CS6230_MAC_Unit_project REPORT

Assignment 2

Done by: DANIEL MARK ISAAC

ROLL NO: NS24Z353

DESCRIPTION:

This report summarises assignment 2 of the MAC project.

GIVEN SPECIFICATION:

The following diagram illustrates the specification:

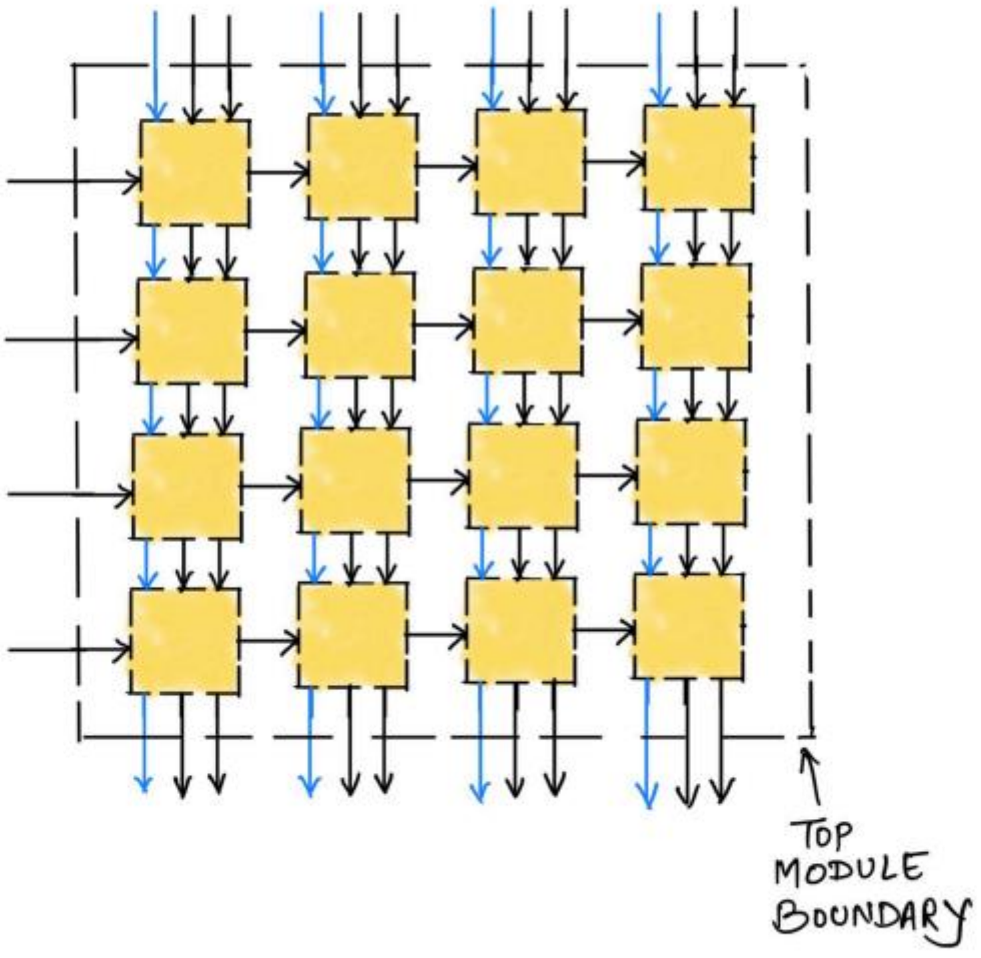


Figure 7. Systolic array built by interconnecting 16 MAC modules into a mesh.

APPROACH TAKEN:

The following list elaborates the approach taken to tackle phase 2 of the project:

1) Made a wrapper for the MAC bsv developed in phase 1, to include the extra output pins.

2) Interconnected two MAC modules in 2x1 format to see whether the inputs and outputs are relayed to second stage as intended.

3) Interconnected four MAC modules in 4x1 (A single column) format to see whether the inputs and outputs are relayed to subsequent stages as intended.

3) Interconnected eight MAC modules in 4x2 (two columns) format to see whether the inputs and outputs are relayed to subsequent stages as intended.

4) Interconnected twelve MAC modules in 4x3 (three columns) format to see whether the inputs and outputs are relayed to subsequent stages as intended.

5) Interconnected sixteen MAC modules in 4x4 (four columns) format to see whether the inputs and outputs are relayed to subsequent stages as intended.

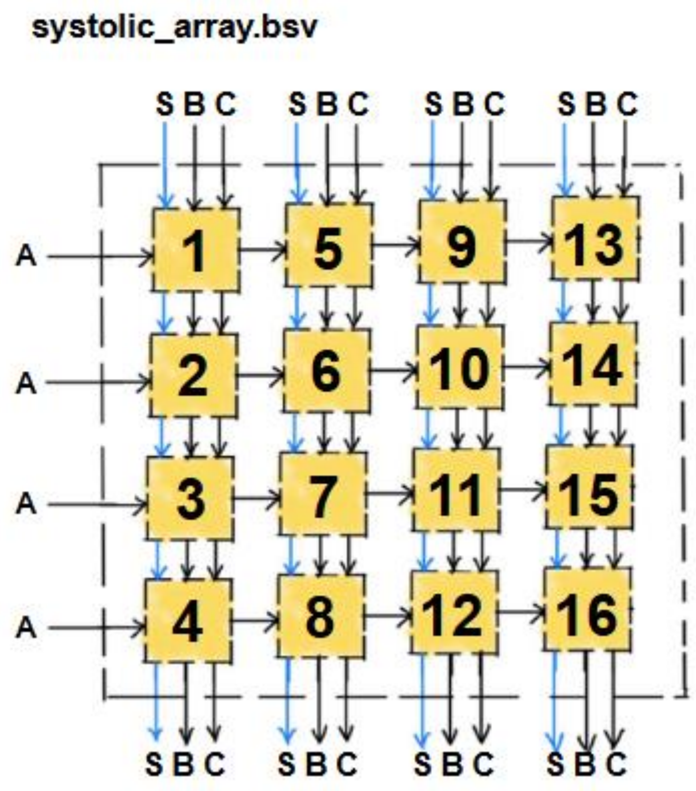
6) Created reference model for systolic array, by connecting sixteen instances of the MAC reference model developed in phase 1, in a mesh structure similar to bsv.

7) Created testbench in cocotb, which will drive random inputs to both RTL and reference model and checks the RTL output with reference model output.

SYSTOLIC ARRAY DESIGN:

In this section the design of systolic array is explained.

Sixteen MAC units are connected in a mesh pattern as shown in the below image:



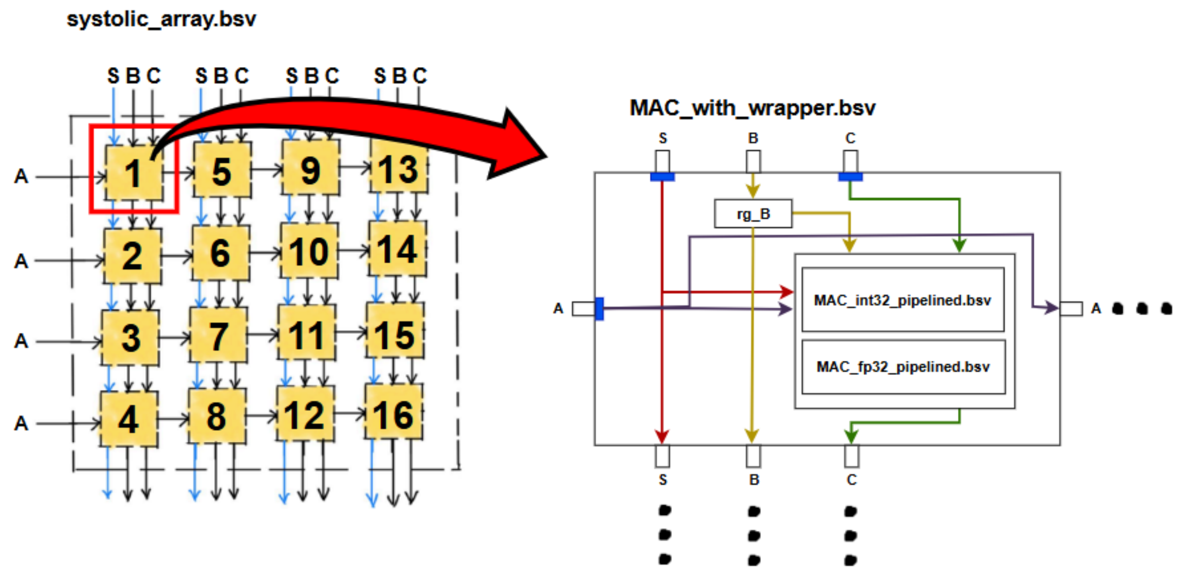
The input A is fed in from the left side of the design. The input B and input S are fed in from top of the design. Input C fed in from top will be fed with zeroes always.

The S and B pins on the bottom of the design are irrelevant. The c pins at the bottom are the output pins which gives the output of multiplication.

Each of the individual MAC units (yellow coloured box) is the same MAC unit developed in phase 1 of this project except for minor modification. The modifications are seamlessly introduced by creating a sort of wrapper code around the pre-existing MAC unit.

The first modification is addition of a register rg_B which can persist its value throughout a calculation unless it gets a new value in which case it propagates its value to MAC unit immediately below it.

The next modification is addition of four output pins which can relay the value of A,B,C,S to the next MAC unit. This relaying mechanism is achieved by using wires to communicate across rules which will be explained in detail in upcoming sections.

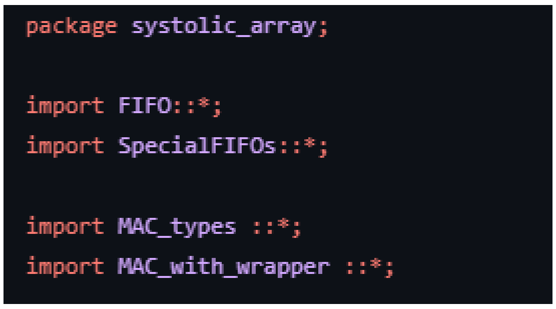


The above image illustrates the connections inside a particular MAC unit. The blue coloured boxes near the inputs A,S,C are FIFOs. From the FIFOs the values are sent to the MAC unit and also relayed to other MAC units. Input B is fed to rg_B which in turn is fed to MAC unit and also relayed to other MAC units as well.

According to input S, the inputs A,B and C are given to either int MAC or float MAC and the output is also gathered from relevant module and passed on.

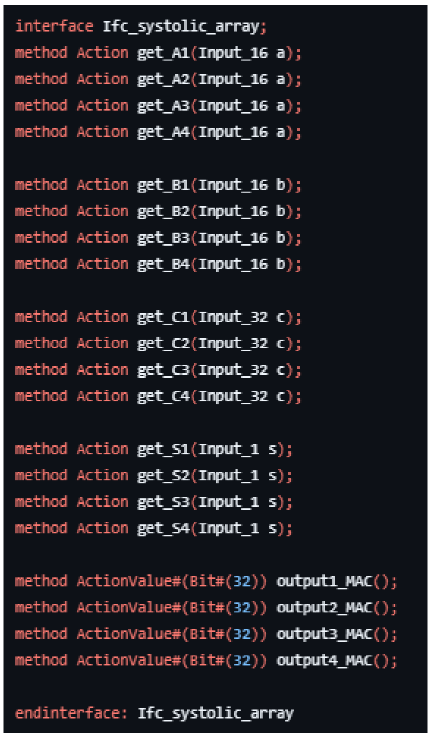
systolic_array.bsv:

In this section, systolic_array.bsv code is explained.



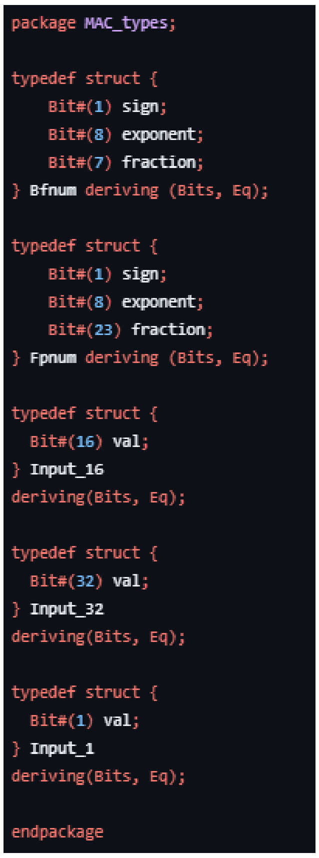
```
package systolic_array;

import FIFO::*;
import SpecialFIFOs::*;

import MAC_types ::*;
import MAC_with_wrapper ::*;
```

The above image shows the imports done.



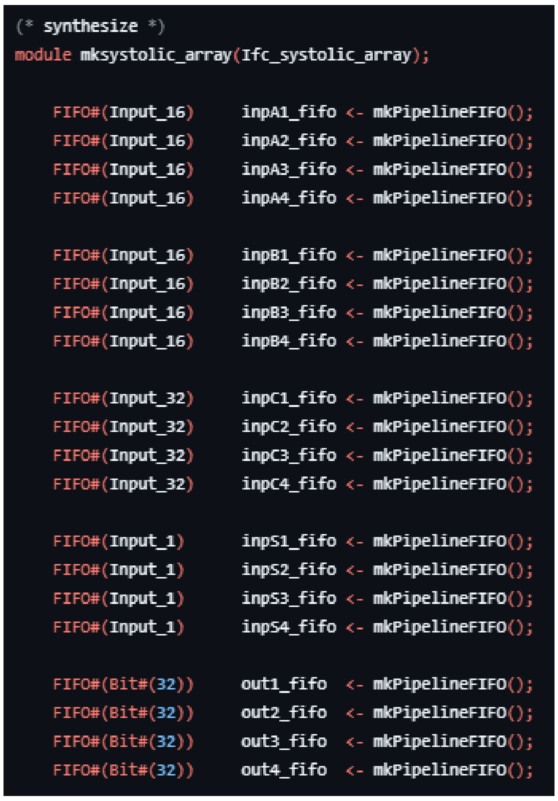
```
interface Ifc_systolic_array;
method Action get_A1(Input_16 a);
method Action get_A2(Input_16 a);
method Action get_A3(Input_16 a);
method Action get_A4(Input_16 a);

method Action get_B1(Input_16 b);
method Action get_B2(Input_16 b);
method Action get_B3(Input_16 b);
method Action get_B4(Input_16 b);

method Action get_C1(Input_32 c);
method Action get_C2(Input_32 c);
method Action get_C3(Input_32 c);
method Action get_C4(Input_32 c);

method Action get_S1(Input_1 s);
method Action get_S2(Input_1 s);
method Action get_S3(Input_1 s);
method Action get_S4(Input_1 s);

method ActionValue#(Bit#(32)) output1_MAC();
method ActionValue#(Bit#(32)) output2_MAC();
method ActionValue#(Bit#(32)) output3_MAC();
method ActionValue#(Bit#(32)) output4_MAC();

endinterface: Ifc_systolic_array
```

The above image shows the interface created for systolic array.

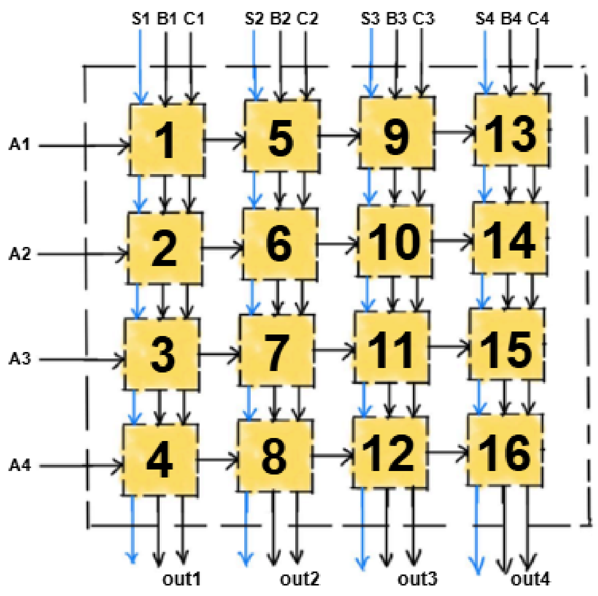


```
package MAC_types;

typedef struct {
    Bit#(1) sign;
    Bit#(8) exponent;
    Bit#(7) fraction;
} Bfnum deriving (Bits, Eq);

typedef struct {
    Bit#(1) sign;
    Bit#(8) exponent;
    Bit#(23) fraction;
} Fpnum deriving (Bits, Eq);

typedef struct {
  Bit#(16) val;
} Input_16
deriving(Bits, Eq);

typedef struct {
  Bit#(32) val;
} Input_32
deriving(Bits, Eq);

typedef struct {
  Bit#(1) val;
} Input_1
deriving(Bits, Eq);

endpackage
```

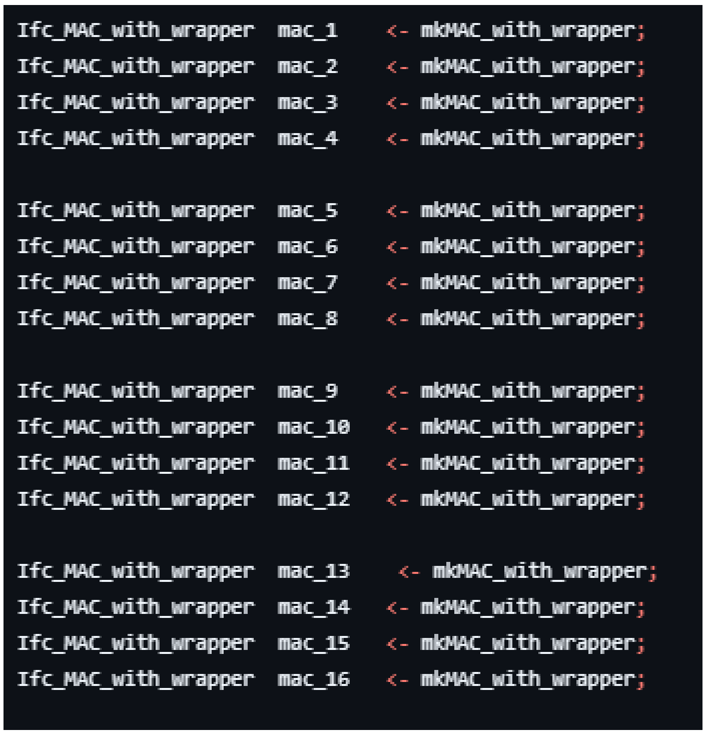
Input_16, Input_32 and Input_1 are defined in MAC_types as shown in the above image



```
(* synthesize *)
module mksystolic_array(Ifc_systolic_array);

    FIFO#(Input_16)     inpA1_fifo <- mkPipelineFIFO();
    FIFO#(Input_16)     inpA2_fifo <- mkPipelineFIFO();
    FIFO#(Input_16)     inpA3_fifo <- mkPipelineFIFO();
    FIFO#(Input_16)     inpA4_fifo <- mkPipelineFIFO();

    FIFO#(Input_16)     inpB1_fifo <- mkPipelineFIFO();
    FIFO#(Input_16)     inpB2_fifo <- mkPipelineFIFO();
    FIFO#(Input_16)     inpB3_fifo <- mkPipelineFIFO();
    FIFO#(Input_16)     inpB4_fifo <- mkPipelineFIFO();

    FIFO#(Input_32)     inpC1_fifo <- mkPipelineFIFO();
    FIFO#(Input_32)     inpC2_fifo <- mkPipelineFIFO();
    FIFO#(Input_32)     inpC3_fifo <- mkPipelineFIFO();
    FIFO#(Input_32)     inpC4_fifo <- mkPipelineFIFO();

    FIFO#(Input_1)      inpS1_fifo <- mkPipelineFIFO();
    FIFO#(Input_1)      inpS2_fifo <- mkPipelineFIFO();
    FIFO#(Input_1)      inpS3_fifo <- mkPipelineFIFO();
    FIFO#(Input_1)      inpS4_fifo <- mkPipelineFIFO();

    FIFO#(Bit#(32))     out1_fifo  <- mkPipelineFIFO();
    FIFO#(Bit#(32))     out2_fifo  <- mkPipelineFIFO();
    FIFO#(Bit#(32))     out3_fifo  <- mkPipelineFIFO();
    FIFO#(Bit#(32))     out4_fifo  <- mkPipelineFIFO();
```

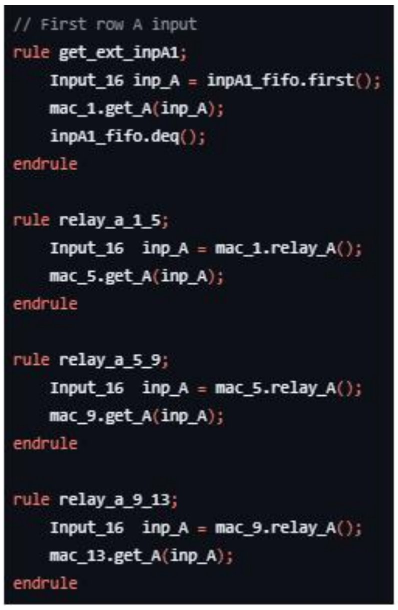


The above image shows the creation of FIFOs required for input and output of systolic array.

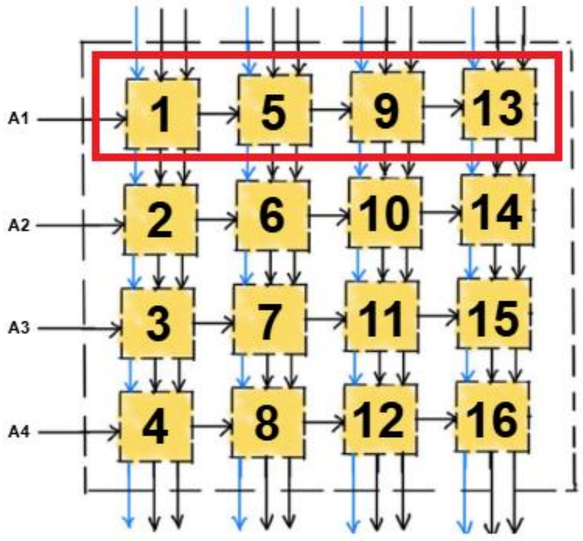


```
Ifc_MAC_with_wrapper  mac_1    <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_2    <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_3    <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_4    <- mkMAC_with_wrapper;

Ifc_MAC_with_wrapper  mac_5    <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_6    <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_7    <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_8    <- mkMAC_with_wrapper;

Ifc_MAC_with_wrapper  mac_9    <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_10   <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_11   <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_12   <- mkMAC_with_wrapper;

Ifc_MAC_with_wrapper  mac_13    <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_14   <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_15   <- mkMAC_with_wrapper;
Ifc_MAC_with_wrapper  mac_16   <- mkMAC_with_wrapper;
```

The above image shows the instantiation of sixteen MAC units.

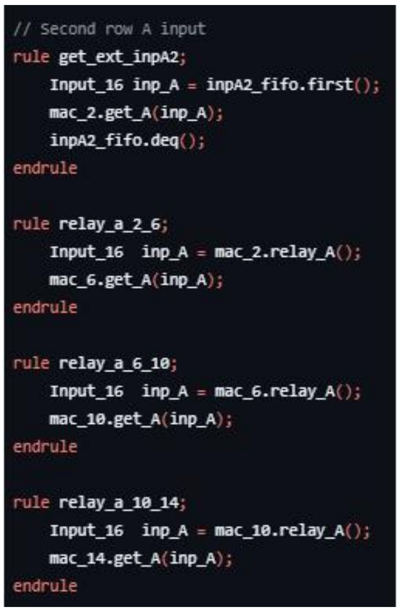


```
// First row A input
rule get_ext_inpA1;
    Input_16 inp_A = inpA1_fifo.first();
    mac_1.get_A(inp_A);
    inpA1_fifo.deq();
endrule

rule relay_a_1_5;
    Input_16  inp_A = mac_1.relay_A();
    mac_5.get_A(inp_A);
endrule

rule relay_a_5_9;
    Input_16  inp_A = mac_5.relay_A();
    mac_9.get_A(inp_A);
endrule

rule relay_a_9_13;
    Input_16  inp_A = mac_9.relay_A();
    mac_13.get_A(inp_A);
endrule
```

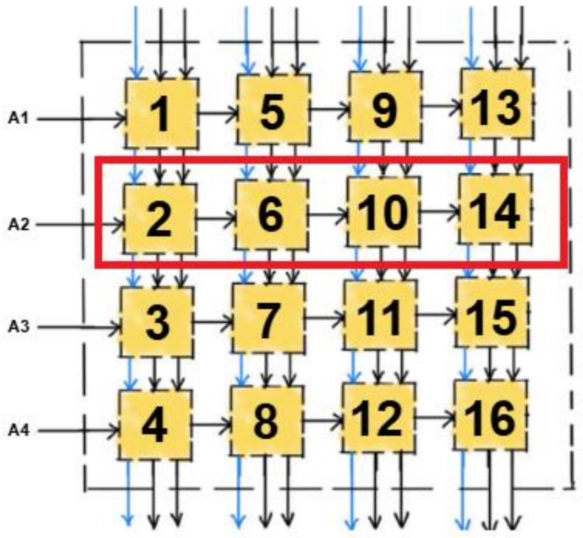


The above image shows the rules written which handles input A1 for first row of the array. The rule “get_ext_inpA1” gets input A1 value from inpA1 FIFO and gives it to get_A method of MAC_1. And it also dequeues inpA1 FIFO.

