = b2[pos];
          pos = pos - 1;
          count = count - 1;
    end
end
integer p;
real sum=0,exp;
reg [63:0] float_out;

initial begin
    #15;
      exp = out_put[14:10];
      exp = exp-15;
      exp = exp + 1023;
      float_out[62:52] = exp;
      float_out[63] = out_put[15];
      float_out[51:42] = out_put[9:0];
      float_out[41:0] = 42'b0;
      //      $display("%b", float_out);
      $display($bitstoreal(float_out));
      //    $display("%b %f %f %f %f %b %b", out_put, int1, int2, f1, f2, s1, s2);
      //  $display("%b %b",final1, final2);
end
endmodule

```

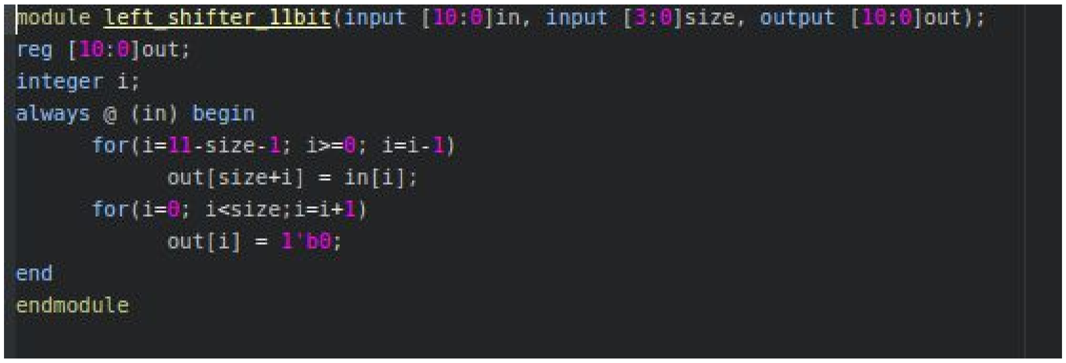
- Converted 16bit Half Precision Floating Point output of Verilog modules to decimal floating point values.
- Used **$bitstoreal()** of SystemVerilog which converts a 64-bit double precision value to decimal float value.

So, I have converted my 16bit output to 64bit value first and then passed the value to the above-said function to get the final answer in decimal floating point value.

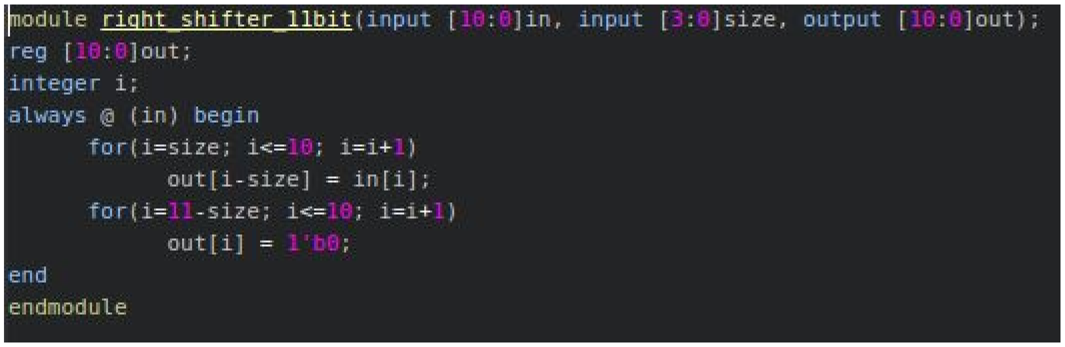


```
reg [63:0] float_out;