The rule “relay_a_1_5” relays the input A1 value from MAC_1 to MAC_5. The rule “relay_a_5_9” relays the input A1 value from MAC_5 to MAC_9. The rule “relay_a_9_13” relays the input A1 value from MAC_9 to MAC_13.

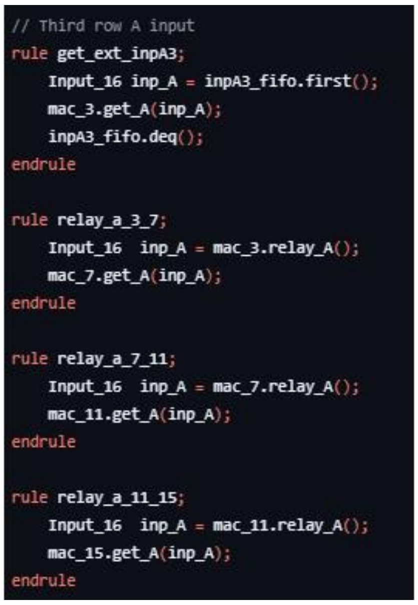


```
// Second row A input
rule get_ext_inpA2;
    Input_16 inp_A = inpA2_fifo.first();
    mac_2.get_A(inp_A);
    inpA2_fifo.deq();
endrule

rule relay_a_2_6;
    Input_16  inp_A = mac_2.relay_A();
    mac_6.get_A(inp_A);
endrule

rule relay_a_6_10;
    Input_16  inp_A = mac_6.relay_A();
    mac_10.get_A(inp_A);
endrule

rule relay_a_10_14;
    Input_16  inp_A = mac_10.relay_A();
    mac_14.get_A(inp_A);
endrule
```

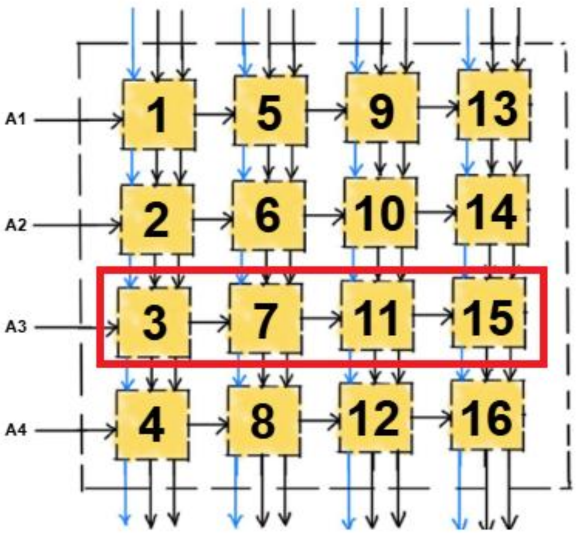


The above image shows the rules written which handles input A2 for second row of the array. The rule “get_ext_inpA2” gets input A2 value from inpA2 FIFO and gives it to get_A method of MAC_2. And it also dequeues inpA2 FIFO.

The rule “relay_a_2_6” relays the input A2 value from MAC_2 to MAC_6. The rule “relay_a_6_10” relays the input A2 value from MAC_6 to MAC_10. The rule “relay_a_10_14” relays the input A2 value from MAC_10 to MAC_14.

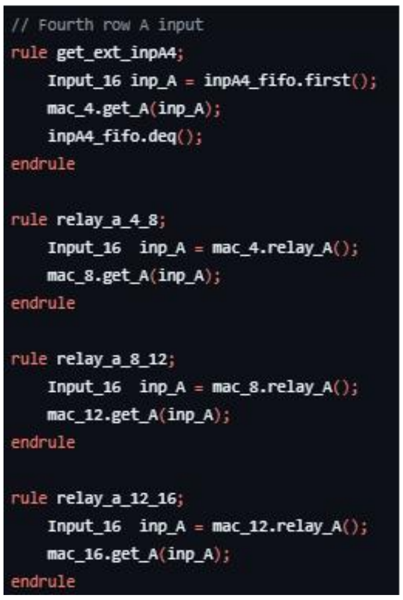


```
// Third row A input
rule get_ext_inpA3;
    Input_16 inp_A = inpA3_fifo.first();
    mac_3.get_A(inp_A);
    inpA3_fifo.deq();
endrule

rule relay_a_3_7;
    Input_16  inp_A = mac_3.relay_A();
    mac_7.get_A(inp_A);
endrule

rule relay_a_7_11;
    Input_16  inp_A = mac_7.relay_A();
    mac_11.get_A(inp_A);
endrule

rule relay_a_11_15;
    Input_16  inp_A = mac_11.relay_A();
    mac_15.get_A(inp_A);
endrule
```

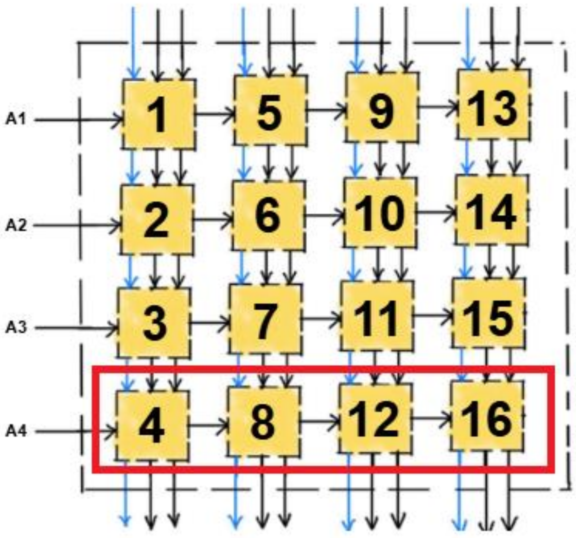


The above image shows the rules written which handles input A3 for third row of the array. The rule “get_ext_inpA3” gets input A3 value from inpA3 FIFO and gives it to get_A method of MAC_3. And it also dequeues inpA3 FIFO.

The rule “relay_a_3_7” relays the input A3 value from MAC_3 to MAC_7. The rule “relay_a_7_11” relays the input A3 value from MAC_7 to MAC_11. The rule “relay_a_11_15” relays the input A3 value from MAC_11 to MAC_15.

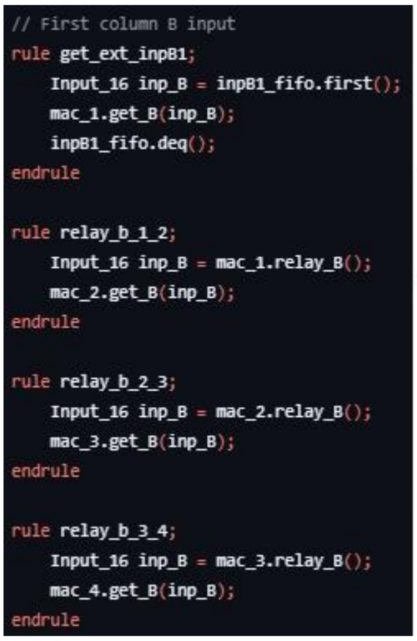


```
// Fourth row A input
rule get_ext_inpA4;
    Input_16 inp_A = inpA4_fifo.first();
    mac_4.get_A(inp_A);
    inpA4_fifo.deq();
endrule

rule relay_a_4_8;
    Input_16  inp_A = mac_4.relay_A();
    mac_8.get_A(inp_A);
endrule

rule relay_a_8_12;
    Input_16  inp_A = mac_8.relay_A();
    mac_12.get_A(inp_A);
endrule

rule relay_a_12_16;
    Input_16  inp_A = mac_12.relay_A();
    mac_16.get_A(inp_A);
endrule
```

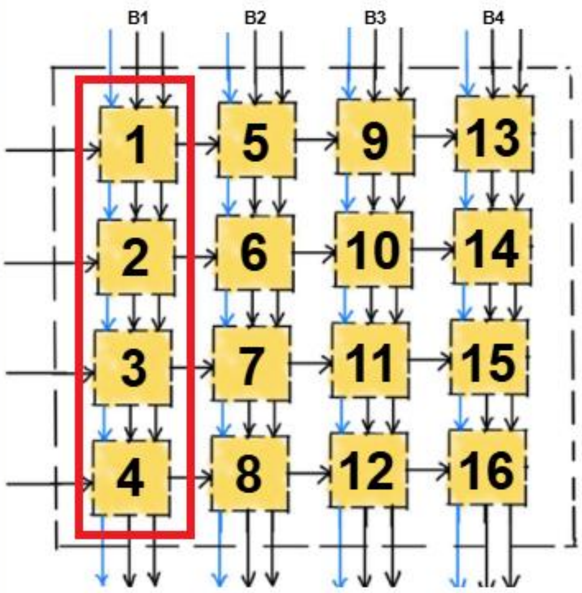


The above image shows the rules written which handles input A4 for fourth row of the array. The rule “get_ext_inpA4” gets input A4 value from inpA4 FIFO and gives it to get_A method of MAC_4. And it also dequeues inpA4 FIFO.

The rule “relay_a_4_8” relays the input A4 value from MAC_4 to MAC_8. The rule “relay_a_8_12” relays the input A4 value from MAC_8 to MAC_12. The rule “relay_a_12_16” relays the input A4 value from MAC_12 to MAC_16.

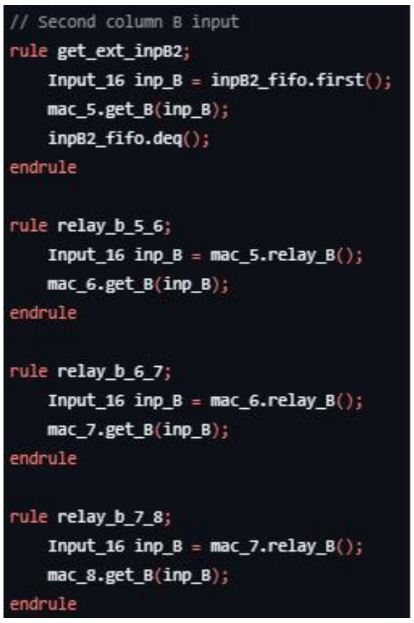


```
// First column B input
rule get_ext_inpB1;
    Input_16 inp_B = inpB1_fifo.first();
    mac_1.get_B(inp_B);
    inpB1_fifo.deq();
endrule

rule relay_b_1_2;
    Input_16 inp_B = mac_1.relay_B();
    mac_2.get_B(inp_B);
endrule

rule relay_b_2_3;
    Input_16 inp_B = mac_2.relay_B();
    mac_3.get_B(inp_B);
endrule

rule relay_b_3_4;
    Input_16 inp_B = mac_3.relay_B();
    mac_4.get_B(inp_B);
endrule
```

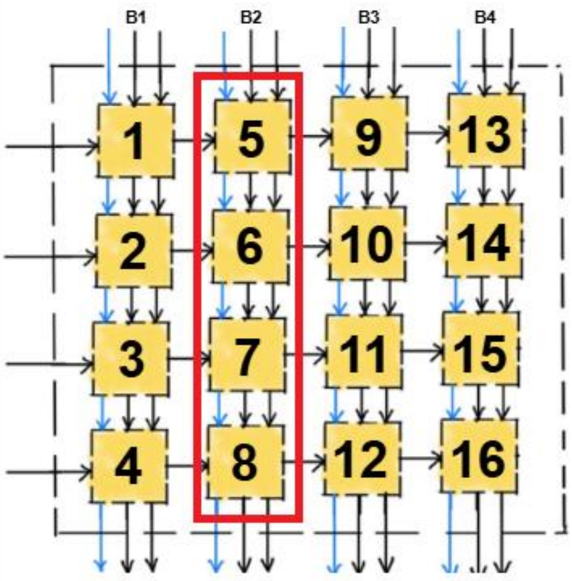


The above image shows the rules written which handles input B1 for first column of the array. The rule "get_ext_inpB1" gets input B1 value from inpB1 FIFO and gives it to get_B method of MAC_1. And it also dequeues inpB1 FIFO.

The rule "relay_b_1_2" relays the input B1 value from MAC_1 to MAC_2. The rule "relay_b_2_3" relays the input B1 value from MAC_2 to MAC_3. The rule "relay_b_3_4" relays the input B1 value from MAC_3 to MAC_4.

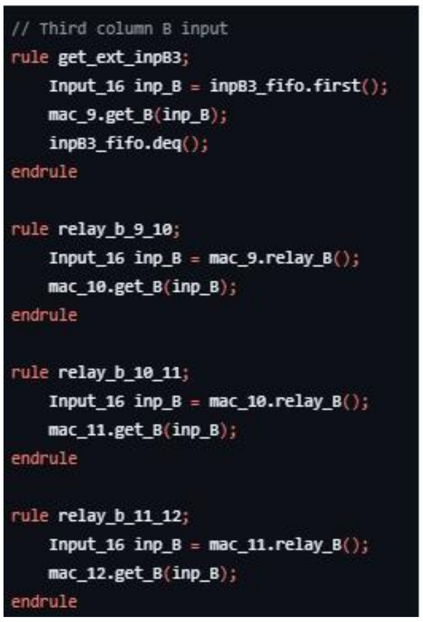


```
// Second column B input
rule get_ext_inpB2;
    Input_16 inp_B = inpB2_fifo.first();
    mac_5.get_B(inp_B);
    inpB2_fifo.deq();
endrule

rule relay_b_5_6;
    Input_16 inp_B = mac_5.relay_B();
    mac_6.get_B(inp_B);
endrule

rule relay_b_6_7;
    Input_16 inp_B = mac_6.relay_B();
    mac_7.get_B(inp_B);
endrule

rule relay_b_7_8;
    Input_16 inp_B = mac_7.relay_B();
    mac_8.get_B(inp_B);
endrule
```

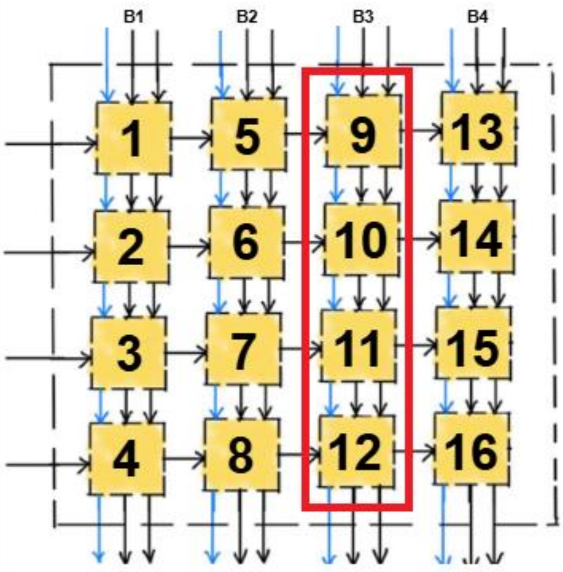


The above image shows the rules written which handles input B2 for second column of the array. The rule "get_ext_inpB2" gets input B2 value from inpB2 FIFO and gives it to get_B method of MAC_5. And it also dequeues inpB2 FIFO.

The rule "relay_b_5_6" relays the input B2 value from MAC_5 to MAC_6. The rule "relay_b_6_7" relays the input B2 value from MAC_6 to MAC_7. The rule "relay_b_7_8" relays the input B2 value from MAC_7 to MAC_8.

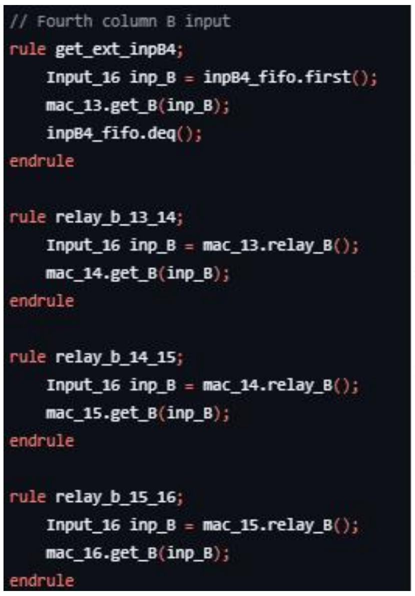


```
// Third column B input
rule get_ext_inpB3;
    Input_16 inp_B = inpB3_fifo.first();
    mac_9.get_B(inp_B);
    inpB3_fifo.deq();
endrule

rule relay_b_9_10;
    Input_16 inp_B = mac_9.relay_B();
    mac_10.get_B(inp_B);
endrule

rule relay_b_10_11;
    Input_16 inp_B = mac_10.relay_B();
    mac_11.get_B(inp_B);
endrule

rule relay_b_11_12;
    Input_16 inp_B = mac_11.relay_B();
    mac_12.get_B(inp_B);
endrule
```

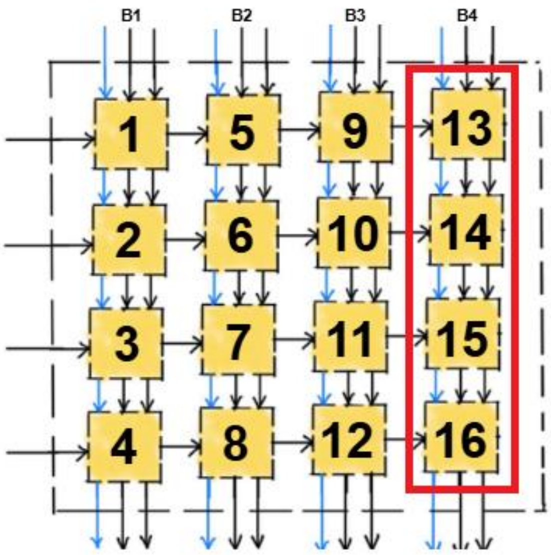


The above image shows the rules written which handles input B3 for third column of the array. The rule “get_ext_inpB3” gets input B3 value from inpB3 FIFO and gives it to get_B method of MAC_9. And it also dequeues inpB3 FIFO.

The rule “relay_b_9_10” relays the input B3 value from MAC_9 to MAC_10. The rule “relay_b_10_11” relays the input B3 value from MAC_10 to MAC_11. The rule “relay_b_11_12” relays the input B3 value from MAC_11 to MAC_12.

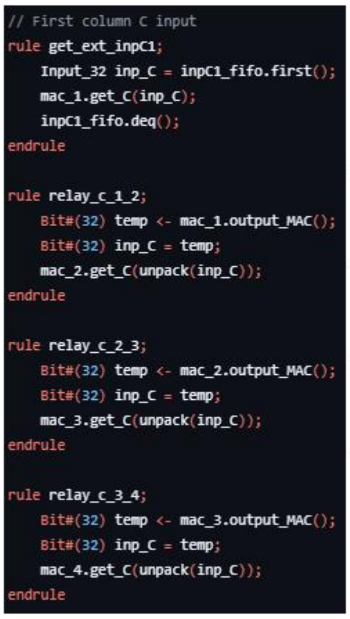


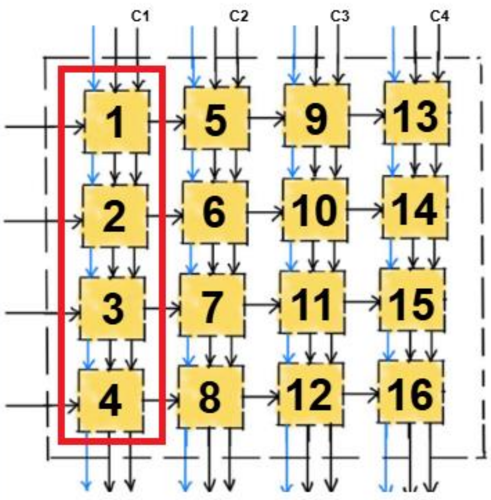
```
// Fourth column B input
rule get_ext_inpB4;
    Input_16 inp_B = inpB4_fifo.first();
    mac_13.get_B(inp_B);
    inpB4_fifo.deq();
endrule

rule relay_b_13_14;
    Input_16 inp_B = mac_13.relay_B();
    mac_14.get_B(inp_B);
endrule

rule relay_b_14_15;
    Input_16 inp_B = mac_14.relay_B();
    mac_15.get_B(inp_B);
endrule

rule relay_b_15_16;
    Input_16 inp_B = mac_15.relay_B();
    mac_16.get_B(inp_B);
endrule
```



The above image shows the rules written which handles input B4 for fourth column of the array. The rule “get_ext_inpB4” gets input B4 value from inpB4 FIFO and gives it to get_B method of MAC_13. And it also dequeues inpB4 FIFO.

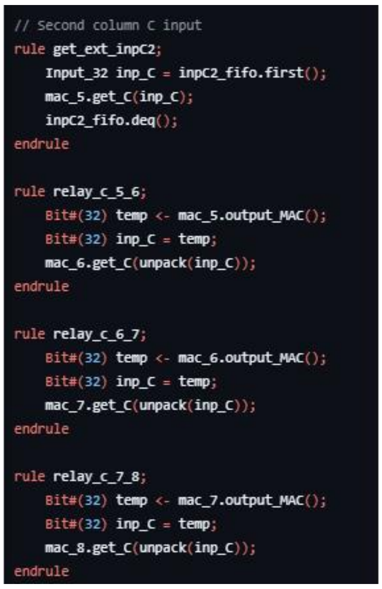
The rule “relay_b_13_14” relays the input B4 value from MAC_13 to MAC_14. The rule “relay_b_14_15” relays the input B4 value from MAC_14 to MAC_15. The rule “relay_b_15_16” relays the input B4 value from MAC_15 to MAC_16.

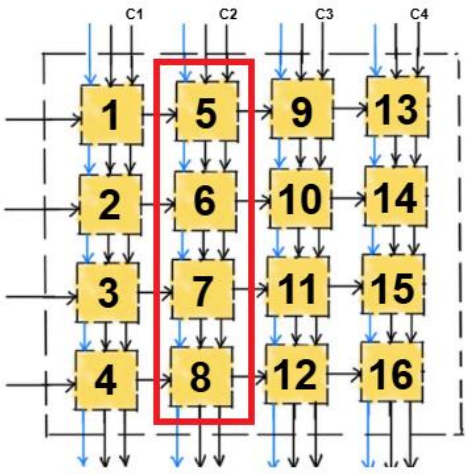

```
// First column C input
rule get_ext_inpC1;
    Input_32 inp_C = inpC1_fifo.first();
    mac_1.get_C(inp_C);
    inpC1_fifo.deq();
endrule

rule relay_c_1_2;
    Bit#(32) temp <- mac_1.output_MAC();
    Bit#(32) inp_C = temp;
    mac_2.get_C(unpack(inp_C));
endrule

rule relay_c_2_3;
    Bit#(32) temp <- mac_2.output_MAC();
    Bit#(32) inp_C = temp;
    mac_3.get_C(unpack(inp_C));
endrule

rule relay_c_3_4;
    Bit#(32) temp <- mac_3.output_MAC();
    Bit#(32) inp_C = temp;
    mac_4.get_C(unpack(inp_C));
endrule
```



The above image shows the rules written which handles input C1 for first column of the array. The rule “get_ext_inpC1” gets input C1 value from inpC1 FIFO and gives it to get_C method of MAC_1. And it also dequeues inpC1 FIFO.

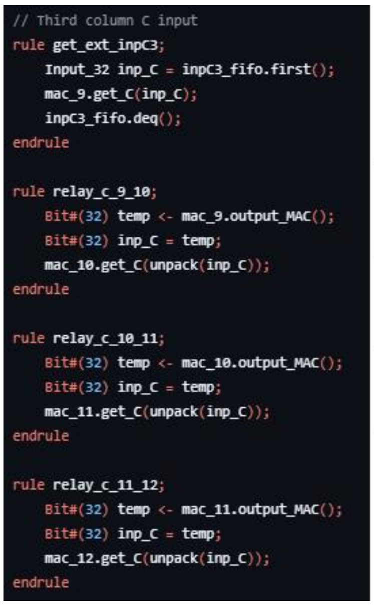
The rule “relay_c_1_2” relays the input C1 value from MAC_1 to MAC_2. The rule “relay_c_2_3” relays the input C1 value from MAC_2 to MAC_3. The rule “relay_c_3_4” relays the input C1 value from MAC_3 to MAC_4.


```
// Second column C input
rule get_ext_inpC2;
    Input_32 inp_C = inpC2_fifo.first();
    mac_5.get_C(inp_C);
    inpC2_fifo.deq();
endrule

rule relay_c_5_6;
    Bit#(32) temp <- mac_5.output_MAC();
    Bit#(32) inp_C = temp;
    mac_6.get_C(unpack(inp_C));
endrule

rule relay_c_6_7;
    Bit#(32) temp <- mac_6.output_MAC();
    Bit#(32) inp_C = temp;
    mac_7.get_C(unpack(inp_C));
endrule

rule relay_c_7_8;
    Bit#(32) temp <- mac_7.output_MAC();
    Bit#(32) inp_C = temp;
    mac_8.get_C(unpack(inp_C));
endrule
```

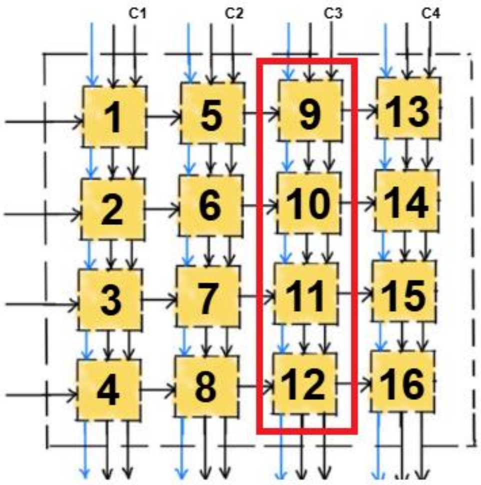


The above image shows the rules written which handles input C2 for second column of the array. The rule “get_ext_inpC2” gets input C2 value from inpC2 FIFO and gives it to get_C method of MAC_5. And it also dequeues inpC2 FIFO.

The rule “relay_c_5_6” relays the input C2 value from MAC_5 to MAC_6. The rule “relay_c_6_7” relays the input C2 value from MAC_6 to MAC_7. The rule “relay_c_7_8” relays the input C2 value from MAC_7 to MAC_8.

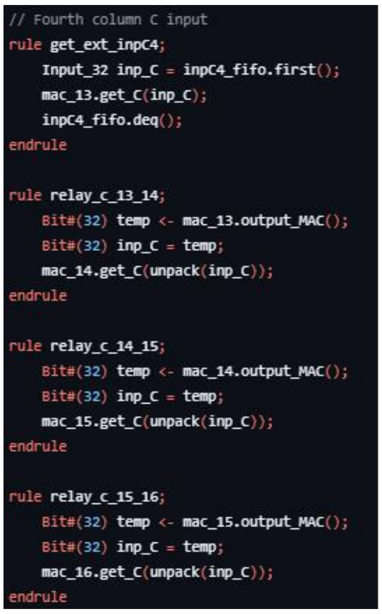


```
// Third column C input
rule get_ext_inpC3;
    Input_32 inp_C = inpC3_fifo.first();
    mac_9.get_C(inp_C);
    inpC3_fifo.deq();
endrule

rule relay_c_9_10;
    Bit#(32) temp <- mac_9.output_MAC();
    Bit#(32) inp_C = temp;
    mac_10.get_C(unpack(inp_C));
endrule

rule relay_c_10_11;
    Bit#(32) temp <- mac_10.output_MAC();
    Bit#(32) inp_C = temp;
    mac_11.get_C(unpack(inp_C));
endrule

rule relay_c_11_12;
    Bit#(32) temp <- mac_11.output_MAC();
    Bit#(32) inp_C = temp;
    mac_12.get_C(unpack(inp_C));
endrule
```

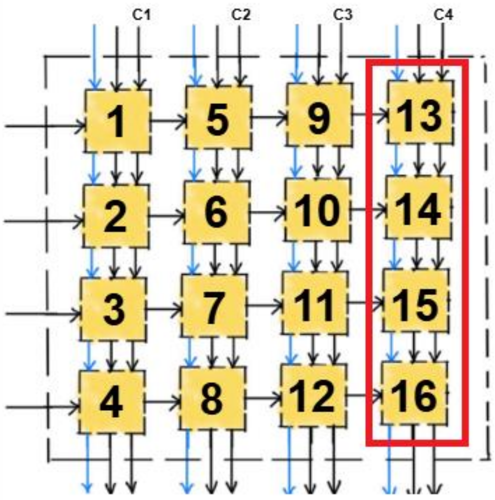


The above image shows the rules written which handles input C3 for third column of the array. The rule “get_ext_inpC3” gets input C3 value from inpC3 FIFO and gives it to get_C method of MAC_9. And it also dequeues inpC3 FIFO.

The rule “relay_c_9_10” relays the input C3 value from MAC_9 to MAC_10. The rule “relay_c_10_11” relays the input C3 value from MAC_10 to MAC_11. The rule “relay_c_11_12” relays the input C3 value from MAC_11 to MAC_12.

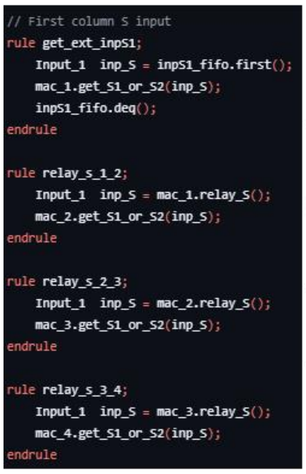


```
// Fourth column C input
rule get_ext_inpC4;
    Input_32 inp_C = inpC4_fifo.first();
    mac_13.get_C(inp_C);
    inpC4_fifo.deq();
endrule

rule relay_c_13_14;
    Bit#(32) temp <- mac_13.output_MAC();
    Bit#(32) inp_C = temp;
    mac_14.get_C(unpack(inp_C));
endrule

rule relay_c_14_15;
    Bit#(32) temp <- mac_14.output_MAC();
    Bit#(32) inp_C = temp;
    mac_15.get_C(unpack(inp_C));
endrule

rule relay_c_15_16;
    Bit#(32) temp <- mac_15.output_MAC();
    Bit#(32) inp_C = temp;
    mac_16.get_C(unpack(inp_C));
endrule
```

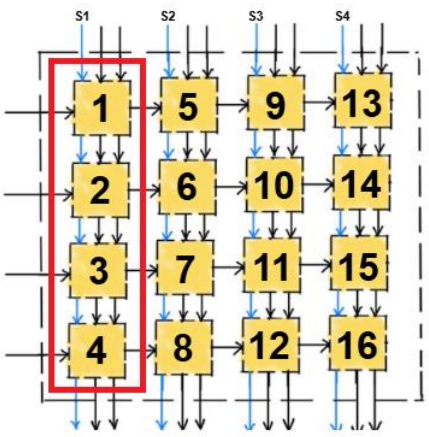


The above image shows the rules written which handles input C4 for fourth column of the array. The rule “get_ext_inpC4” gets input C4 value from inpC4 FIFO and gives it to get_C method of MAC_13. And it also dequeues inpC4 FIFO.

The rule “relay_c_13_14” relays the input C4 value from MAC_13 to MAC_14. The rule “relay_c_14_15” relays the input C4 value from MAC_14 to MAC_15. The rule “relay_c_15_16” relays the input C4 value from MAC_15 to MAC_16.

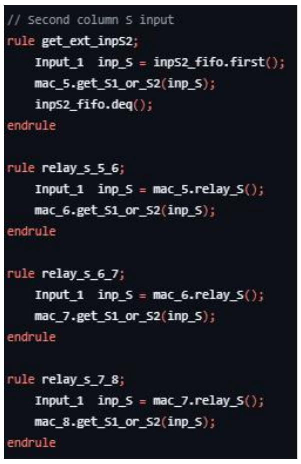


```
// First column S input
rule get_ext_inpS1;
    Input_1  inp_S = inpS1_fifo.first();
    mac_1.get_S1_or_S2(inp_S);
    inpS1_fifo.deq();
endrule

rule relay_s_1_2;
    Input_1  inp_S = mac_1.relay_S();
    mac_2.get_S1_or_S2(inp_S);
endrule

rule relay_s_2_3;
    Input_1  inp_S = mac_2.relay_S();
    mac_3.get_S1_or_S2(inp_S);
endrule

rule relay_s_3_4;
    Input_1  inp_S = mac_3.relay_S();
    mac_4.get_S1_or_S2(inp_S);
endrule
```

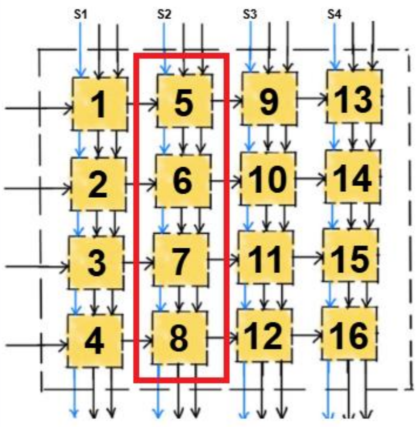


The above image shows the rules written which handles input S1 for first column of the array. The rule “get_ext_inpS1” gets input S1 value from inpS1 FIFO and gives it to get_S1_or_S2 method of MAC_1. And it also dequeues inpS1 FIFO.

The rule “relay_s_1_2” relays the input S1 value from MAC_1 to MAC_2. The rule “relay_s_2_3” relays the input S1 value from MAC_2 to MAC_3. The rule “relay_s_3_4” relays the input S1 value from MAC_3 to MAC_4.

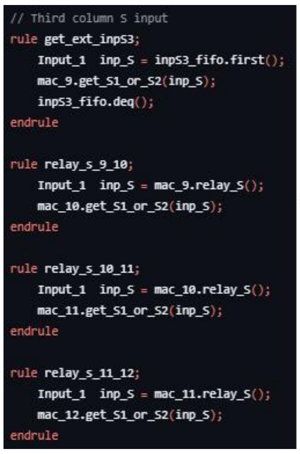


```
// Second column S input
rule get_ext_inpS2;
    Input_1  inp_S = inpS2_fifo.first();
    mac_5.get_S1_or_S2(inp_S);
    inpS2_fifo.deq();
endrule

rule relay_s_5_6;
    Input_1  inp_S = mac_5.relay_S();
    mac_6.get_S1_or_S2(inp_S);
endrule

rule relay_s_6_7;
    Input_1  inp_S = mac_6.relay_S();
    mac_7.get_S1_or_S2(inp_S);
endrule

rule relay_s_7_8;
    Input_1  inp_S = mac_7.relay_S();
    mac_8.get_S1_or_S2(inp_S);
endrule
```

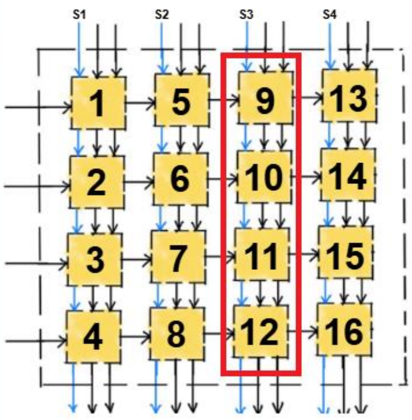


The above image shows the rules written which handles input S2 for second column of the array. The rule “get_ext_inpS2” gets input S2 value from inpS2 FIFO and gives it to get_S1_or_S2 method of MAC_5. And it also dequeues inpS2 FIFO.

The rule “relay_s_5_6” relays the input S2 value from MAC_5 to MAC_6. The rule “relay_s_6_7” relays the input S2 value from MAC_6 to MAC_7. The rule “relay_s_7_8” relays the input S2 value from MAC_7 to MAC_8.

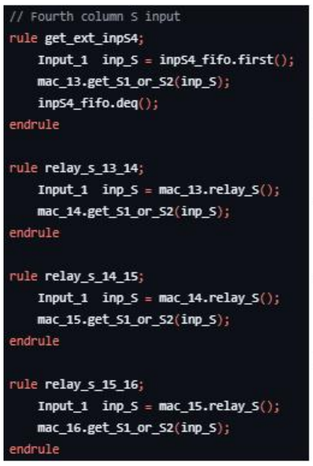


```
// Third column S input
rule get_ext_inpS3;
    Input_1  inp_S = inpS3_fifo.first();
    mac_9.get_S1_or_S2(inp_S);
    inpS3_fifo.deq();
endrule

rule relay_s_9_10;
    Input_1  inp_S = mac_9.relay_S();
    mac_10.get_S1_or_S2(inp_S);
endrule

rule relay_s_10_11;
    Input_1  inp_S = mac_10.relay_S();
    mac_11.get_S1_or_S2(inp_S);
endrule

rule relay_s_11_12;
    Input_1  inp_S = mac_11.relay_S();
    mac_12.get_S1_or_S2(inp_S);
endrule
```

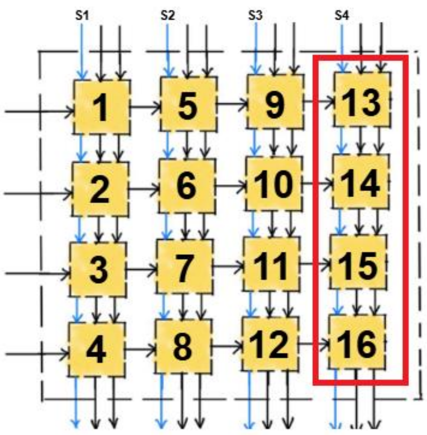


The above image shows the rules written which handles input S3 for third column of the array. The rule “get_ext_inpS3” gets input S3 value from inpS3 FIFO and gives it to get_S1_or_S2 method of MAC_9. And it also dequeues inpS3 FIFO.

The rule “relay_s_9_10” relays the input S3 value from MAC_9 to MAC_10. The rule “relay_s_10_11” relays the input S3 value from MAC_10 to MAC_11. The rule “relay_s_11_12” relays the input S3 value from MAC_11 to MAC_12.

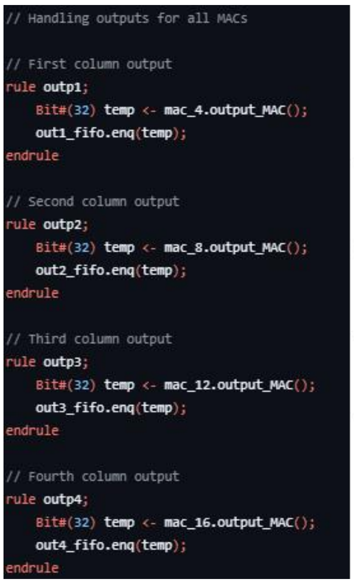


```
// Fourth column S input
rule get_ext_inpS4;
    Input_1  inp_S = inpS4_fifo.first();
    mac_13.get_S1_or_S2(inp_S);
    inpS4_fifo.deq();
endrule

rule relay_s_13_14;
    Input_1  inp_S = mac_13.relay_S();
    mac_14.get_S1_or_S2(inp_S);
endrule

rule relay_s_14_15;
    Input_1  inp_S = mac_14.relay_S();
    mac_15.get_S1_or_S2(inp_S);
endrule

rule relay_s_15_16;
    Input_1  inp_S = mac_15.relay_S();
    mac_16.get_S1_or_S2(inp_S);
endrule
```

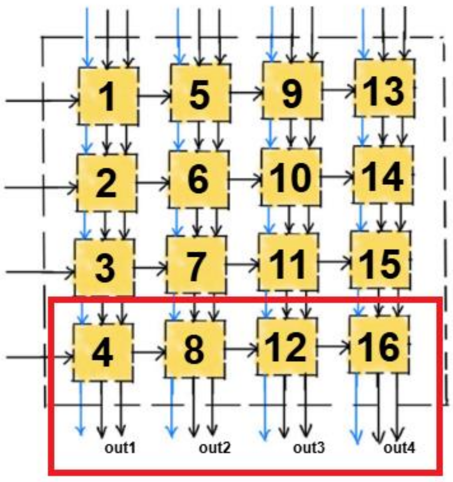


The above image shows the rules written which handles input S4 for fourth column of the array. The rule “get_ext_inpS4” gets input S4 value from inpS4 FIFO and gives it to get_S1_or_S2 method of MAC_13. And it also dequeues inpS4 FIFO.

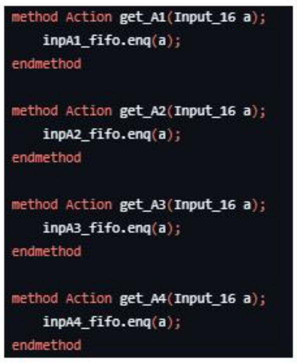
The rule “relay_s_13_14” relays the input S4 value from MAC_13 to MAC_14. The rule “relay_s_14_15” relays the input S4 value from MAC_14 to MAC_15. The rule “relay_s_15_16” relays the input S4 value from MAC_15 to MAC_16.



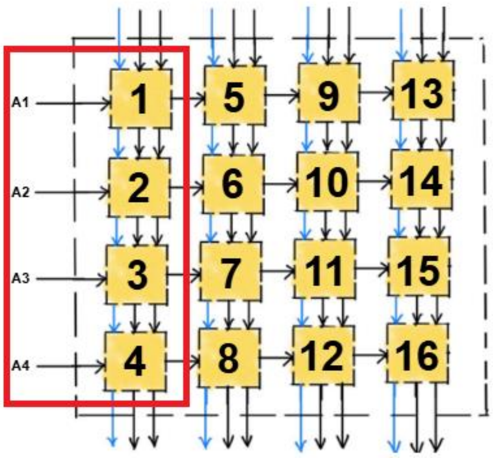
```
// Handling outputs for all MACs

// First column output
rule outp1;
    Bit#(32) temp <- mac_4.output_MAC();
    out1_fifo.enq(temp);
endrule

// Second column output
rule outp2;
    Bit#(32) temp <- mac_8.output_MAC();
    out2_fifo.enq(temp);
endrule

// Third column output
rule outp3;
    Bit#(32) temp <- mac_12.output_MAC();
    out3_fifo.enq(temp);
endrule

// Fourth column output
rule outp4;
    Bit#(32) temp <- mac_16.output_MAC();
    out4_fifo.enq(temp);
endrule
```



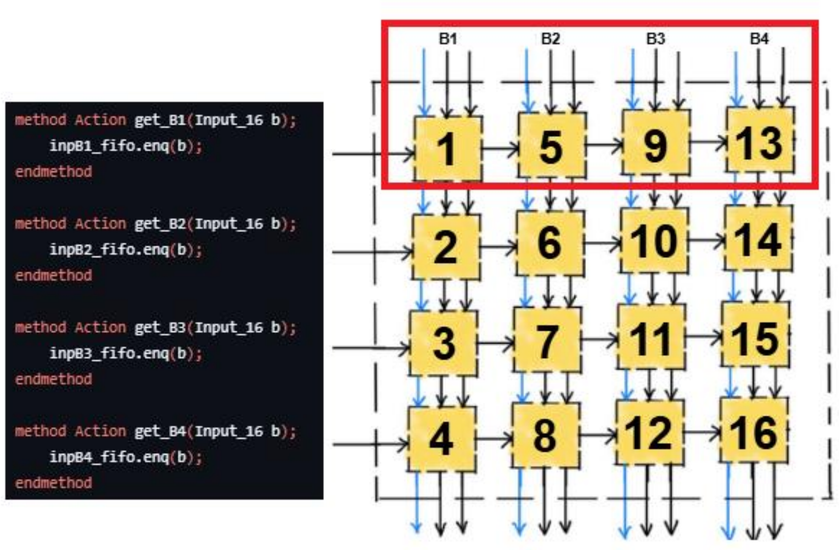
The above image shows the rules “outp1”, “outp2”, “outp3” and “outp4” which are responsible for collecting the first, second, third and fourth column outputs in out1 FIFO, out2 FIFO, out3 FIFO and out4 FIFO respectively.



```
method Action get_A1(Input_16 a);
    inpA1_fifo.enq(a);
endmethod

method Action get_A2(Input_16 a);
    inpA2_fifo.enq(a);
endmethod

method Action get_A3(Input_16 a);
    inpA3_fifo.enq(a);
endmethod

method Action get_A4(Input_16 a);
    inpA4_fifo.enq(a);
endmethod
```

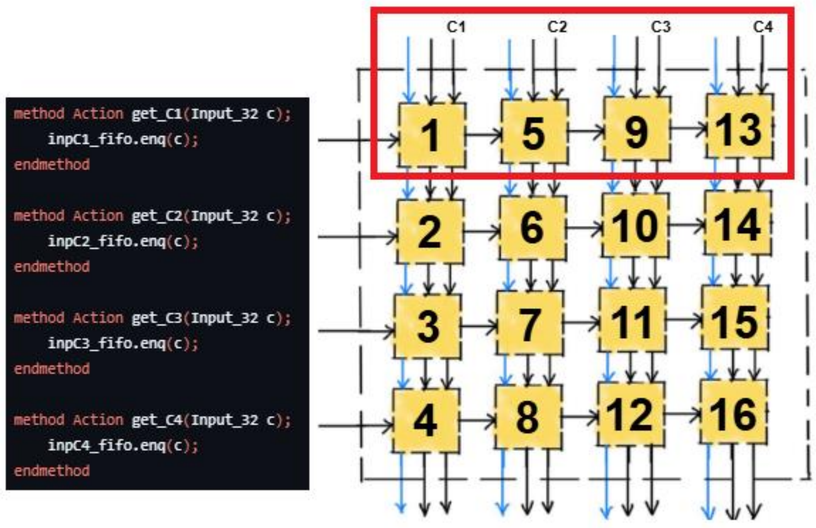


The above image shows the methods used for getting the inputs A1, A2, A3, A4 and loading it into the FIFOs inpA1, inpA2, inpA3 and inpA4 respectively.



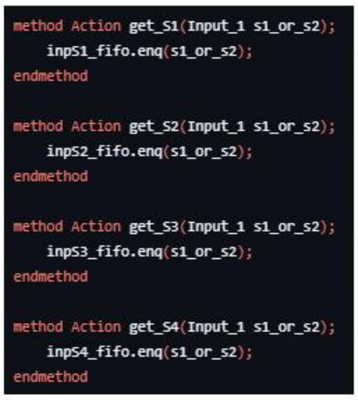
```
method Action get_B1(Input_16 b);
    inpB1_fifo.enq(b);
endmethod

method Action get_B2(Input_16 b);
    inpB2_fifo.enq(b);
endmethod

method Action get_B3(Input_16 b);
    inpB3_fifo.enq(b);
endmethod

method Action get_B4(Input_16 b);
    inpB4_fifo.enq(b);
endmethod
```

The above image shows the methods used for getting the inputs B1, B2, B3, B4 and loading it into the FIFOs inpB1, inpB2, inpB3 and inpB4 respectively.

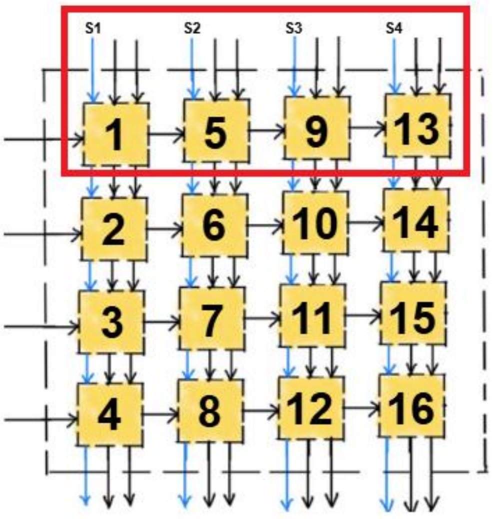


```
method Action get_C1(Input_32 c);
    inpC1_fifo.enq(c);
endmethod

method Action get_C2(Input_32 c);
    inpC2_fifo.enq(c);
endmethod

method Action get_C3(Input_32 c);
    inpC3_fifo.enq(c);
endmethod

method Action get_C4(Input_32 c);
    inpC4_fifo.enq(c);
endmethod
```

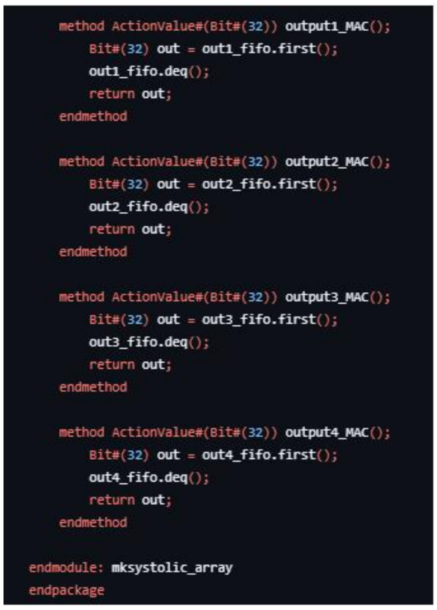
The above image shows the methods used for getting the inputs C1, C2, C3, C4 and loading it into the FIFOs inpC1, inpC2, inpC3 and inpC4 respectively.



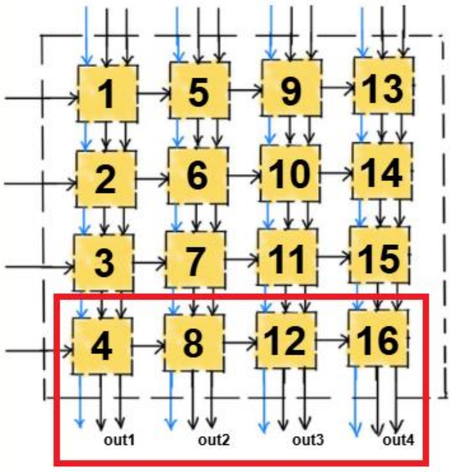
```
method Action get_S1(Input_1 s1_or_s2);
    inpS1_fifo.enq(s1_or_s2);
endmethod

method Action get_S2(Input_1 s1_or_s2);
    inpS2_fifo.enq(s1_or_s2);
endmethod

method Action get_S3(Input_1 s1_or_s2);
    inpS3_fifo.enq(s1_or_s2);
endmethod

method Action get_S4(Input_1 s1_or_s2);
    inpS4_fifo.enq(s1_or_s2);
endmethod
```



The above image shows the methods used for getting the inputs S1, S2, S3, S4 and loading it into the FIFOs inpS1, inpS2, inpS3 and inpS4 respectively.