initial begin
    #15;
      exp = out_put[14:10];
      exp = exp-15;
      exp = exp + 1023;
      float_out[62:52] = exp;
      float_out[63] = out_put[15];
      float_out[51:42] = out_put[9:0];
      float_out[41:0] = 42'b0;
      $display($bitstoreal(float_out));
      $display("%b %f %f %f %f %b %b", out_put, int1, int2, f1, f2, s1, s2);
      $display("%b %b",final1, final2);
end
```

- Also, I have updated the shift operators “<<”, “>>” with a **11bit left shifter** and a **11bit right shifter** in floating-point Arithmetic values.

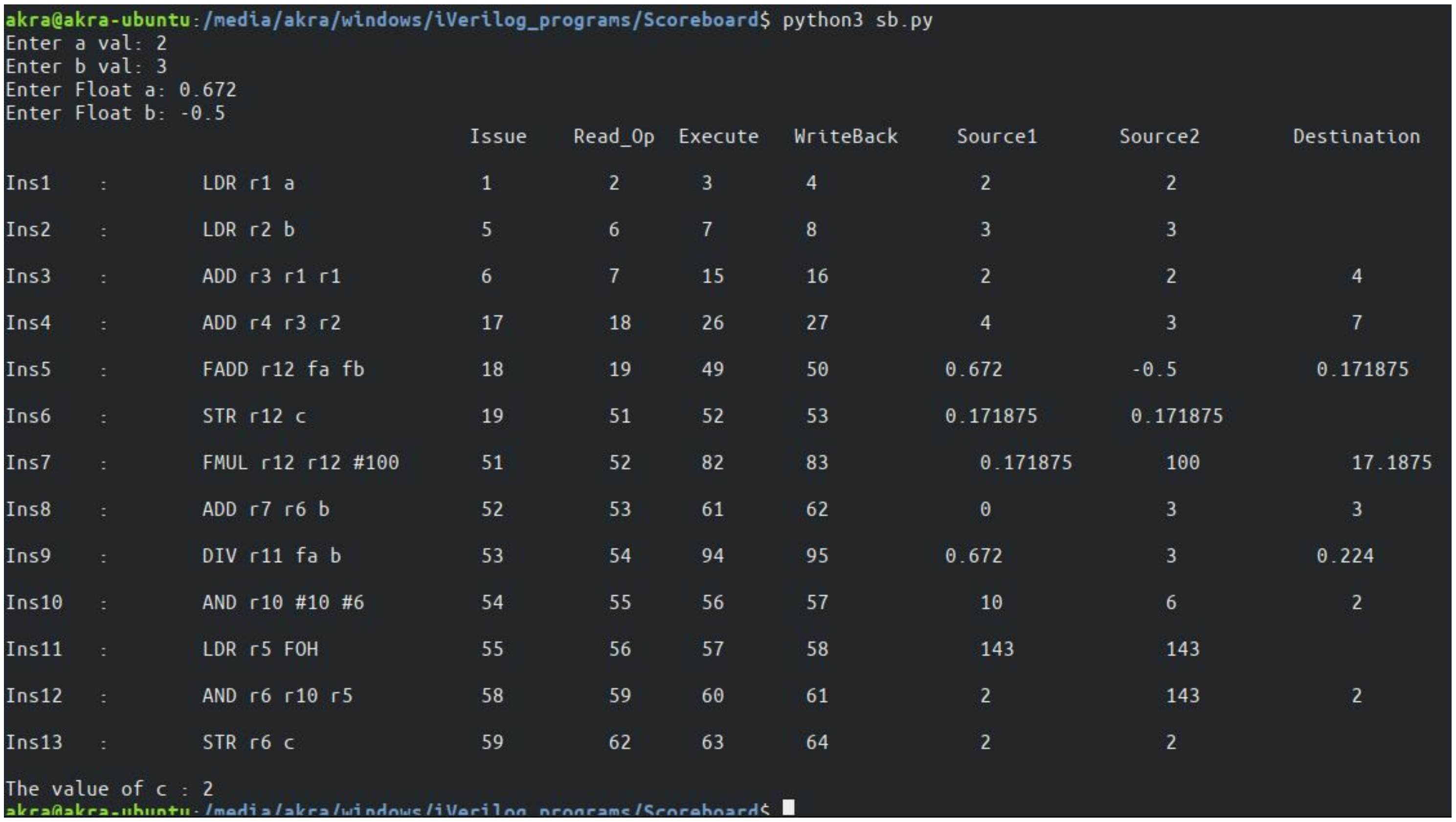


```
module left_shifter_11bit(input [10:0]in, input [3:0]size, output [10:0]out);
reg [10:0]out;
integer i;
always @ (in) begin
      for(i=11-size-1; i>=0; i=i-1)
            out[size+i] = in[i];
      for(i=0; i<size;i=i+1)
            out[i] = 1'b0;
end
endmodule
```



```
module right_shifter_11bit(input [10:0]in, input [3:0]size, output [10:0]out);
reg [10:0]out;
integer i;
always @ (in) begin
      for(i=size; i<=10; i=i+1)
            out[i-size] = in[i];
      for(i=11-size; i<=10; i=i+1)
            out[i] = 1'b0;
end
endmodule
```

Output:-



```
akra@akra-ubuntu:/media/akra/windows/iVerilog_programs/Scoreboard$ python3 sb.py
Enter a val: 2
Enter b val: 3
Enter Float a: 0.672
Enter Float b: -0.5
                                        Issue    Read_Op  Execute   WriteBack     Source1       Source2        Destination

Ins1     :        LDR r1 a               1          2       3        4              2              2

Ins2     :        LDR r2 b               5          6       7        8              3              3

Ins3     :        ADD r3 r1 r1           6          7       15       16             2              2               4

Ins4     :        ADD r4 r3 r2           17         18      26       27             4              3               7

Ins5     :        FADD r12 fa fb         18         19      49       50          0.672          -0.5             0.171875

Ins6     :        STR r12 c              19         51      52       53          0.171875       0.171875

Ins7     :        FMUL r12 r12 #100      51         52      82       83             0.171875       100             17.1875

Ins8     :        ADD r7 r6 b            52         53      61       62             0              3               3

Ins9     :        DIV r11 fa b           53         54      94       95          0.672           3             0.224

Ins10    :        AND r10 #10 #6         54         55      56       57             10             6               2

Ins11    :        LDR r5 FOH             55         56      57       58             143            143

Ins12    :        AND r6 r10 r5          58         59      60       61             2              143             2

Ins13    :        STR r6 c               59         62      63       64             2              2

The value of c : 2
akra@akra-ubuntu:/media/akra/windows/iVerilog_programs/Scoreboard$
```

-- 0 --