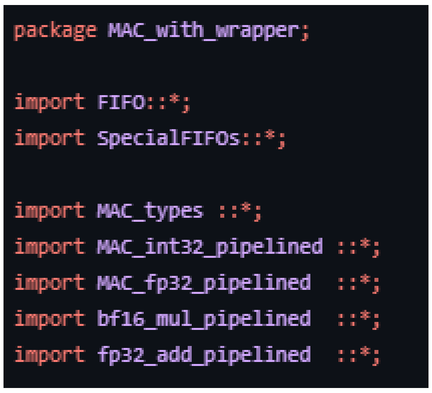
```
    method ActionValue#(Bit#(32)) output1_MAC();
        Bit#(32) out = out1_fifo.first();
        out1_fifo.deq();
        return out;
    endmethod

    method ActionValue#(Bit#(32)) output2_MAC();
        Bit#(32) out = out2_fifo.first();
        out2_fifo.deq();
        return out;
    endmethod

    method ActionValue#(Bit#(32)) output3_MAC();
        Bit#(32) out = out3_fifo.first();
        out3_fifo.deq();
        return out;
    endmethod

    method ActionValue#(Bit#(32)) output4_MAC();
        Bit#(32) out = out4_fifo.first();
        out4_fifo.deq();
        return out;
    endmethod

endmodule: mksystolic_array
endpackage
```



The above image shows the methods used for getting the outputs from the FIFOs out1 FIFO, out2 FIFO, out3 FIFO, out4 FIFO and returning it to outer world.

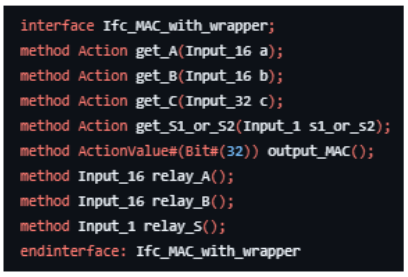
MAC_with_wrapper.bsv:

In this section, a single MAC unit is explained

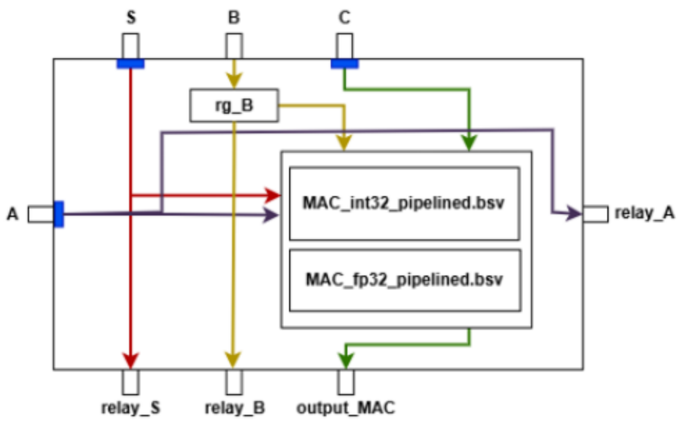


```
package MAC_with_wrapper;

import FIFO::*;
import SpecialFIFOs::*;

import MAC_types ::*;
import MAC_int32_pipelined ::*;
import MAC_fp32_pipelined  ::*;
import bf16_mul_pipelined  ::*;
import fp32_add_pipelined  ::*;
```

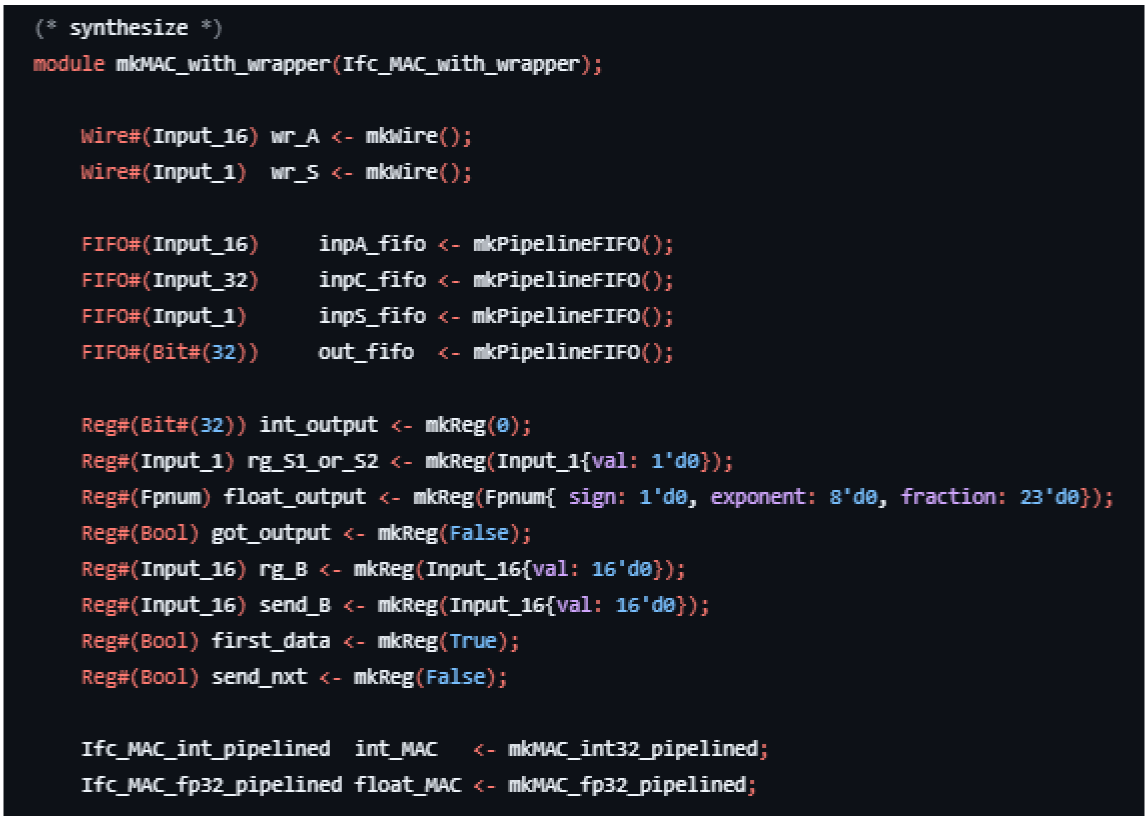
The above image shows the imports done.



```
interface Ifc_MAC_with_wrapper;
method Action get_A(Input_16 a);
method Action get_B(Input_16 b);
method Action get_C(Input_32 c);
method Action get_S1_or_S2(Input_1 s1_or_s2);
method ActionValue#(Bit#(32)) output_MAC();
method Input_16 relay_A();
method Input_16 relay_B();
method Input_1 relay_S();
endinterface: Ifc_MAC_with_wrapper
```



The above image shows the interface created.



```
(* synthesize *)
module mkMAC_with_wrapper(Ifc_MAC_with_wrapper);

    Wire#(Input_16) wr_A <- mkWire();
    Wire#(Input_1)  wr_S <- mkWire();

    FIFO#(Input_16)     inpA_fifo <- mkPipelineFIFO();
    FIFO#(Input_32)     inpC_fifo <- mkPipelineFIFO();
    FIFO#(Input_1)      inpS_fifo <- mkPipelineFIFO();
    FIFO#(Bit#(32))     out_fifo  <- mkPipelineFIFO();

    Reg#(Bit#(32)) int_output <- mkReg(0);
    Reg#(Input_1) rg_S1_or_S2 <- mkReg(Input_1{val: 1'd0});
    Reg#(Fpnum) float_output <- mkReg(Fpnum{ sign: 1'd0, exponent: 8'd0, fraction: 23'd0});
    Reg#(Bool) got_output <- mkReg(False);
    Reg#(Input_16) rg_B <- mkReg(Input_16{val: 16'd0});
    Reg#(Input_16) send_B <- mkReg(Input_16{val: 16'd0});
    Reg#(Bool) first_data <- mkReg(True);
    Reg#(Bool) send_nxt <- mkReg(False);

    Ifc_MAC_int_pipelined  int_MAC   <- mkMAC_int32_pipelined;
    Ifc_MAC_fp32_pipelined float_MAC <- mkMAC_fp32_pipelined;
```

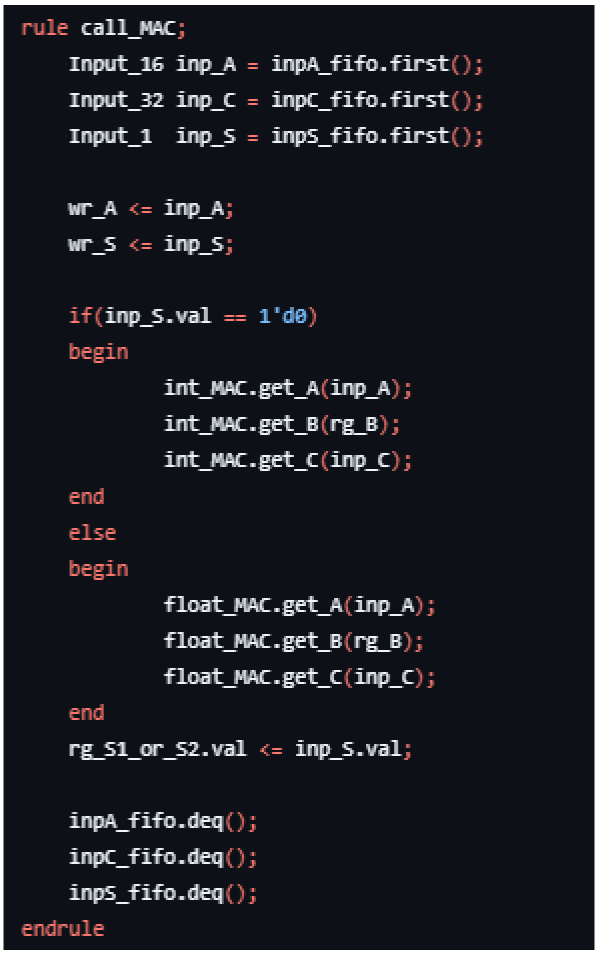
The above image shows the wires, FIFOs, registers and module instantiation of int and float MAC.

The wires wr_A and wr_S are means in which the values get relayed to other MAC units.

inpA_fifo, inpC_fifo and inpS_fifo are input FIFOs which will obtain the values of A,C and S from previous MAC unit stage. out_fifo carries the output to next stage of MAC unit.

There are a set of internal registers being created whose purpose will be explained along with rules in upcoming sections.

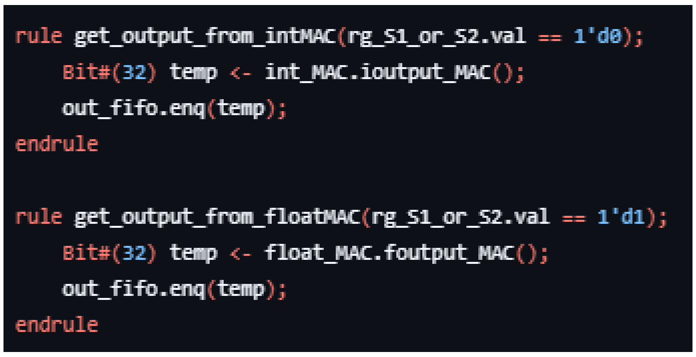
Int and float MAC units are instantiated.



```
rule call_MAC;
    Input_16 inp_A = inpA_fifo.first();
    Input_32 inp_C = inpC_fifo.first();
    Input_1  inp_S = inpS_fifo.first();

    wr_A <= inp_A;
    wr_S <= inp_S;

    if(inp_S.val == 1'd0)
    begin
            int_MAC.get_A(inp_A);
            int_MAC.get_B(rg_B);
            int_MAC.get_C(inp_C);
    end
    else
    begin
            float_MAC.get_A(inp_A);
            float_MAC.get_B(rg_B);
            float_MAC.get_C(inp_C);
    end
    rg_S1_or_S2.val <= inp_S.val;

    inpA_fifo.deq();
    inpC_fifo.deq();
    inpS_fifo.deq();
endrule
```

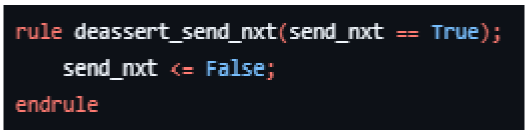
The above image shows the actions performed by the rule “call_MAC”. This rule is responsible for providing inputs to MAC.

According to value of S, inputs are given to int MAC or float MAC from FIFOs. Then the FIFOs are dequeued. The value of A and S are also passed through the wires wr_A and wr_S.



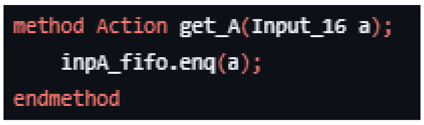
```
rule get_output_from_intMAC(rg_S1_or_S2.val == 1'd0);
    Bit#(32) temp <- int_MAC.ioutput_MAC();
    out_fifo.enq(temp);
endrule

rule get_output_from_floatMAC(rg_S1_or_S2.val == 1'd1);
    Bit#(32) temp <- float_MAC.foutput_MAC();
    out_fifo.enq(temp);
endrule
```

The above image shows two rules which are responsible for getting the output from int MAC and float MAC respectively and enqueueing it into out_fifo.



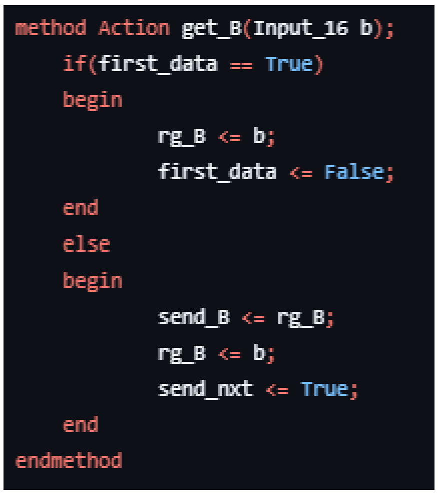
```
rule deassert_send_nxt(send_nxt == True);
    send_nxt <= False;
endrule
```

The above image shows the rule “deassert_send_nxt” whose purpose is to make the value of the register “send_nxt” to be false whenever it has the value as true.



```
method Action get_A(Input_16 a);
    inpA_fifo.enq(a);
endmethod
```

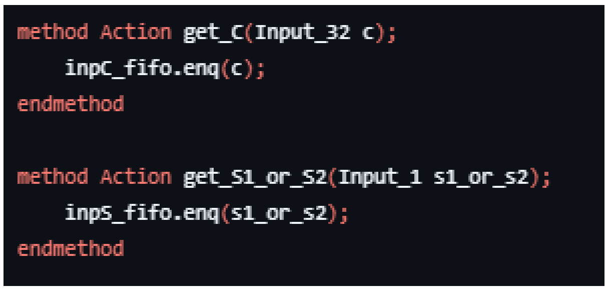
The above image shows how the method get_A functions. It just enqueues the value of input A into FIFO inpA_fifo.



```
method Action get_B(Input_16 b);
    if(first_data == True)
    begin
            rg_B <= b;
            first_data <= False;
    end
    else
    begin
            send_B <= rg_B;
            rg_B <= b;
            send_nxt <= True;
    end
endmethod
```

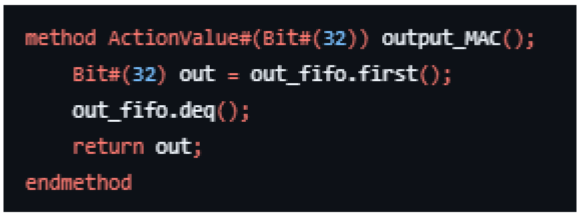
The above image shows how the method get_B functions. When the register “first_data” is true (Which is true by default on reset), value of b is just loaded into rg_B and first_data is set to false.

If first_data is false, current value of rg_B is put in send_B register, new value of b is loaded into rg_B and the register “send_nxt” is set to true.



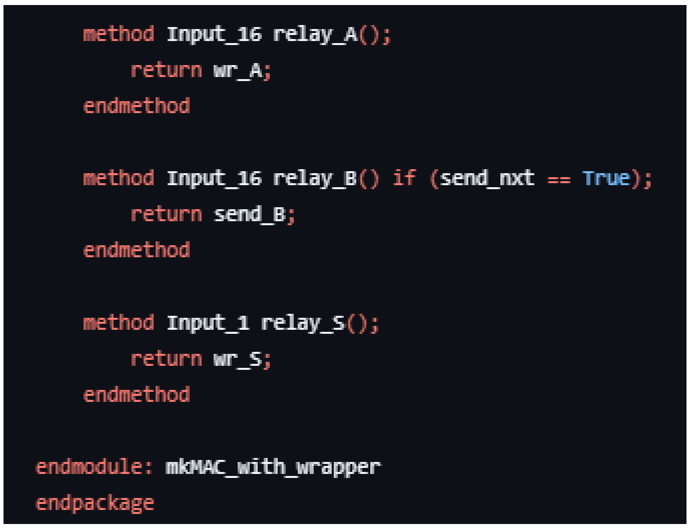
```
method Action get_C(Input_32 c);
    inpC_fifo.enq(c);
endmethod

method Action get_S1_or_S2(Input_1 s1_or_s2);
    inpS_fifo.enq(s1_or_s2);
endmethod
```

The above image shows the rest of the methods which handles the remaining of the inputs, namely C and S. The values of C and S are enqueued in the FIFOs inpC_fifo and inpS_fifo respectively.



```
method ActionValue#(Bit#(32)) output_MAC();
    Bit#(32) out = out_fifo.first();
    out_fifo.deq();
    return out;
endmethod
```

The above image shows the method which returns the output by dequeuing it from FIFO.



```
     method Input_16 relay_A();
         return wr_A;
     endmethod

     method Input_16 relay_B() if (send_nxt == True);
         return send_B;
     endmethod

     method Input_1 relay_S();
         return wr_S;
     endmethod

 endmodule: mkMAC_with_wrapper
 endpackage
```

The above image shows three methods responsible for relaying the inputs A, B and S to subsequent stages of MAC units.

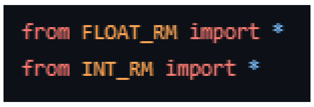
Relay_A and relay_S just return the value present in the wires wr_A and wr_S respectively.

Relay_B returns the values present in send_B whenever send_nxt is true (This was required to obtain expected behaviour).

The int MAC and float MAC units are exactly the same modules which were developed in phase 1.

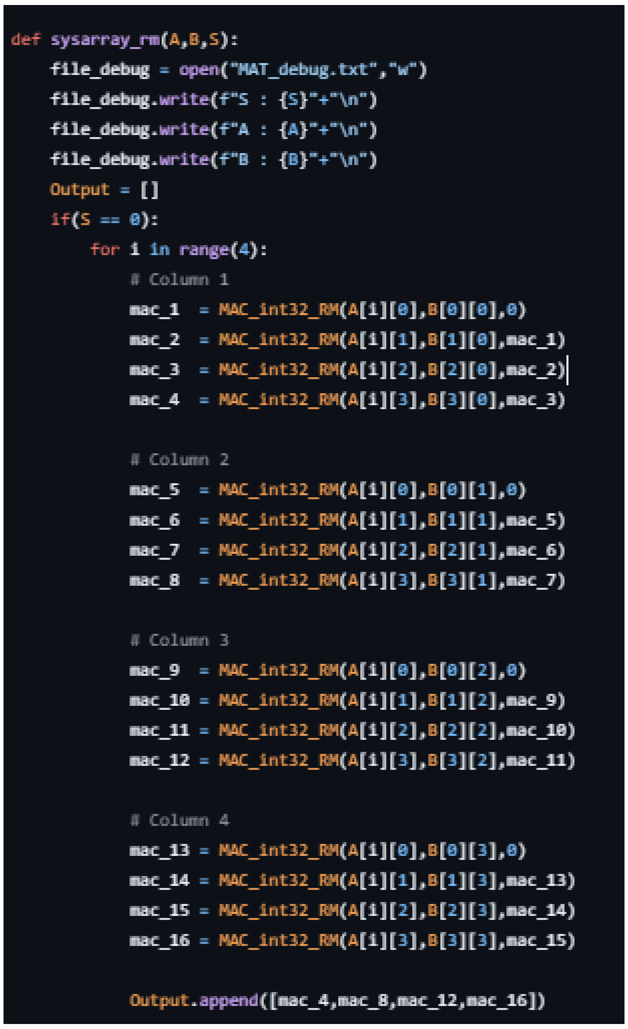
<u>REFERENCE MODEL:</u>

In this section, systolic array reference model is explained.



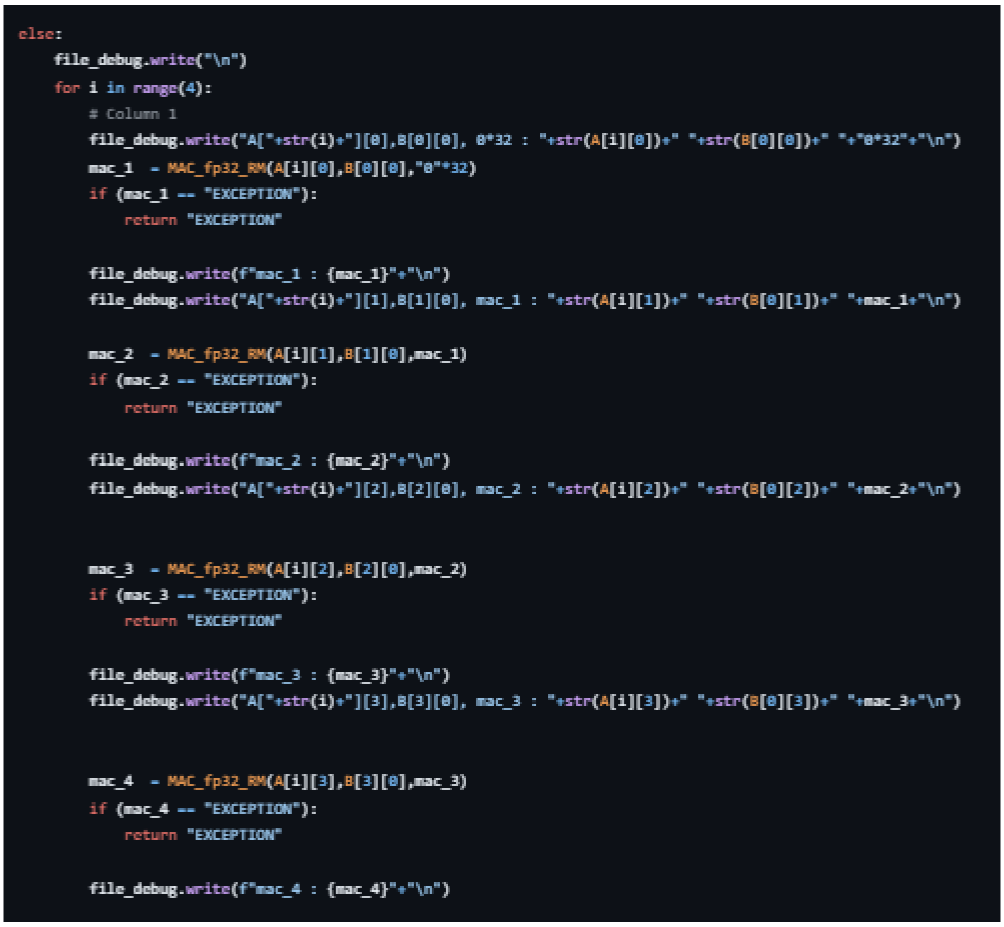
```
from FLOAT_RM import *
from INT_RM import *
```

The above images shows the reference models of int and float MAC being imported. (These were developed in phase 1).



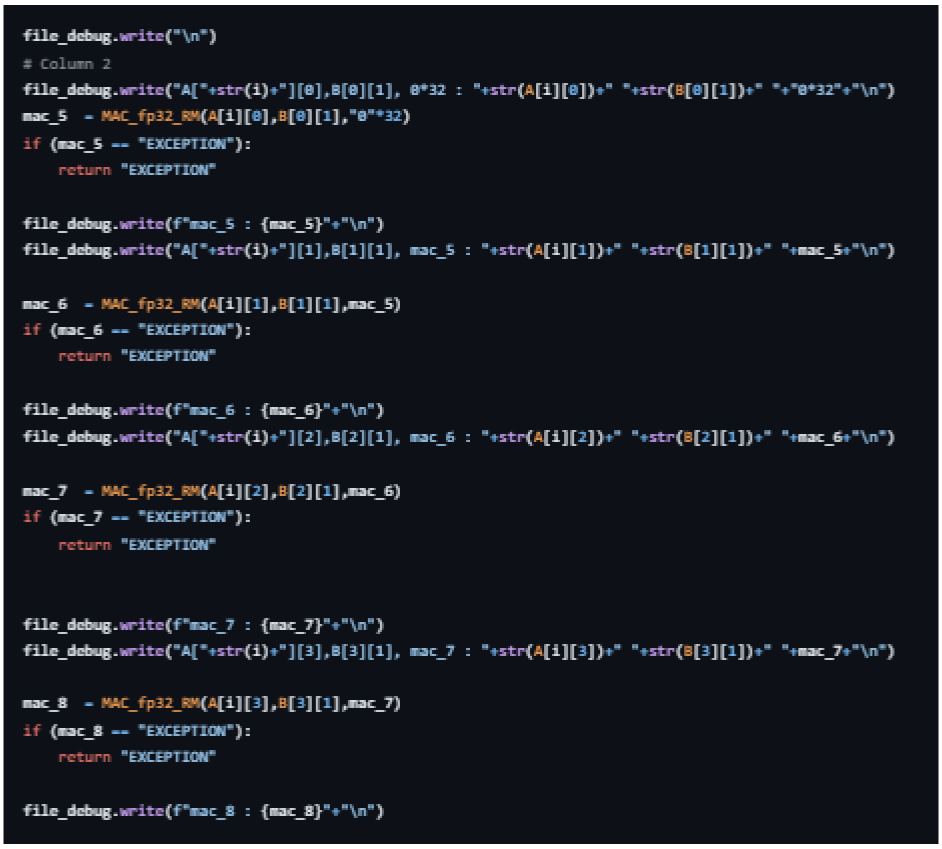
```
def sysarray_rm(A,B,S):
    file_debug = open("MAT_debug.txt","w")
    file_debug.write(f"S : {S}"+"\n")
    file_debug.write(f"A : {A}"+"\n")
    file_debug.write(f"B : {B}"+"\n")
    Output = []
    if(S == 0):
        for i in range(4):
            # Column 1
            mac_1  = MAC_int32_RM(A[i][0],B[0][0],0)
            mac_2  = MAC_int32_RM(A[i][1],B[1][0],mac_1)
            mac_3  = MAC_int32_RM(A[i][2],B[2][0],mac_2)
            mac_4  = MAC_int32_RM(A[i][3],B[3][0],mac_3)

            # Column 2
            mac_5  = MAC_int32_RM(A[i][0],B[0][1],0)
            mac_6  = MAC_int32_RM(A[i][1],B[1][1],mac_5)
            mac_7  = MAC_int32_RM(A[i][2],B[2][1],mac_6)
            mac_8  = MAC_int32_RM(A[i][3],B[3][1],mac_7)

            # Column 3
            mac_9  = MAC_int32_RM(A[i][0],B[0][2],0)
            mac_10 = MAC_int32_RM(A[i][1],B[1][2],mac_9)
            mac_11 = MAC_int32_RM(A[i][2],B[2][2],mac_10)
            mac_12 = MAC_int32_RM(A[i][3],B[3][2],mac_11)

            # Column 4
            mac_13 = MAC_int32_RM(A[i][0],B[0][3],0)
            mac_14 = MAC_int32_RM(A[i][1],B[1][3],mac_13)
            mac_15 = MAC_int32_RM(A[i][2],B[2][3],mac_14)
            mac_16 = MAC_int32_RM(A[i][3],B[3][3],mac_15)

            Output.append([mac_4,mac_8,mac_12,mac_16])
```

The above image shows the systolic array reference model for int. The first four lines of code opens text file to write debug statements. The code inside for loop is main code which actually performs int matrix multiplication. It is a series of function calls to MAC_int32_RM with appropriate inputs to mimic the mesh network we made with bsv. At each iteration of for loop, a single row of answer is obtained, which are continuously appended to output matrix.



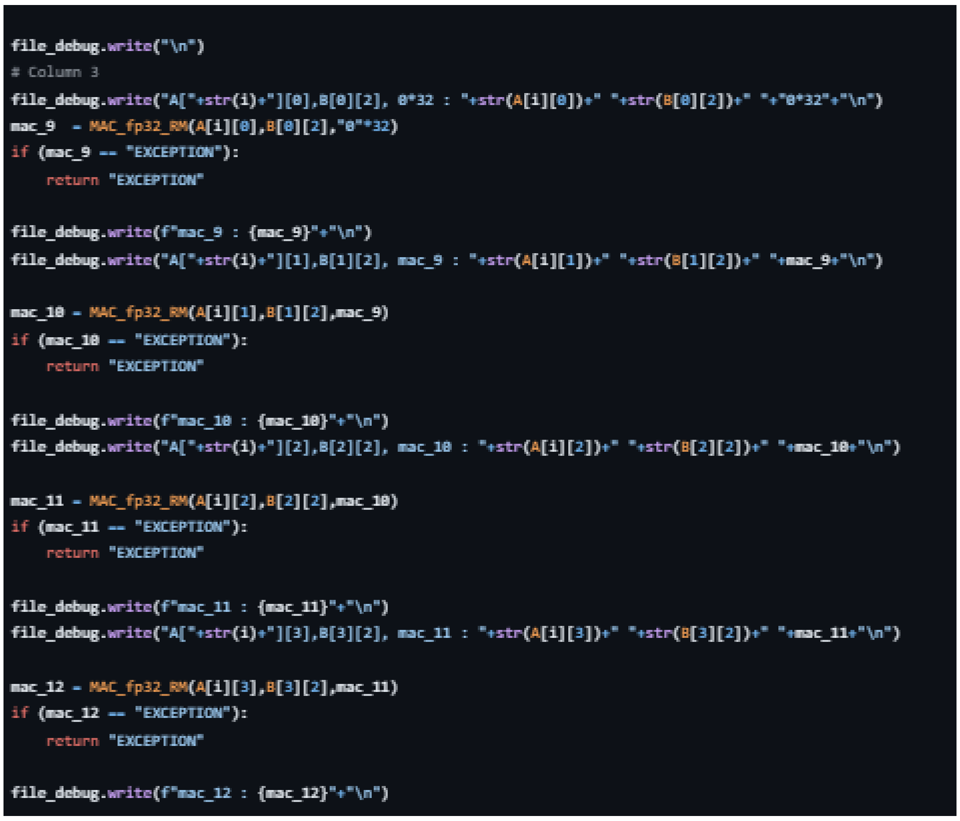
```
else:
    file_debug.write("\n")
    for i in range(4):
        # Column 1
        file_debug.write("A["+str(i)+"][0],B[0][0], 0*32 : "+str(A[i][0])+" "+str(B[0][0])+" "+"0*32"+"\n")
        mac_1  = MAC_fp32_RM(A[i][0],B[0][0],"0"*32)
        if (mac_1 == "EXCEPTION"):
            return "EXCEPTION"

        file_debug.write(f"mac_1 : {mac_1}"+"\n")
        file_debug.write("A["+str(i)+"][1],B[1][0], mac_1 : "+str(A[i][1])+" "+str(B[0][1])+" "+mac_1+"\n")

        mac_2  = MAC_fp32_RM(A[i][1],B[1][0],mac_1)
        if (mac_2 == "EXCEPTION"):
            return "EXCEPTION"

        file_debug.write(f"mac_2 : {mac_2}"+"\n")
        file_debug.write("A["+str(i)+"][2],B[2][0], mac_2 : "+str(A[i][2])+" "+str(B[0][2])+" "+mac_2+"\n")


        mac_3  = MAC_fp32_RM(A[i][2],B[2][0],mac_2)
        if (mac_3 == "EXCEPTION"):
            return "EXCEPTION"

        file_debug.write(f"mac_3 : {mac_3}"+"\n")
        file_debug.write("A["+str(i)+"][3],B[3][0], mac_3 : "+str(A[i][3])+" "+str(B[0][3])+" "+mac_3+"\n")


        mac_4  = MAC_fp32_RM(A[i][3],B[3][0],mac_3)
        if (mac_4 == "EXCEPTION"):
            return "EXCEPTION"

        file_debug.write(f"mac_4 : {mac_4}"+"\n")
```

The above image shows the code which does matrix multiplication of floating numbers. It is a series of function calls to MAC_fp32_RM with appropriate inputs to mimic the mesh network created in bsv. The above image focuses on first column of operation in mesh network. At any point in calculation, if overflow is detected, "EXCEPTION" is returned. Also, debug statements are written into text files for later debugging.



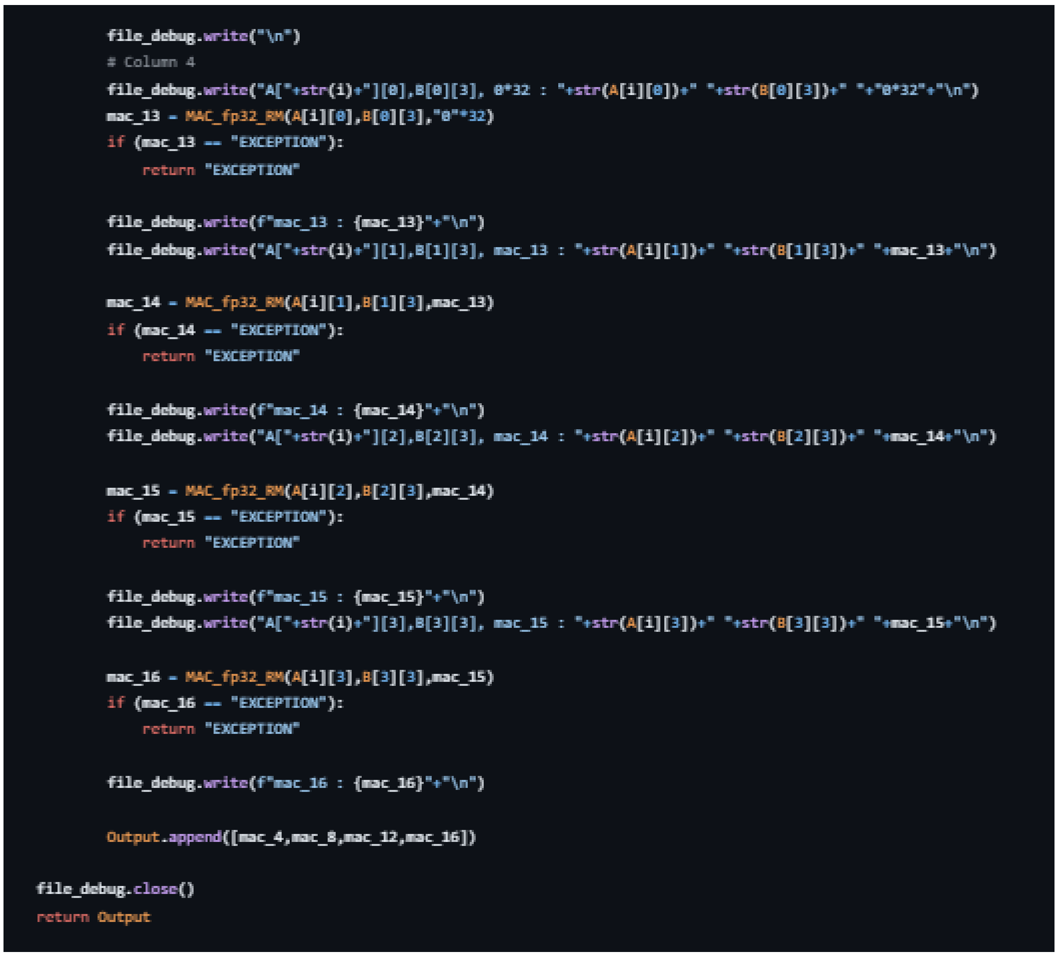
```
file_debug.write("\n")
# Column 2
file_debug.write("A["+str(i)+"][0],B[0][1], 0*32 : "+str(A[i][0])+" "+str(B[0][1])+" "+"0*32"+"\n")
mac_5  = MAC_fp32_RM(A[i][0],B[0][1],"0"*32)
if (mac_5 == "EXCEPTION"):
    return "EXCEPTION"

file_debug.write(f"mac_5 : {mac_5}"+"\n")
file_debug.write("A["+str(i)+"][1],B[1][1], mac_5 : "+str(A[i][1])+" "+str(B[1][1])+" "+mac_5+"\n")

mac_6  = MAC_fp32_RM(A[i][1],B[1][1],mac_5)
if (mac_6 == "EXCEPTION"):
    return "EXCEPTION"

file_debug.write(f"mac_6 : {mac_6}"+"\n")
file_debug.write("A["+str(i)+"][2],B[2][1], mac_6 : "+str(A[i][2])+" "+str(B[2][1])+" "+mac_6+"\n")

mac_7  = MAC_fp32_RM(A[i][2],B[2][1],mac_6)
if (mac_7 == "EXCEPTION"):
    return "EXCEPTION"


file_debug.write(f"mac_7 : {mac_7}"+"\n")
file_debug.write("A["+str(i)+"][3],B[3][1], mac_7 : "+str(A[i][3])+" "+str(B[3][1])+" "+mac_7+"\n")

mac_8  = MAC_fp32_RM(A[i][3],B[3][1],mac_7)
if (mac_8 == "EXCEPTION"):
    return "EXCEPTION"

file_debug.write(f"mac_8 : {mac_8}"+"\n")
```

The above image focuses on the second column of operations in the mesh.



```
file_debug.write("\n")
# Column 3
file_debug.write("A["+str(i)+"][0],B[0][2], 0*32 : "+str(A[i][0])+" "+str(B[0][2])+" "+"0*32"+"\n")
mac_9  = MAC_fp32_RM(A[i][0],B[0][2],"0"*32)
if (mac_9 == "EXCEPTION"):
    return "EXCEPTION"

file_debug.write(f"mac_9 : {mac_9}"+"\n")
file_debug.write("A["+str(i)+"][1],B[1][2], mac_9 : "+str(A[i][1])+" "+str(B[1][2])+" "+mac_9+"\n")

mac_10 = MAC_fp32_RM(A[i][1],B[1][2],mac_9)
if (mac_10 == "EXCEPTION"):
    return "EXCEPTION"

file_debug.write(f"mac_10 : {mac_10}"+"\n")
file_debug.write("A["+str(i)+"][2],B[2][2], mac_10 : "+str(A[i][2])+" "+str(B[2][2])+" "+mac_10+"\n")

mac_11 = MAC_fp32_RM(A[i][2],B[2][2],mac_10)
if (mac_11 == "EXCEPTION"):
    return "EXCEPTION"

file_debug.write(f"mac_11 : {mac_11}"+"\n")
file_debug.write("A["+str(i)+"][3],B[3][2], mac_11 : "+str(A[i][3])+" "+str(B[3][2])+" "+mac_11+"\n")

mac_12 = MAC_fp32_RM(A[i][3],B[3][2],mac_11)
if (mac_12 == "EXCEPTION"):
    return "EXCEPTION"

file_debug.write(f"mac_12 : {mac_12}"+"\n")
```

The above image focuses on the third column of operations in the mesh network.



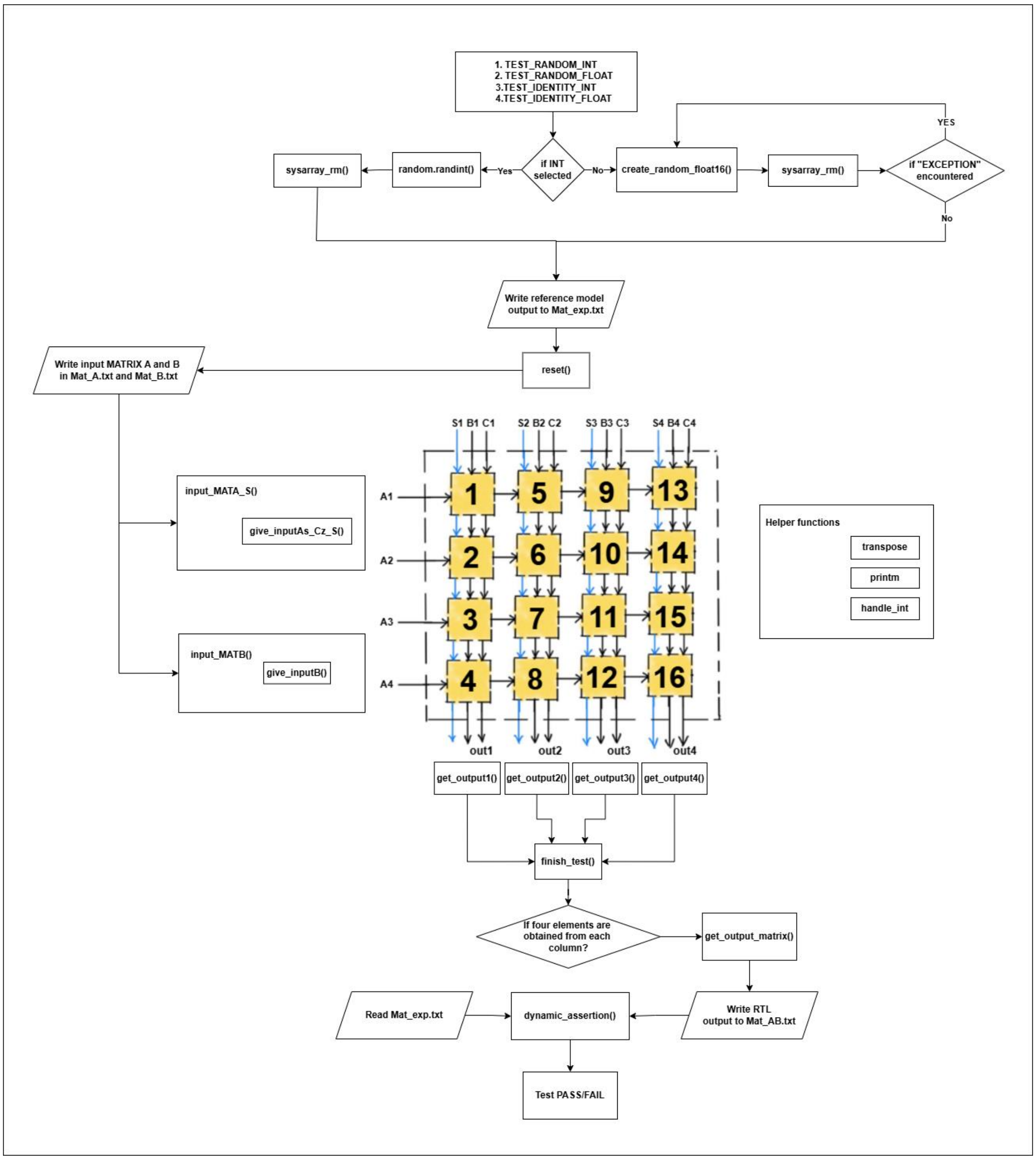
```python
        file_debug.write("\n")
        # Column 4
        file_debug.write("A["+str(i)+"][0],B[0][3], 0*32 : "+str(A[i][0])+" "+str(B[0][3])+" "+"0*32"+"\n")
        mac_13 = MAC_fp32_RM(A[i][0],B[0][3],"0"*32)
        if (mac_13 == "EXCEPTION"):
            return "EXCEPTION"

        file_debug.write(f"mac_13 : {mac_13}"+"\n")
        file_debug.write("A["+str(i)+"][1],B[1][3], mac_13 : "+str(A[i][1])+" "+str(B[1][3])+" "+mac_13+"\n")

        mac_14 = MAC_fp32_RM(A[i][1],B[1][3],mac_13)
        if (mac_14 == "EXCEPTION"):
            return "EXCEPTION"

        file_debug.write(f"mac_14 : {mac_14}"+"\n")
        file_debug.write("A["+str(i)+"][2],B[2][3], mac_14 : "+str(A[i][2])+" "+str(B[2][3])+" "+mac_14+"\n")

        mac_15 = MAC_fp32_RM(A[i][2],B[2][3],mac_14)
        if (mac_15 == "EXCEPTION"):
            return "EXCEPTION"

        file_debug.write(f"mac_15 : {mac_15}"+"\n")
        file_debug.write("A["+str(i)+"][3],B[3][3], mac_15 : "+str(A[i][3])+" "+str(B[3][3])+" "+mac_15+"\n")

        mac_16 = MAC_fp32_RM(A[i][3],B[3][3],mac_15)
        if (mac_16 == "EXCEPTION"):
            return "EXCEPTION"

        file_debug.write(f"mac_16 : {mac_16}"+"\n")

        Output.append([mac_4,mac_8,mac_12,mac_16])

    file_debug.close()
    return Output
```

The above image focuses on the fourth column of operations in the mesh network. At each iteration of for loop, a single row of answer is obtained, which are continuously appended to output matrix.

Finally, the output matrix is returned.

TESTBENCH:

This section explains the working of testbench. The following image shows the overall working of the testbench.

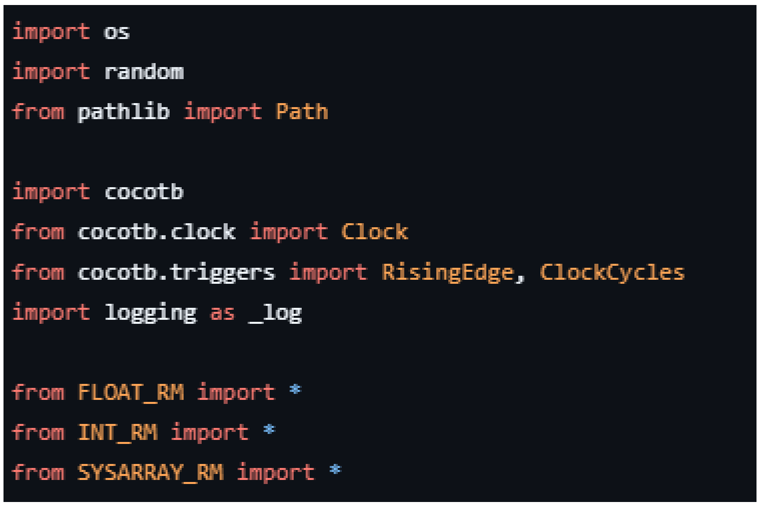


The testbench has four modes of operation:

- TEST_RANDOM_INT
- TEST_RANDOM_FLOAT
- TEST_IDENTITY_INT
- TEST_IDENTITY_FLOAT

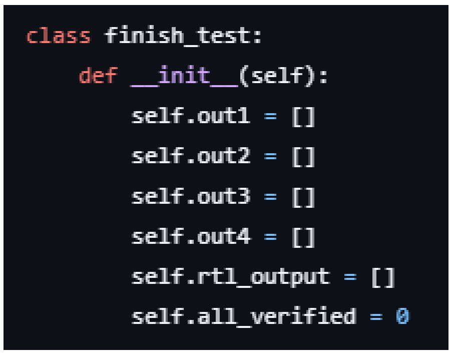
The user can choose the mode of operation and number of testcases via cocotb plusargs given through terminal. The randomly generated A and B input matrices are written to Mat_A.txt and Mat_B.txt respectively. The reference model output is written to Mat_exp.txt and the RTL output is written to Mat_AB.txt.

Now let's see what exactly the functions in testbench do:



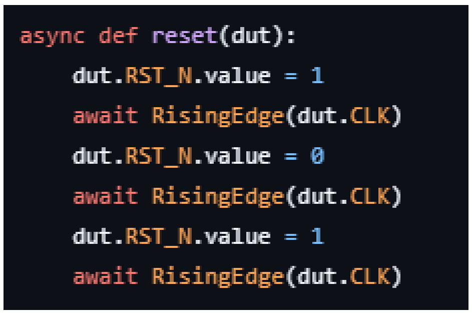
```python
import os
import random
from pathlib import Path

import cocotb
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge, ClockCycles
import logging as _log

from FLOAT_RM import *
from INT_RM import *
from SYSARRAY_RM import *
```

The above image shows the imports done in testbench.



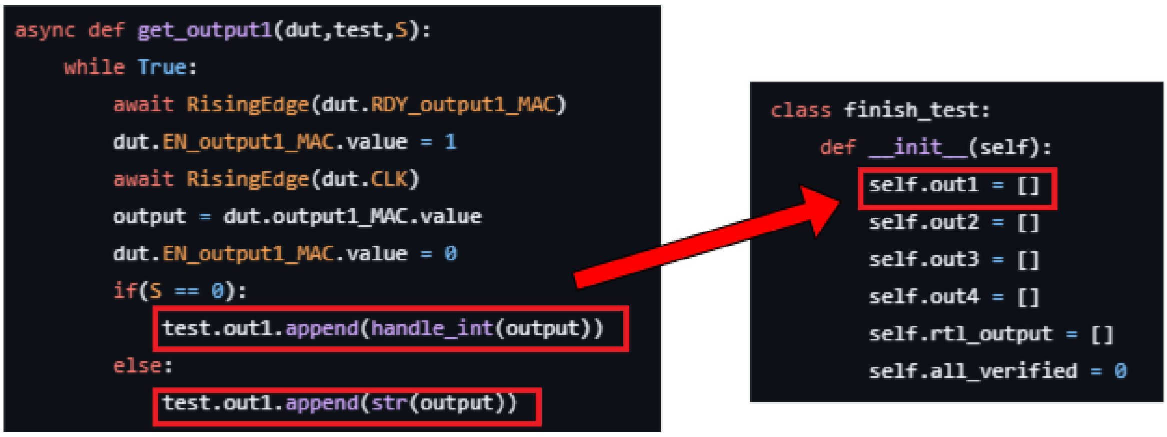
```python
class finish_test:
    def __init__(self):
        self.out1 = []
        self.out2 = []
        self.out3 = []
        self.out4 = []
        self.rtl_output = []
        self.all_verified = 0
```

The above image shows a class declaration. The object created from the class finish_test helps in gathering the output of RTL. self.out1 is a list, intended to store the answers of first column of systolic array. self.out2 is a list, intended to store the answers of second column of systolic array. self.out3 is a list, intended to store the answers of third column of systolic array. self.out4 is a list, intended to store the answers of fourth column of systolic array. Once self.out1, self.out2, self.out3 and self.out4 have four elements each, it means a complete 4x4 output matrix is obtained. When that occurs the matrix as a whole is appended to self.rtl_output list and self.out1, self.out2, self.out3 and self.out4 are cleared. Self.all_verified is used as a flag variable to know when all answers are checked with reference model output.



```
async def reset(dut):
    dut.RST_N.value = 1
    await RisingEdge(dut.CLK)
    dut.RST_N.value = 0
    await RisingEdge(dut.CLK)
    dut.RST_N.value = 1
    await RisingEdge(dut.CLK)
```

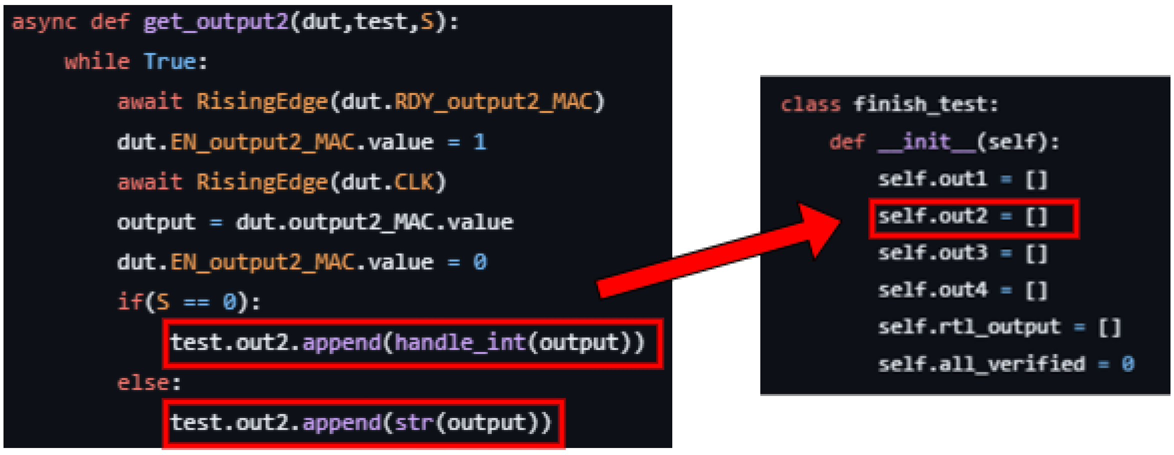
The above image shows the reset function. When this function is called systolic array is reset.



```
async def get_output1(dut,test,S):
    while True:
        await RisingEdge(dut.RDY_output1_MAC)
        dut.EN_output1_MAC.value = 1
        await RisingEdge(dut.CLK)
        output = dut.output1_MAC.value
        dut.EN_output1_MAC.value = 0
        if(S == 0):
            test.out1.append(handle_int(output))
        else:
            test.out1.append(str(output))
```

```
class finish_test:
    def __init__(self):
        self.out1 = []
        self.out2 = []
        self.out3 = []
        self.out4 = []
        self.rtl_output = []
        self.all_verified = 0
```

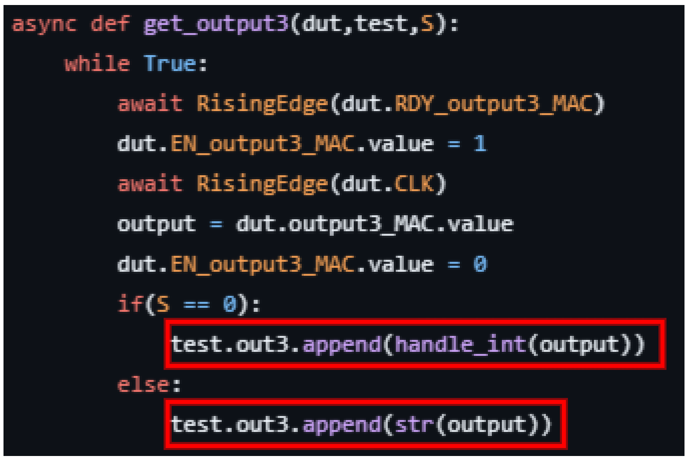
The above image shows the logic present inside get_output1 function. This function runs indefinitely until all tests are run and answers are checked. This function samples the output of first column whenever the output is ready and appends the value to self.out1 list as shown. If S is zero, then handle_int is called to handle negative integers before appending.



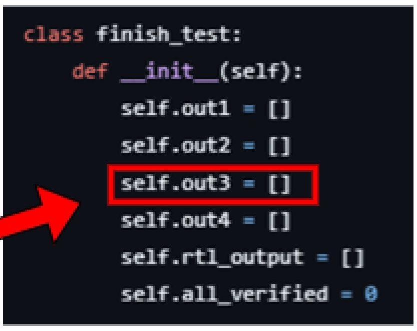
```
async def get_output2(dut,test,S):
    while True:
        await RisingEdge(dut.RDY_output2_MAC)
        dut.EN_output2_MAC.value = 1
        await RisingEdge(dut.CLK)
        output = dut.output2_MAC.value
        dut.EN_output2_MAC.value = 0
        if(S == 0):
            test.out2.append(handle_int(output))
        else:
            test.out2.append(str(output))
```

```
class finish_test:
    def __init__(self):
        self.out1 = []
        self.out2 = []
        self.out3 = []
        self.out4 = []
        self.rtl_output = []
        self.all_verified = 0
```

The above image shows the logic present inside get_output2 function. This function runs indefinitely until all tests are run and answers are checked. This function samples the output of second column whenever the output is ready and appends the value to self.out2 list as shown. If S is zero, then handle_int is called to handle negative integers before appending.

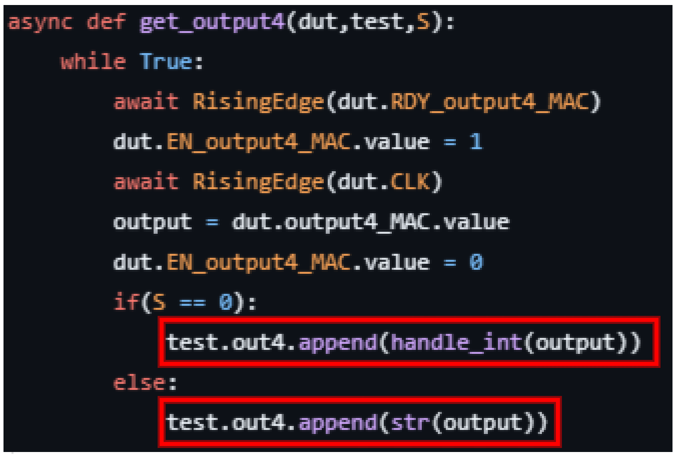


```
async def get_output3(dut,test,S):
    while True:
        await RisingEdge(dut.RDY_output3_MAC)
        dut.EN_output3_MAC.value = 1
        await RisingEdge(dut.CLK)
        output = dut.output3_MAC.value
        dut.EN_output3_MAC.value = 0
        if(S == 0):
            test.out3.append(handle_int(output))
        else:
            test.out3.append(str(output))
```

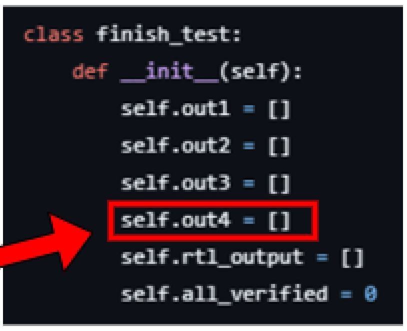


```
class finish_test:
    def __init__(self):
        self.out1 = []
        self.out2 = []
        self.out3 = []
        self.out4 = []
        self.rtl_output = []
        self.all_verified = 0
```

The above image shows the logic present inside get_output3 function. This function runs indefinitely until all tests are run and answers are checked. This function samples the output of third column whenever the output is ready and appends the value to self.out3 list as shown. If S is zero, then handle_int is called to handle negative integers before appending.

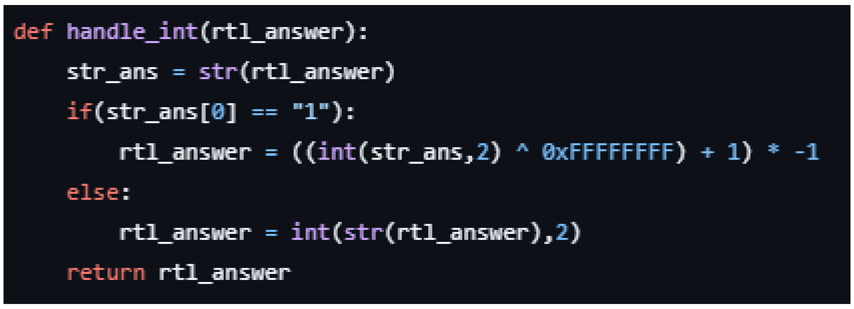


```
async def get_output4(dut,test,S):
    while True:
        await RisingEdge(dut.RDY_output4_MAC)
        dut.EN_output4_MAC.value = 1
        await RisingEdge(dut.CLK)
        output = dut.output4_MAC.value
        dut.EN_output4_MAC.value = 0
        if(S == 0):
            test.out4.append(handle_int(output))
        else:
            test.out4.append(str(output))
```



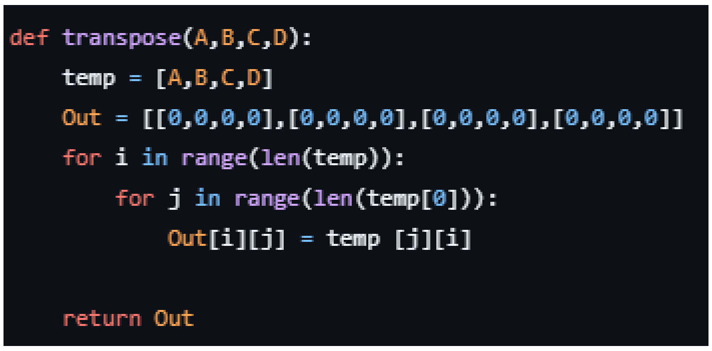
```
class finish_test:
    def __init__(self):
        self.out1 = []
        self.out2 = []
        self.out3 = []
        self.out4 = []
        self.rtl_output = []
        self.all_verified = 0
```

The above image shows the logic present inside get_output4 function. This function runs indefinitely until all tests are run and answers are checked. This function samples the output of fourth column whenever the output is ready and appends the value to self.out4 list as shown. If S is zero, then handle_int is called to handle negative integers before appending.



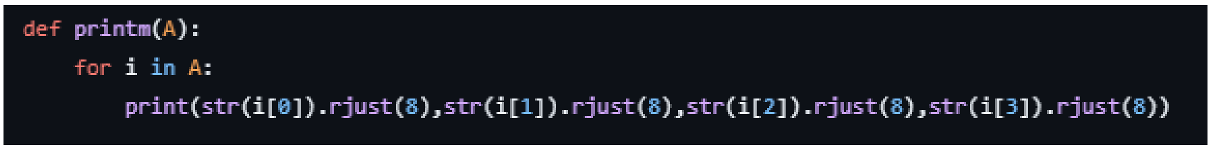
```python
def handle_int(rtl_answer):
    str_ans = str(rtl_answer)
    if(str_ans[0] == "1"):
        rtl_answer = ((int(str_ans,2) ^ 0xFFFFFFFF) + 1) * -1
    else:
        rtl_answer = int(str(rtl_answer),2)
    return rtl_answer
```

The above image shows the logic of handle_int function. If sign bit is one, it takes the two's compliment and puts negative sign in the front.



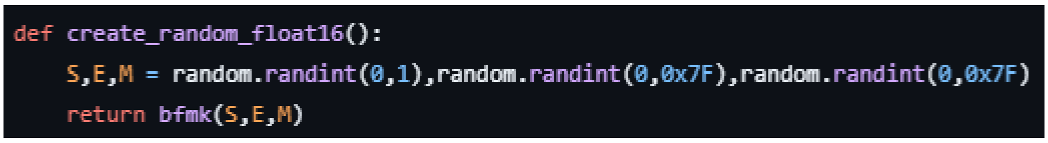
```python
def transpose(A,B,C,D):
    temp = [A,B,C,D]
    Out = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
    for i in range(len(temp)):
        for j in range(len(temp[0])):
            Out[i][j] = temp [j][i]

    return Out
```

The above image shows the transpose function. This function is used to transpose the output generated by the RTL to get correct answer.



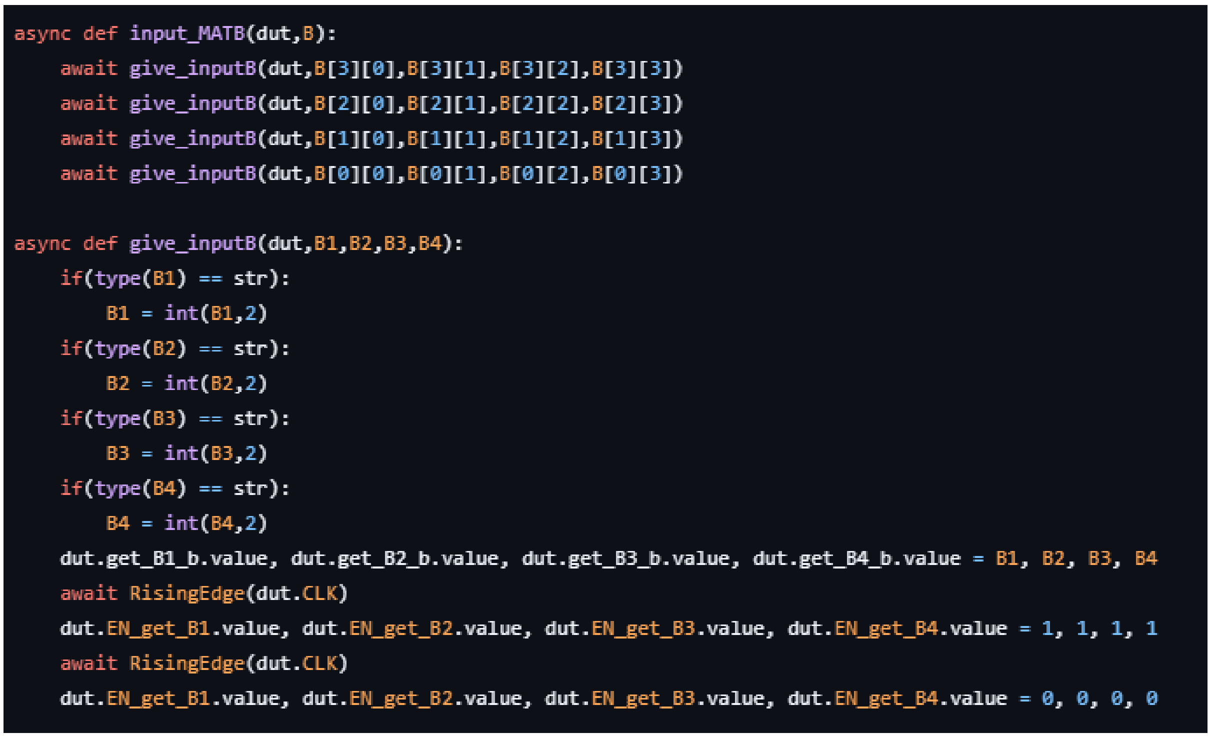
```python
def printm(A):
    for i in A:
        print(str(i[0]).rjust(8),str(i[1]).rjust(8),str(i[2]).rjust(8),str(i[3]).rjust(8))
```

The code in the above image just prints the elements of a matrix in proper format for easier visualisation.



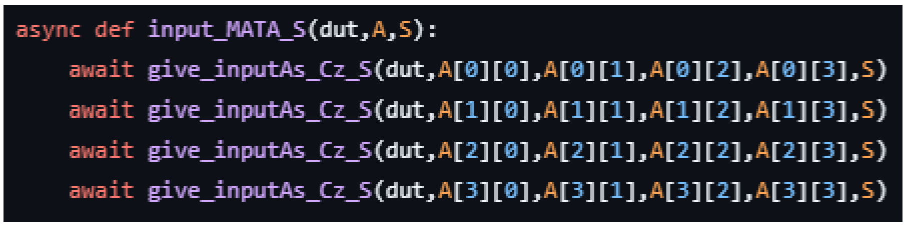
```python
def create_random_float16():
    S,E,M = random.randint(0,1),random.randint(0,0x7F),random.randint(0,0x7F)
    return bfmk(S,E,M)
```

The above function is used to generate random Bfloat16 numbers.



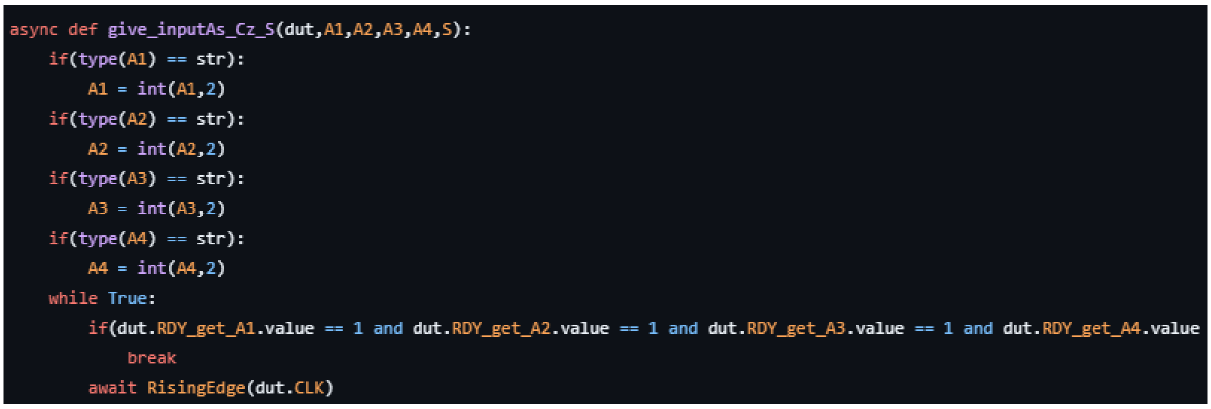
```python
async def input_MATB(dut,B):
    await give_inputB(dut,B[3][0],B[3][1],B[3][2],B[3][3])
    await give_inputB(dut,B[2][0],B[2][1],B[2][2],B[2][3])
    await give_inputB(dut,B[1][0],B[1][1],B[1][2],B[1][3])
    await give_inputB(dut,B[0][0],B[0][1],B[0][2],B[0][3])

async def give_inputB(dut,B1,B2,B3,B4):
    if(type(B1) == str):
        B1 = int(B1,2)
    if(type(B2) == str):
        B2 = int(B2,2)
    if(type(B3) == str):
        B3 = int(B3,2)
    if(type(B4) == str):
        B4 = int(B4,2)
    dut.get_B1_b.value, dut.get_B2_b.value, dut.get_B3_b.value, dut.get_B4_b.value = B1, B2, B3, B4
    await RisingEdge(dut.CLK)
    dut.EN_get_B1.value, dut.EN_get_B2.value, dut.EN_get_B3.value, dut.EN_get_B4.value = 1, 1, 1, 1
    await RisingEdge(dut.CLK)
    dut.EN_get_B1.value, dut.EN_get_B2.value, dut.EN_get_B3.value, dut.EN_get_B4.value = 0, 0, 0, 0
```

The input_MATB function shifts in B matrix through the systolic array for computation. Input_MATB function in turn calls give_inputB to input matrix B.

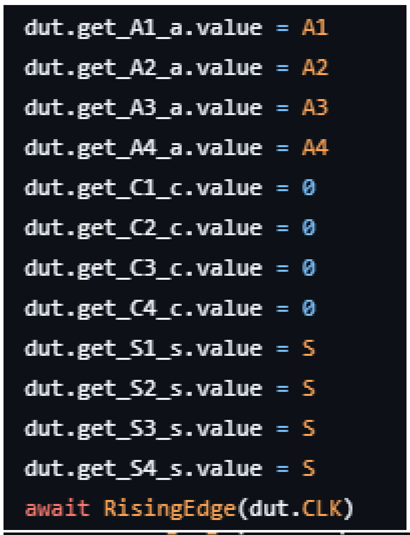


```python
async def input_MATA_S(dut,A,S):
    await give_inputAs_Cz_S(dut,A[0][0],A[0][1],A[0][2],A[0][3],S)
    await give_inputAs_Cz_S(dut,A[1][0],A[1][1],A[1][2],A[1][3],S)
    await give_inputAs_Cz_S(dut,A[2][0],A[2][1],A[2][2],A[2][3],S)
    await give_inputAs_Cz_S(dut,A[3][0],A[3][1],A[3][2],A[3][3],S)
```

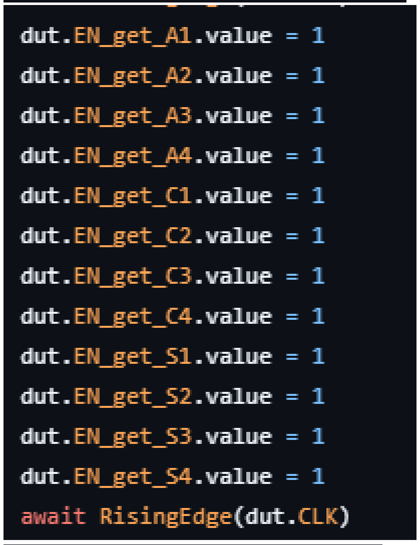
The input_MATA_S function shifts in MATRIX A and the value S into the systolic array.



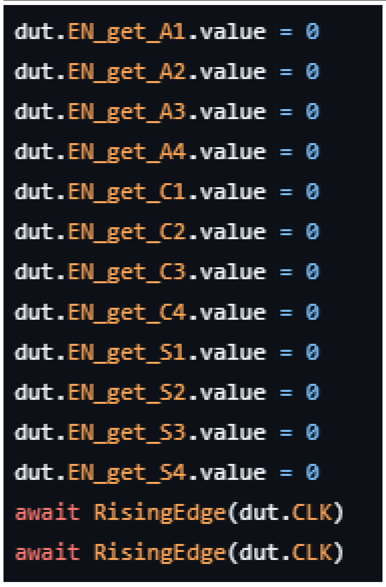
```python
async def give_inputAs_Cz_S(dut,A1,A2,A3,A4,S):
    if(type(A1) == str):
        A1 = int(A1,2)
    if(type(A2) == str):
        A2 = int(A2,2)
    if(type(A3) == str):
        A3 = int(A3,2)
    if(type(A4) == str):
        A4 = int(A4,2)
    while True:
        if(dut.RDY_get_A1.value == 1 and dut.RDY_get_A2.value == 1 and dut.RDY_get_A3.value == 1 and dut.RDY_get_A4.value
            break
        await RisingEdge(dut.CLK)
```



```
dut.get_A1_a.value = A1
dut.get_A2_a.value = A2
dut.get_A3_a.value = A3
dut.get_A4_a.value = A4
dut.get_C1_c.value = 0
dut.get_C2_c.value = 0
dut.get_C3_c.value = 0
dut.get_C4_c.value = 0
dut.get_S1_s.value = S
dut.get_S2_s.value = S
dut.get_S3_s.value = S
dut.get_S4_s.value = S
await RisingEdge(dut.CLK)
```

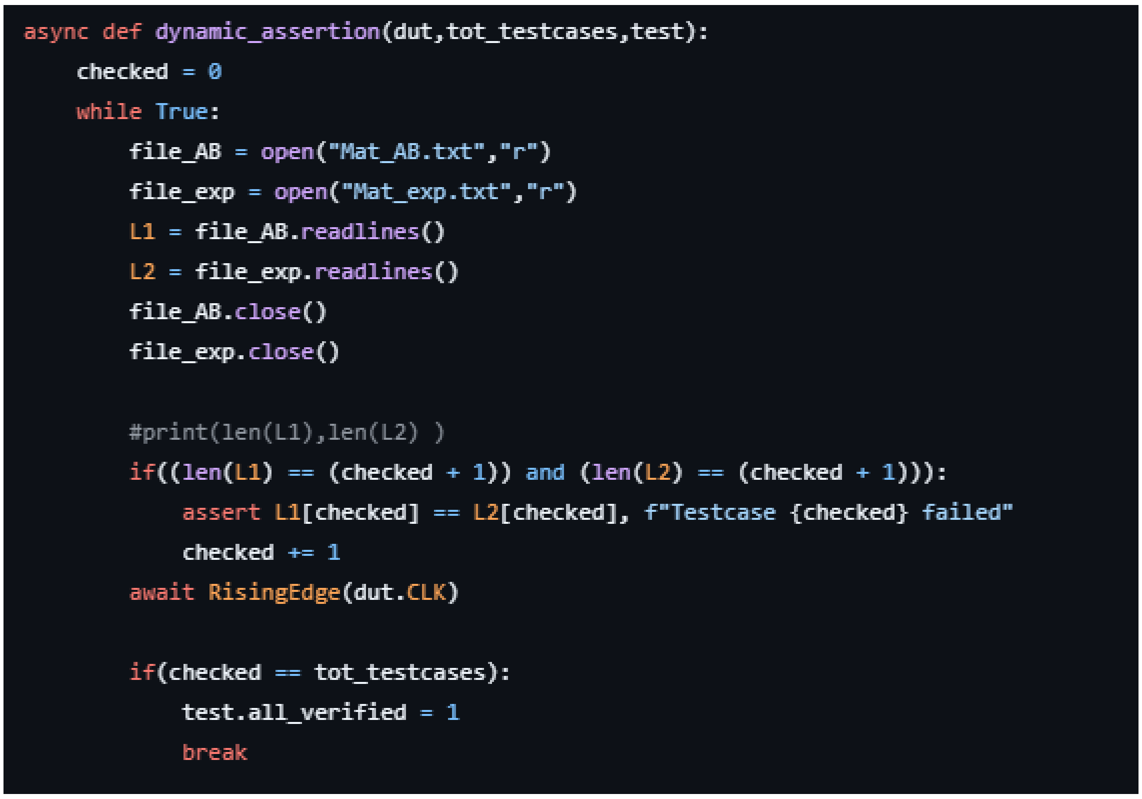


```
dut.EN_get_A1.value = 1
dut.EN_get_A2.value = 1
dut.EN_get_A3.value = 1
dut.EN_get_A4.value = 1
dut.EN_get_C1.value = 1
dut.EN_get_C2.value = 1
dut.EN_get_C3.value = 1
dut.EN_get_C4.value = 1
dut.EN_get_S1.value = 1
dut.EN_get_S2.value = 1
dut.EN_get_S3.value = 1
dut.EN_get_S4.value = 1
await RisingEdge(dut.CLK)
```



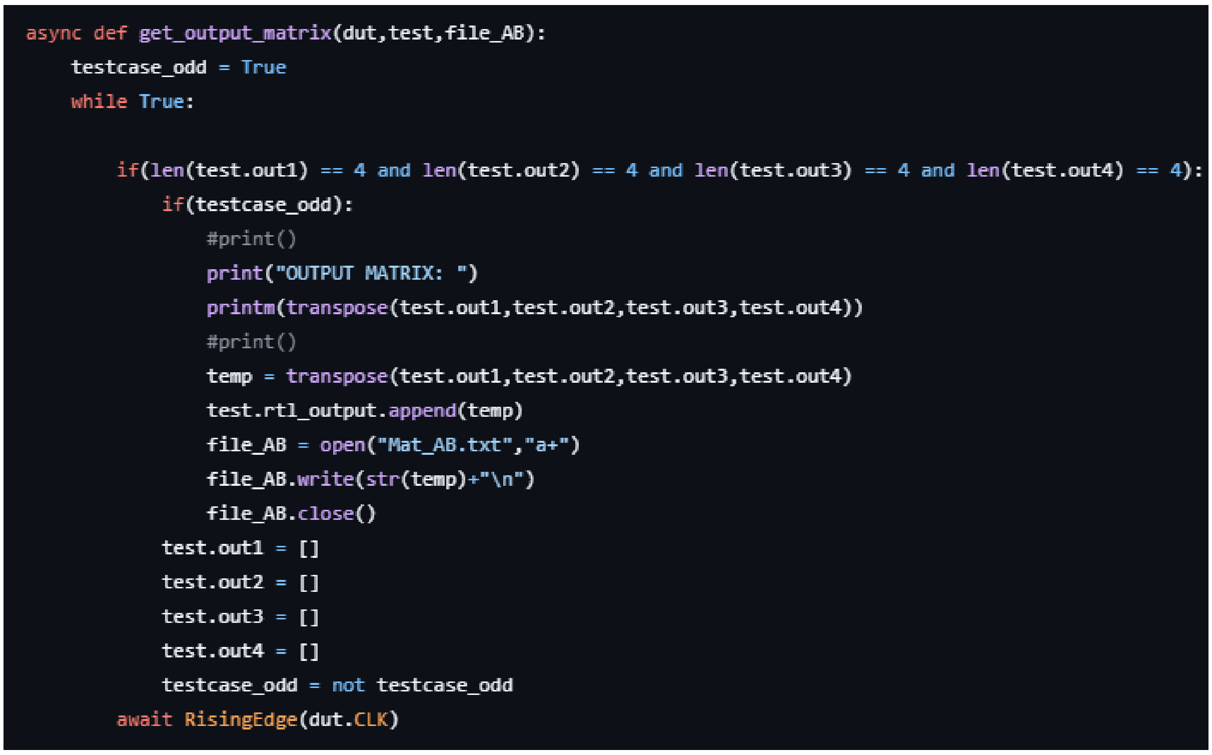
```
dut.EN_get_A1.value = 0
dut.EN_get_A2.value = 0
dut.EN_get_A3.value = 0
dut.EN_get_A4.value = 0
dut.EN_get_C1.value = 0
dut.EN_get_C2.value = 0
dut.EN_get_C3.value = 0
dut.EN_get_C4.value = 0
dut.EN_get_S1.value = 0
dut.EN_get_S2.value = 0
dut.EN_get_S3.value = 0
dut.EN_get_S4.value = 0
await RisingEdge(dut.CLK)
await RisingEdge(dut.CLK)
```

The above four images show how the values of MATRIX A and S are fed. The function waits until the RTL is ready to receive all inputs before providing the inputs. This ensures proper operation.



```python
async def dynamic_assertion(dut,tot_testcases,test):
    checked = 0
    while True:
        file_AB = open("Mat_AB.txt","r")
        file_exp = open("Mat_exp.txt","r")
        L1 = file_AB.readlines()
        L2 = file_exp.readlines()
        file_AB.close()
        file_exp.close()

        #print(len(L1),len(L2) )
        if((len(L1) == (checked + 1)) and (len(L2) == (checked + 1))):
            assert L1[checked] == L2[checked], f"Testcase {checked} failed"
            checked += 1
        await RisingEdge(dut.CLK)

        if(checked == tot_testcases):
            test.all_verified = 1
            break
```

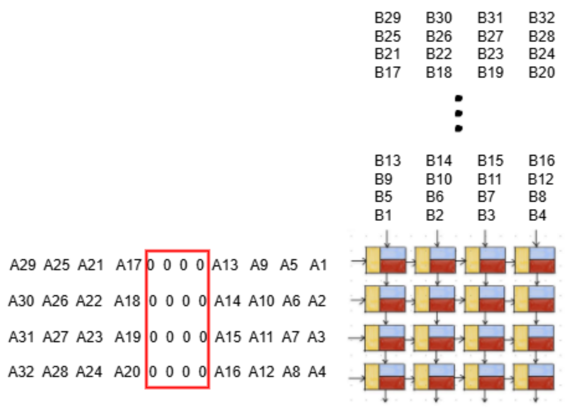
`The above function is really important, it keeps checking the generated outputs with the reference model. If it detects any wrong answer it terminates the simulation and notifies the user that a particular answer mismatched. The variable “checked” is used like a pointer which points to last checked position in the text files. Initially its value is zero. So, it waits till both outputs of RTL and reference model are ready by constantly polling the number of lines in Mat_exp.txt (Reference model output) and Mat_AB.txt (RTL output). When new pair of output is ready, they are checked. If they are same then “checked” variable is incremented. If they are not the same and the test is failed and simulation is terminated. This process keeps happening until desired number of testcases are checked.



```python
async def get_output_matrix(dut,test,file_AB):
    testcase_odd = True
    while True:

        if(len(test.out1) == 4 and len(test.out2) == 4 and len(test.out3) == 4 and len(test.out4) == 4):
            if(testcase_odd):
                #print()
                print("OUTPUT MATRIX: ")
                printm(transpose(test.out1,test.out2,test.out3,test.out4))
                #print()
                temp = transpose(test.out1,test.out2,test.out3,test.out4)
                test.rtl_output.append(temp)
                file_AB = open("Mat_AB.txt","a+")
                file_AB.write(str(temp)+"\n")
                file_AB.close()
            test.out1 = []
            test.out2 = []
            test.out3 = []
            test.out4 = []
            testcase_odd = not testcase_odd
        await RisingEdge(dut.CLK)
```

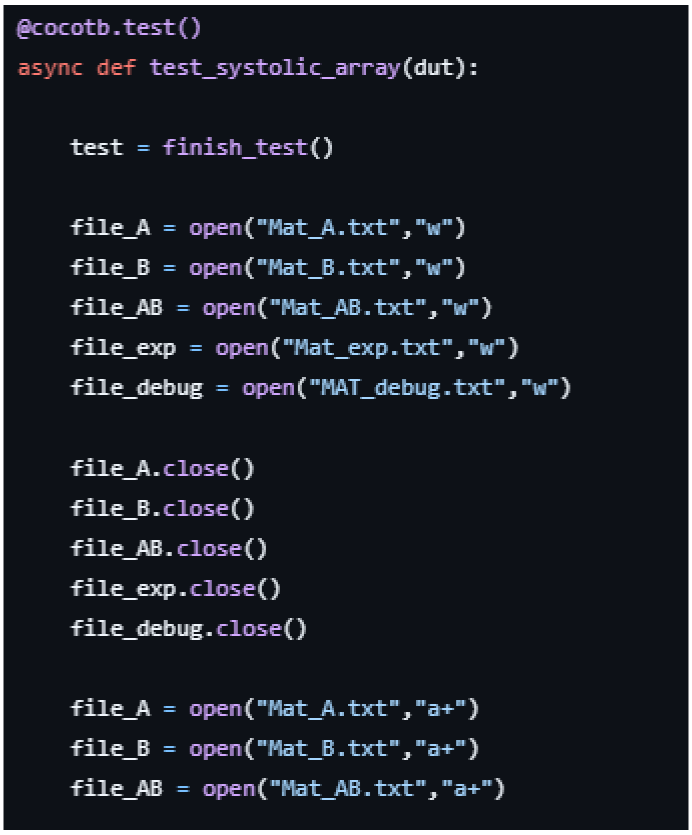
The above image shows the logic used in get_output_matrix function. Whenever four elements are collected by get_output1, get_output2, get_output3 and get_output4 function, the above code detects it and writes the output 4x4 matrix (after taking transpose) to Mat_AB.txt file and prints it in terminal as well. After that the test.out1 to test.out4 lists are cleared to get new output.

The “testcase_odd” variable is used as a flag, which keeps toggling between true and false after each output. This is done because the output of first set of inputs will become available only after the first four members of the set of inputs are fed in. This will inconvenient for testing. So, a decision was made to provide all zeroes to A input after providing first set of inputs until the output is obtained. Then new inputs can be fed. This results in giving testing inputs on odd runs and zeroes on even runs.



B29 B30 B31 B32
B25 B26 B27 B28
B21 B22 B23 B24
B17 B18 B19 B20
B13 B14 B15 B16
B9 B10 B11 B12
B5 B6 B7 B8
B1 B2 B3 B4
A29 A25 A21 A17 0 0 0 0 A13 A9 A5 A1
A30 A26 A22 A18 0 0 0 0 A14 A10 A6 A2
A31 A27 A23 A19 0 0 0 0 A15 A11 A7 A3
A32 A28 A24 A20 0 0 0 0 A16 A12 A8 A4

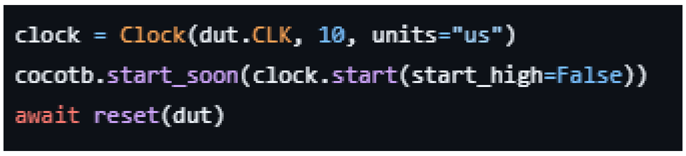
The above image illustrates the way zeroes are fed in as a buffer after each calculation.



```python
@cocotb.test()
async def test_systolic_array(dut):

    test = finish_test()

    file_A = open("Mat_A.txt","w")
    file_B = open("Mat_B.txt","w")
    file_AB = open("Mat_AB.txt","w")
    file_exp = open("Mat_exp.txt","w")
    file_debug = open("MAT_debug.txt","w")

    file_A.close()
    file_B.close()
    file_AB.close()
    file_exp.close()
    file_debug.close()

    file_A = open("Mat_A.txt","a+")
    file_B = open("Mat_B.txt","a+")
    file_AB = open("Mat_AB.txt","a+")
```

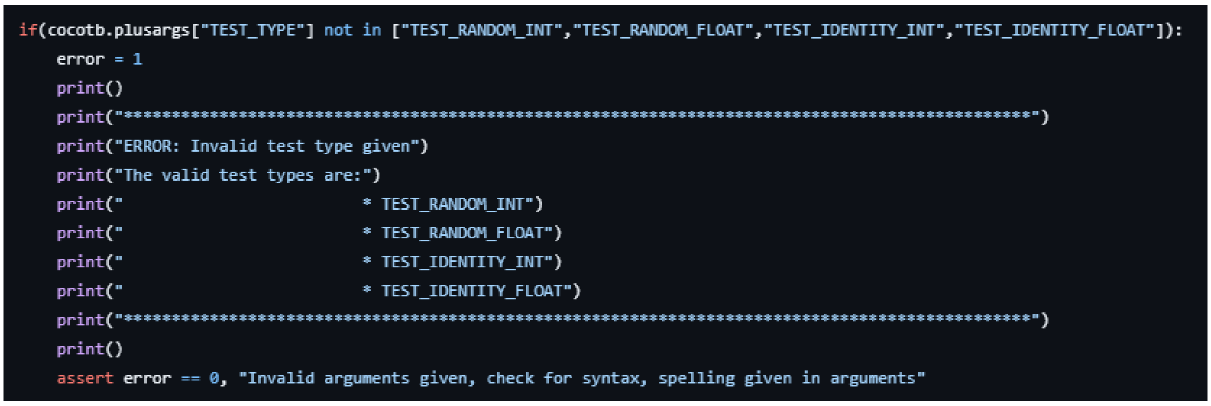
The above image shows the main function in testbench which coordinates all the functions. First an object is created from the class "finish_test" which is explained before.

Then Mat_A.txt, Mat_B.txt, Mat_AB.txt, Mat_exp.txt and MAT_debug.txt are cleared and opened.



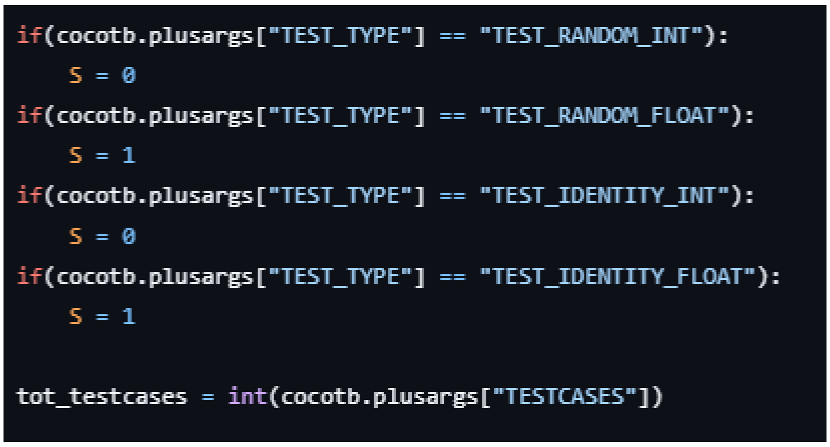
```python
clock = Clock(dut.CLK, 10, units="us")
cocotb.start_soon(clock.start(start_high=False))
await reset(dut)
```

The above code starts the free wheeling clock and resets the systolic array.



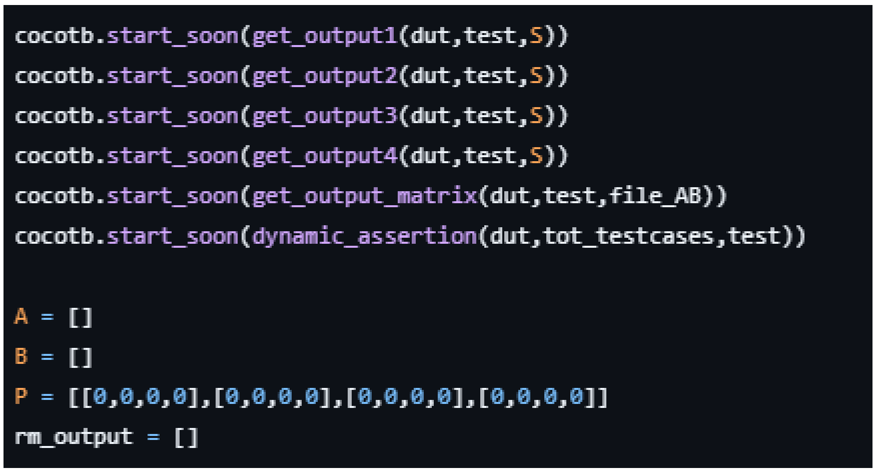
```python
if(cocotb.plusargs["TEST_TYPE"] not in ["TEST_RANDOM_INT","TEST_RANDOM_FLOAT","TEST_IDENTITY_INT","TEST_IDENTITY_FLOAT"]):
    error = 1
    print()
    print("********************************************************************************************")
    print("ERROR: Invalid test type given")
    print("The valid test types are:")
    print("                         * TEST_RANDOM_INT")
    print("                         * TEST_RANDOM_FLOAT")
    print("                         * TEST_IDENTITY_INT")
    print("                         * TEST_IDENTITY_FLOAT")
    print("********************************************************************************************")
    print()
    assert error == 0, "Invalid arguments given, check for syntax, spelling given in arguments"
```

The above lines of code check whether the arguments given by user through plusargs is valid or not. If invalid, suggestions are given to user as to what are the expected valid arguments that the user can pass.



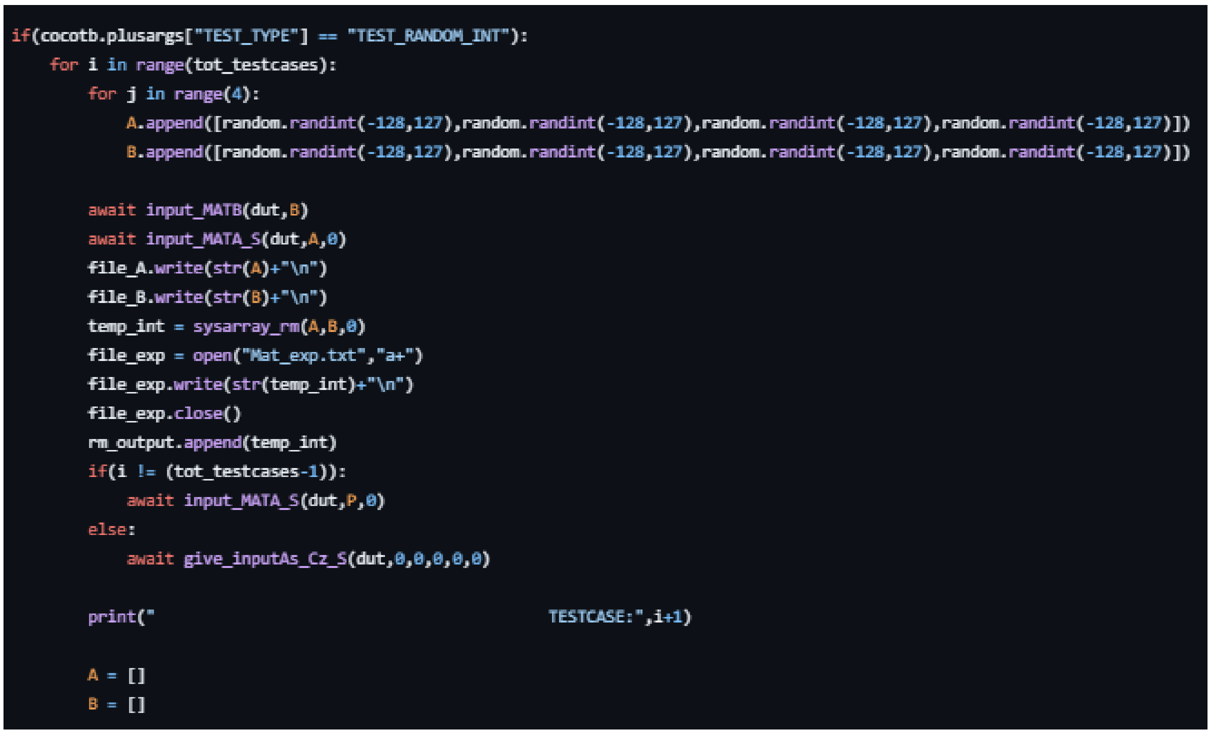
```python
if(cocotb.plusargs["TEST_TYPE"] == "TEST_RANDOM_INT"):
    S = 0
if(cocotb.plusargs["TEST_TYPE"] == "TEST_RANDOM_FLOAT"):
    S = 1
if(cocotb.plusargs["TEST_TYPE"] == "TEST_IDENTITY_INT"):
    S = 0
if(cocotb.plusargs["TEST_TYPE"] == "TEST_IDENTITY_FLOAT"):
    S = 1

tot_testcases = int(cocotb.plusargs["TESTCASES"])
```

The above lines set S according to the type of test chosen by the user. tot_testcases variable contains the total number of testcases the user wishes to run (also given via plusargs).



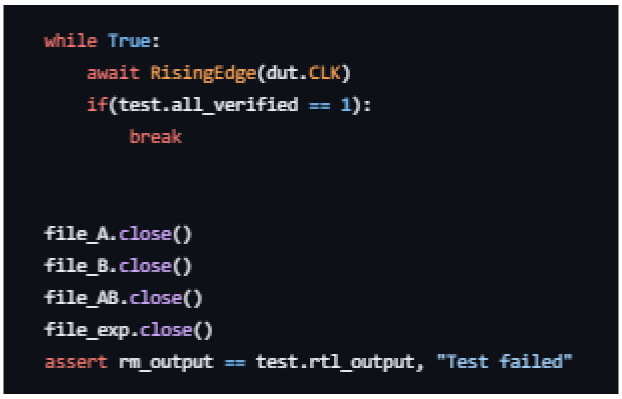
```python
cocotb.start_soon(get_output1(dut,test,S))
cocotb.start_soon(get_output2(dut,test,S))
cocotb.start_soon(get_output3(dut,test,S))
cocotb.start_soon(get_output4(dut,test,S))
cocotb.start_soon(get_output_matrix(dut,test,file_AB))
cocotb.start_soon(dynamic_assertion(dut,tot_testcases,test))

A = []
B = []
P = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
rm_output = []
```

The above lines of code start the parallel processes to observe the output and do dynamic assertions. A and B variables will hold the input matrices. P matrix is the buffer matrix full of zeroes which is fed in between each calculation to flush out the output. rm_output is a variable which is going to hold reference model outputs.



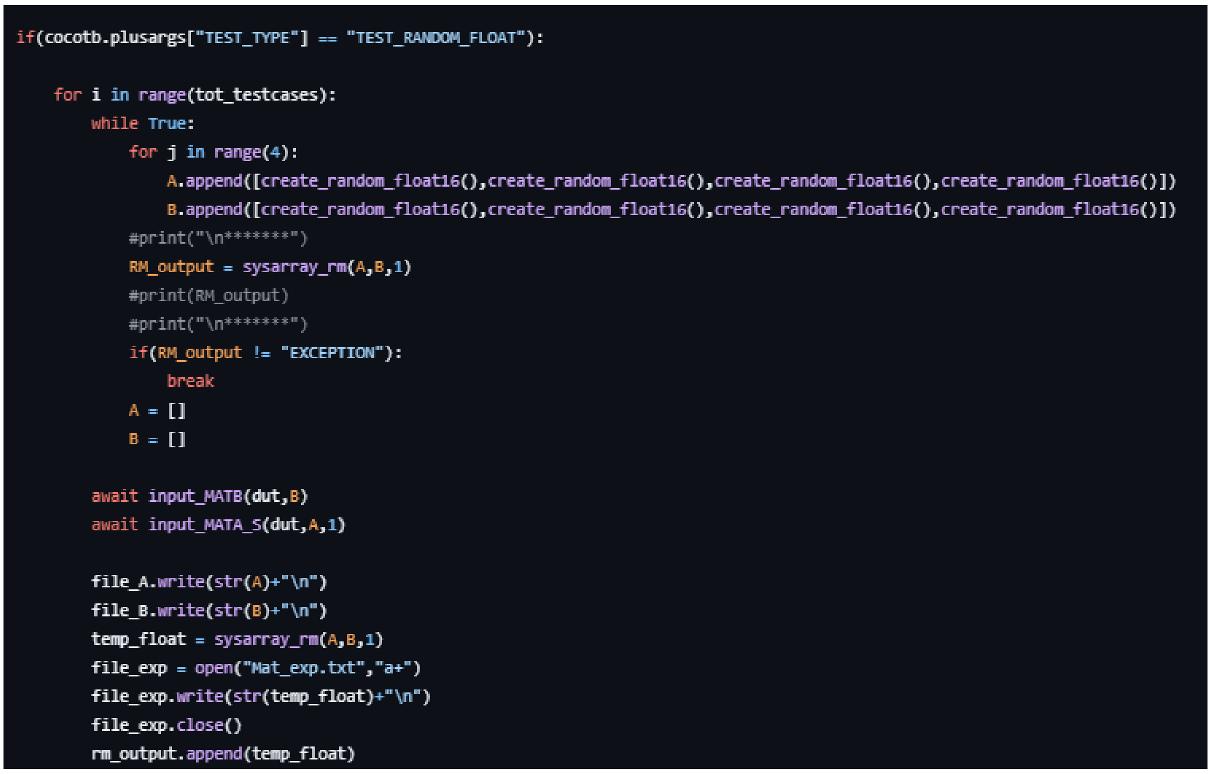
```
if(cocotb.plusargs["TEST_TYPE"] == "TEST_RANDOM_INT"):
    for i in range(tot_testcases):
        for j in range(4):
            A.append([random.randint(-128,127),random.randint(-128,127),random.randint(-128,127),random.randint(-128,127)])
            B.append([random.randint(-128,127),random.randint(-128,127),random.randint(-128,127),random.randint(-128,127)])

        await input_MATB(dut,B)
        await input_MATA_S(dut,A,0)
        file_A.write(str(A)+"\n")
        file_B.write(str(B)+"\n")
        temp_int = sysarray_rm(A,B,0)
        file_exp = open("Mat_exp.txt","a+")
        file_exp.write(str(temp_int)+"\n")
        file_exp.close()
        rm_output.append(temp_int)
        if(i != (tot_testcases-1)):
            await input_MATA_S(dut,P,0)
        else:
            await give_inputAs_Cz_S(dut,0,0,0,0,0)

        print("                                              TESTCASE:",i+1)

        A = []
        B = []
```

If the user chose “TEST_RANDOM_INT” the above code comes to life. It creates two matrix with random integer values and feeds it into the systolic array. This is followed by giving the same inputs to reference model and capturing the output. Then P matrix (Buffer matrix with all zeroes) is fed in to get RTL output. This process is repeated over and over until desired number of testcases as dictated by “tot_testcases” is reached.

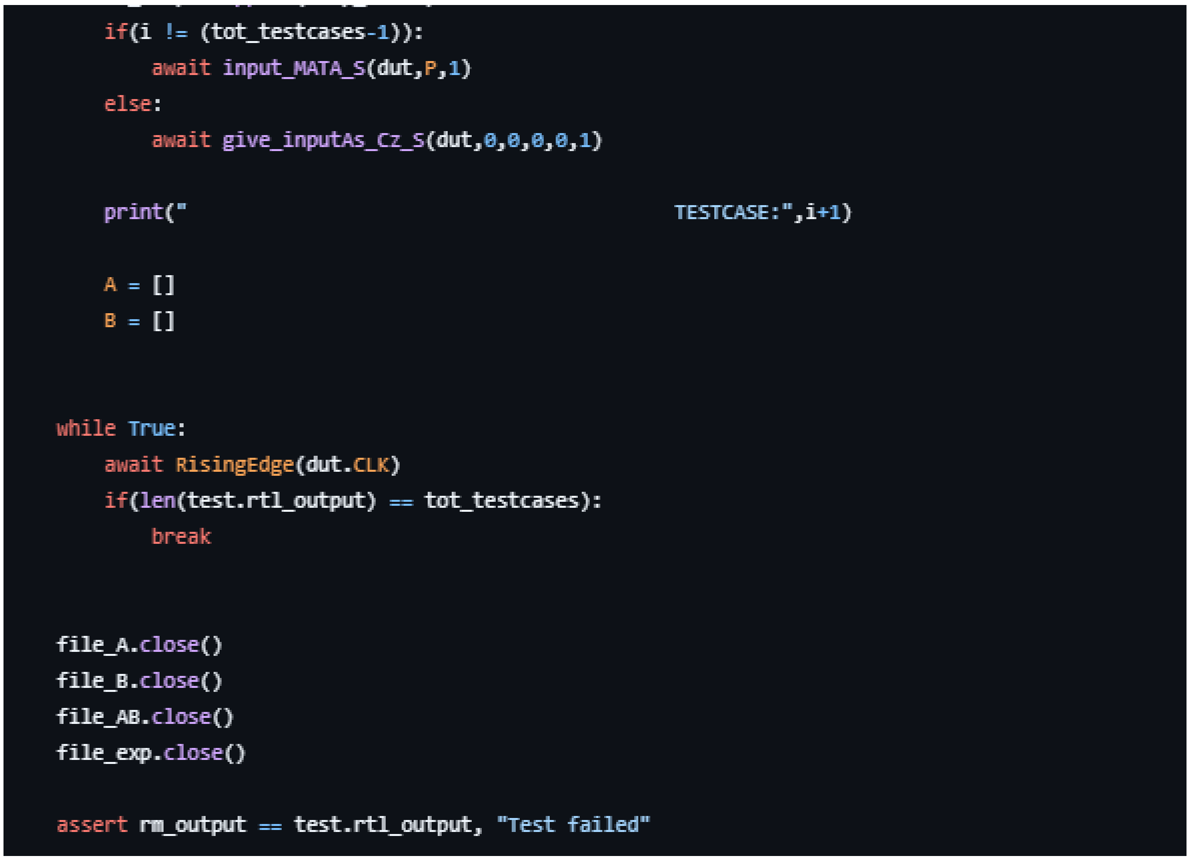


```
while True:
    await RisingEdge(dut.CLK)
    if(test.all_verified == 1):
        break


file_A.close()
file_B.close()
file_AB.close()
file_exp.close()
assert rm_output == test.rtl_output, "Test failed"
```

After providing all the inputs, we should wait for the output to get generated and must check them. The above code waits till “all_verified” variable inside finish_test class is set by dynamic_assertion function. Once the wait is over, a grand assertion is done at the end to check all the answers once again.



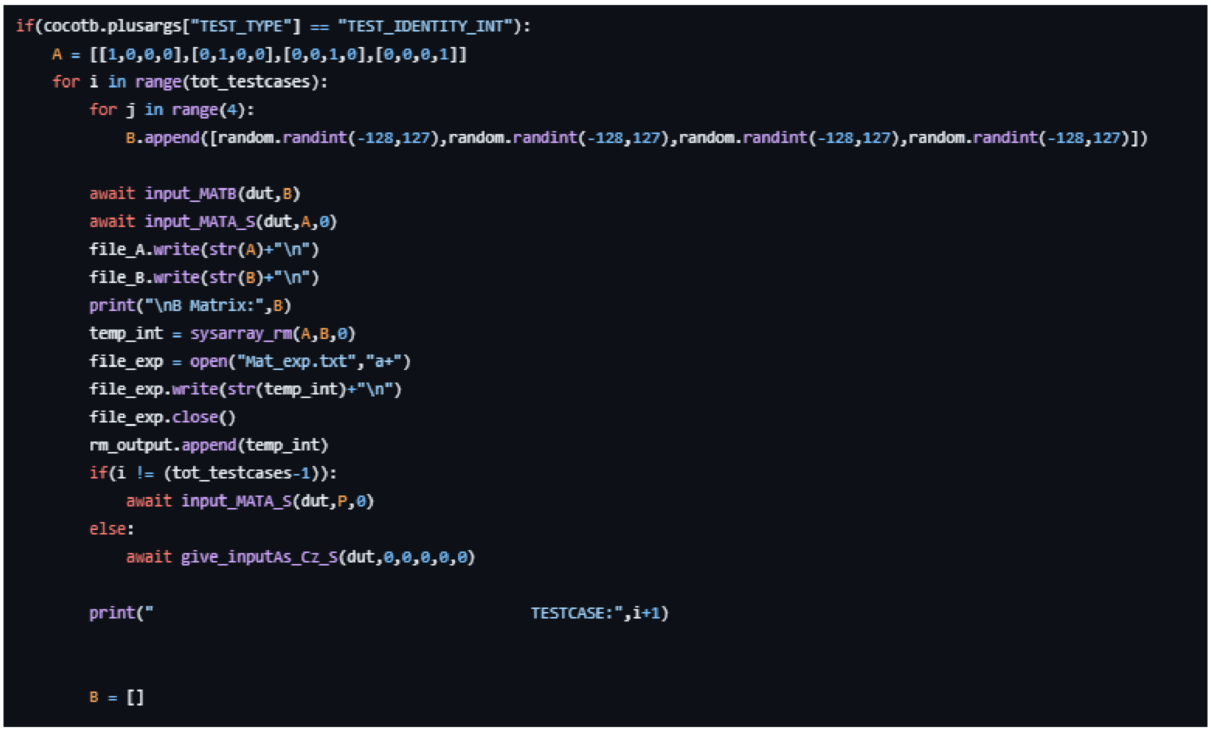
```
if(cocotb.plusargs["TEST_TYPE"] == "TEST_RANDOM_FLOAT"):

    for i in range(tot_testcases):
        while True:
            for j in range(4):
                A.append([create_random_float16(),create_random_float16(),create_random_float16(),create_random_float16()])
                B.append([create_random_float16(),create_random_float16(),create_random_float16(),create_random_float16()])
            #print("\n*******")
            RM_output = sysarray_rm(A,B,1)
            #print(RM_output)
            #print("\n*******")
            if(RM_output != "EXCEPTION"):
                break
            A = []
            B = []

        await input_MATB(dut,B)
        await input_MATA_S(dut,A,1)

        file_A.write(str(A)+"\n")
        file_B.write(str(B)+"\n")
        temp_float = sysarray_rm(A,B,1)
        file_exp = open("Mat_exp.txt","a+")
        file_exp.write(str(temp_float)+"\n")
        file_exp.close()
        rm_output.append(temp_float)
```



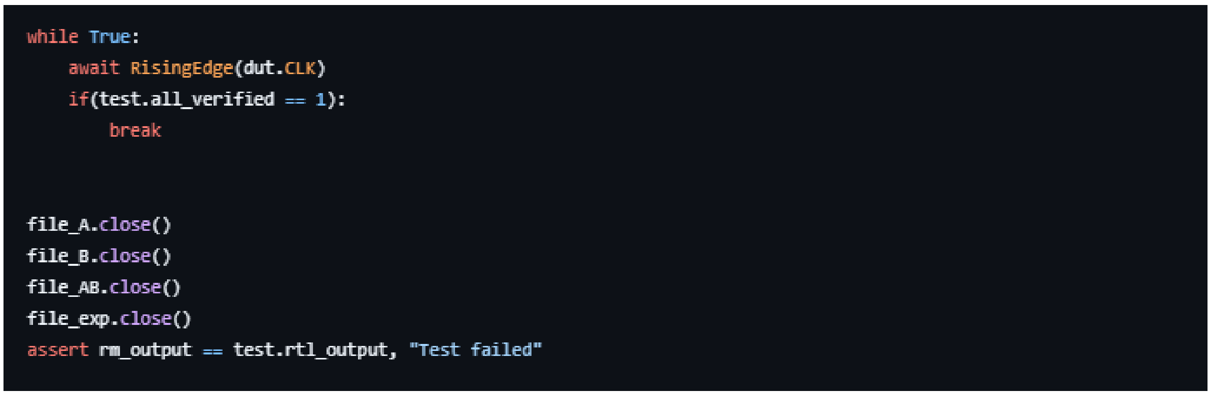
```
        if(i != (tot_testcases-1)):
            await input_MATA_S(dut,P,1)
        else:
            await give_inputAs_Cz_S(dut,0,0,0,0,1)

        print("                                        TESTCASE:",i+1)

        A = []
        B = []


    while True:
        await RisingEdge(dut.CLK)
        if(len(test.rtl_output) == tot_testcases):
            break


    file_A.close()
    file_B.close()
    file_AB.close()
    file_exp.close()

    assert rm_output == test.rtl_output, "Test failed"
```

The above images show the logic used to test “TEST_RANDOM_FLOAT”. First, random floating-point matrices are created with create_random_float16() function and sent to the reference model to see if there is any “EXCEPTION” (Due to overflow). If there is any exception, the input values are discarded and new random values are tried. This process keeps happening until a valid output is given by the reference model (without any overflow). Then only the inputs are fed to RTL as a testcase. Again, the reference model output is written to Mat_exp.txt and RTL output will be written to Mat_AB.txt. After each input is fed, the matrix P (buffer matrix with all zeroes) is fed in to get output.

After feeding in the desired number of inputs, the code waits for the RTL to provide output for all the testcases by polling test.rtl_output. Once all the output is obtained assertion is performed.



```
if(cocotb.plusargs["TEST_TYPE"] == "TEST_IDENTITY_INT"):
    A = [[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]]
    for i in range(tot_testcases):
        for j in range(4):
            B.append([random.randint(-128,127),random.randint(-128,127),random.randint(-128,127),random.randint(-128,127)])

        await input_MATB(dut,B)
        await input_MATA_S(dut,A,0)
        file_A.write(str(A)+"\n")
        file_B.write(str(B)+"\n")
        print("\nB Matrix:",B)
        temp_int = sysarray_rm(A,B,0)
        file_exp = open("Mat_exp.txt","a+")
        file_exp.write(str(temp_int)+"\n")
        file_exp.close()
        rm_output.append(temp_int)
        if(i != (tot_testcases-1)):
            await input_MATA_S(dut,P,0)
        else:
            await give_inputAs_Cz_S(dut,0,0,0,0,0)

        print("                                        TESTCASE:",i+1)


        B = []
```



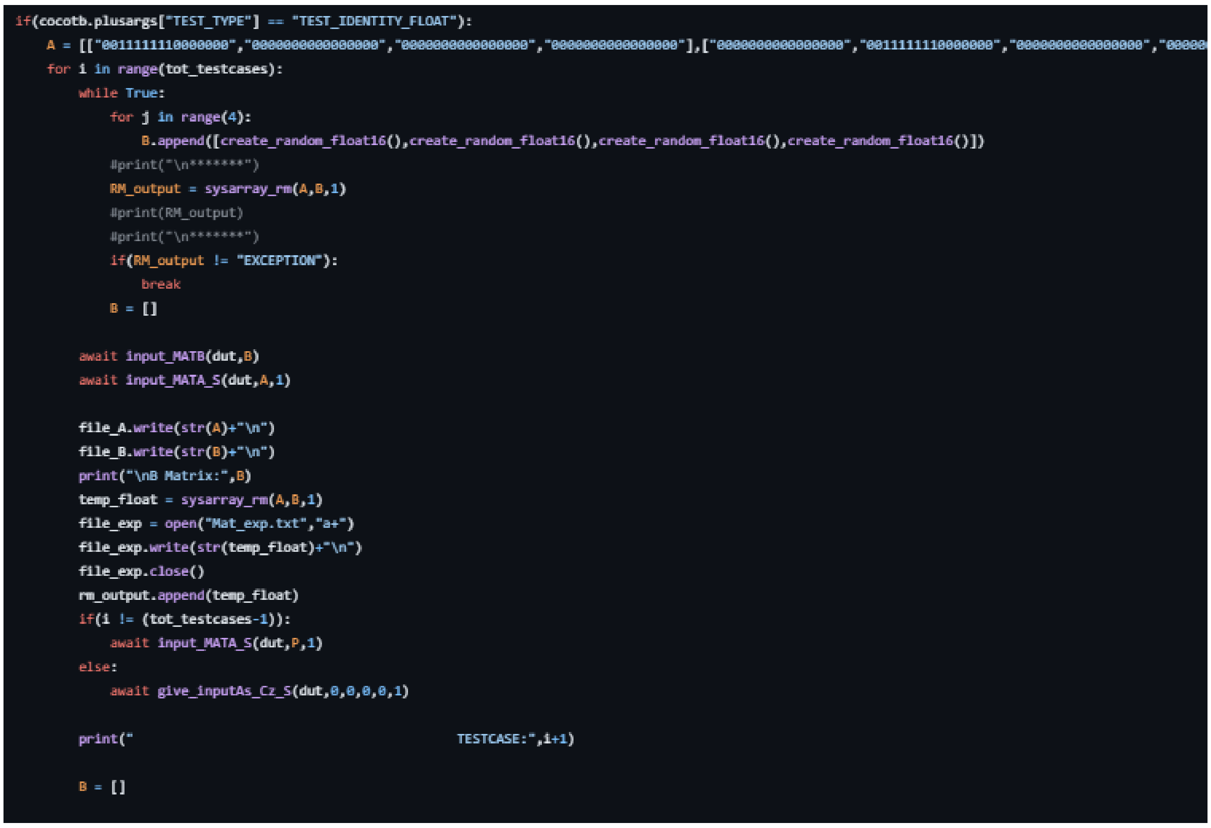
```
while True:
    await RisingEdge(dut.CLK)
    if(test.all_verified == 1):
        break


file_A.close()
file_B.close()
file_AB.close()
file_exp.close()
assert rm_output == test.rtl_output, "Test failed"
```

The above code shows how “TEST_IDENTITY_INT” is handled. In this mode, A matrix is fixed as identity matrix. And random B matrix is generated. The output of multiplication of B matrix with identity matrix will be B matrix itself. This test allows for easier verification of matrix multiplication. In “TEST_IDENTITY_INT” only integer data type is used.

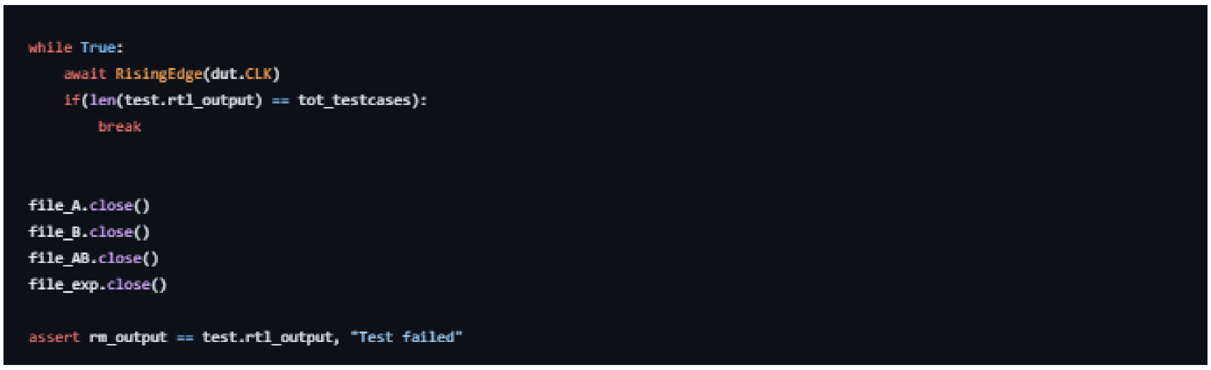
First, a random B matrix is generated with random integer values. Then matrix B and identity matrix are fed to both RTL and reference model. The outputs are written to

Mat_exp.txt (reference model) and Mat_AB.txt (RTL output). Then P matrix (Buffer matrix with all zeroes) is fed in to get RTL output. This process is repeated over and over until desired number of testcases as dictated by “tot_testcases” is reached.

After providing all the inputs, we should wait for the output to get generated and must check them. The above code waits till “all_verified” variable inside finish_test class is set by dynamic_assertion function. Once the wait is over, a grand assertion is done at the end to check all the answers once again.



```
if(cocotb.plusargs["TEST_TYPE"] == "TEST_IDENTITY_FLOAT"):
    A = [["0011111110000000","0000000000000000","0000000000000000","0000000000000000"],["0000000000000000","0011111110000000","0000000000000000","00000
    for i in range(tot_testcases):
        while True:
            for j in range(4):
                B.append([create_random_float16(),create_random_float16(),create_random_float16(),create_random_float16()])
            #print("\n*******")
            RM_output = sysarray_rm(A,B,1)
            #print(RM_output)
            #print("\n*******")
            if(RM_output != "EXCEPTION"):
                break
            B = []

        await input_MATB(dut,B)
        await input_MATA_S(dut,A,1)

        file_A.write(str(A)+"\n")
        file_B.write(str(B)+"\n")
        print("\nB Matrix:",B)
        temp_float = sysarray_rm(A,B,1)
        file_exp = open("Mat_exp.txt","a+")
        file_exp.write(str(temp_float)+"\n")
        file_exp.close()
        rm_output.append(temp_float)
        if(i != (tot_testcases-1)):
            await input_MATA_S(dut,P,1)
        else:
            await give_inputAs_Cz_S(dut,0,0,0,0,1)

        print("                                                TESTCASE:",i+1)

        B = []
```



```
while True:
    await RisingEdge(dut.CLK)
    if(len(test.rtl_output) == tot_testcases):
        break


file_A.close()
file_B.close()
file_AB.close()
file_exp.close()

assert rm_output == test.rtl_output, "Test failed"
```

The above code shows how “TEST_IDENTITY_FLOAT” is handled. In this mode, A matrix is fixed as identity matrix in bfloat16 format. And random B matrix is generated. The output of multiplication of B matrix with identity matrix will be B matrix itself. This test allows for easier verification of matrix multiplication. In “TEST_IDENTITY_FLOAT” only bfloat16 data type is used.

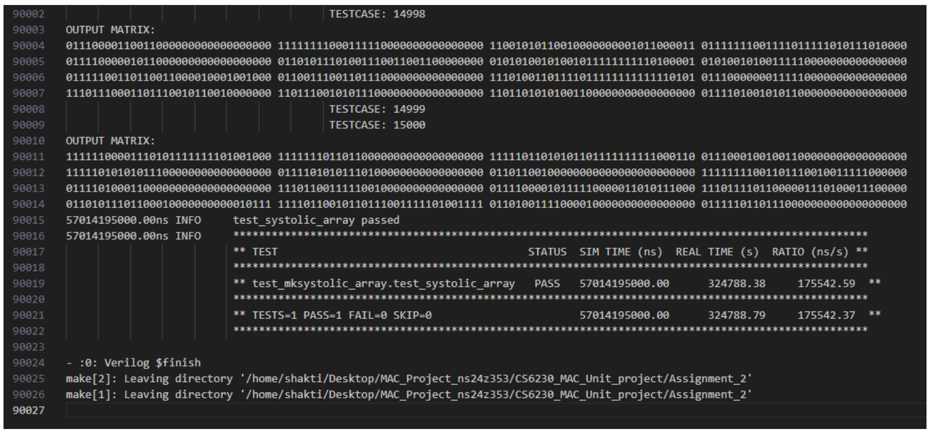
First, a random B matrix is generated with random bfloat16 values. Then matrix B and identity matrix are fed to reference model to check whether any overflow exceptions occur. If any exception are found, new random float values are tried. This process is repeated until reference model gives valid non-exception values. Then only the inputs are fed to RTL as a testcase. Again, the reference model output is written to Mat_exp.txt and RTL output will be written to Mat_AB.txt. After each input is fed, the matrix P (buffer matrix with all zeroes) is fed in to get output.

After feeding in the desired number of inputs, the code waits for the RTL to provide output for all the testcases by polling test.rtl_output. Once all the output is obtained assertion is performed.

<u>RESULT:</u>

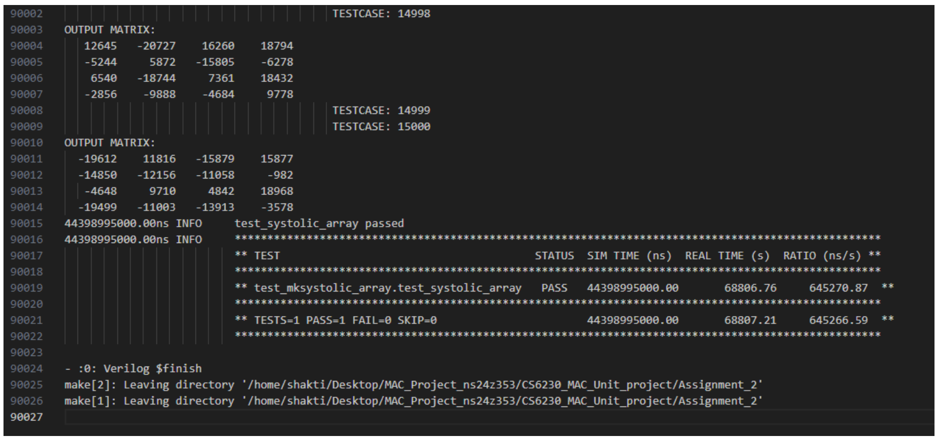
The systolic array is tested extensively by running 15,000 testcases for FLOAT and 15,000 testcases for INT. All 30,000 testcases passed with no problems.

The following image shows the 15,000 testcases passing for FLOAT testcases:



```
                                                      TESTCASE: 14998
OUTPUT MATRIX:
01110000110011000000000000000000 11111111000111110000000000000000 11001010110010000000001011000011 01111111001111011111010111010000
01111000000101100000000000000000 01101011101001110011001100000000 01010100101001011111111110100001 01010010100111110000000000000000
01111100110110011000010001001000 01100111001101110000000000000000 11101001101111011111111111110101 01110000000111110000000000000000
11101110001101110010110010000000 11011100101011100000000000000000 11011010101001100000000000000000 01111010010101100000000000000000
                                                      TESTCASE: 14999
                                                      TESTCASE: 15000
OUTPUT MATRIX:
11111100001110101111111101001000 11111110110110000000000000000000 11111011010101101111111111000110 01110001001001100000000000000000
11111010101011100000000000000000 01111010101110100000000000000000 01101100100000000000000000000000 11111111001101110010011111000000
01111010001100000000000000000000 11101100111110010000000000000000 01111000010111110000011010111000 11101111011000001110100011100000
01101011101100010000000000010111 11110110010110111001111101001111 01101001111000010000000000000000 01111101101110000000000000000000
57014195000.00ns INFO     test_systolic_array passed
57014195000.00ns INFO     **************************************************************************************
                          ** TEST                                        STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                          **************************************************************************************
                          ** test_mksystolic_array.test_systolic_array   PASS   57014195000.00      324788.38     175542.59  **
                          **************************************************************************************
                          ** TESTS=1 PASS=1 FAIL=0 SKIP=0                       57014195000.00      324788.79     175542.37  **
                          **************************************************************************************

- :0: Verilog $finish
make[2]: Leaving directory '/home/shakti/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Assignment_2'
make[1]: Leaving directory '/home/shakti/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Assignment_2'
```

The following image shows the 15,000 testcases passing for INT testcases:



```
                                                      TESTCASE: 14998
OUTPUT MATRIX:
   12645   -20727    16260    18794
   -5244     5872   -15805    -6278
    6540   -18744     7361    18432
   -2856    -9888    -4684     9778
                                                      TESTCASE: 14999
                                                      TESTCASE: 15000
OUTPUT MATRIX:
  -19612    11816   -15879    15877
  -14850   -12156   -11058     -982
   -4648     9710     4842    18968
  -19499   -11003   -13913    -3578
44398995000.00ns INFO     test_systolic_array passed
44398995000.00ns INFO     **************************************************************************************
                          ** TEST                                        STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                          **************************************************************************************
                          ** test_mksystolic_array.test_systolic_array   PASS   44398995000.00       68806.76     645270.87  **
                          **************************************************************************************
                          ** TESTS=1 PASS=1 FAIL=0 SKIP=0                       44398995000.00       68807.21     645266.59  **
                          **************************************************************************************

- :0: Verilog $finish
make[2]: Leaving directory '/home/shakti/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Assignment_2'
make[1]: Leaving directory '/home/shakti/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Assignment_2'
```

The log files which captured the above two simulation runs are present in FLOAT_SYSTOLIC_ARRAY_RESULTS/ folder and INT_SYSTOLIC_ARRAY_RESULTS/ folder. They contain text files which contains all the inputs generated, all the reference model outputs and all RTL outputs as proof of working.