Homework 5 – SystemC High-Level Modeling and Communication Modeling

Taghizadeh,
810198373

I. Introduction

In this homework, we are about to make a full digital system containing processor, data memory and some peripherals like an intermediate component and 16-bit multiplier which are connected via a FIFO queue channel.

The processor is going to execute the following program:

```
Write 128 32-bit data to data memory
Read 128 32-bit data from data memory
Loop 4 times:
        if (FIFO queue channel is empty) {
                Write a 32 32-bit data into the
                        intermediate component
                issue start of multiplier
                do something else
                Loop 32 times:
                        If ( mult result is ready)
                                Read mult result
                        Else
                                do something else
                end
        }
        else
                do something else

end
```

II. Digital System

A. *Intermediate Component in SystemC*

The intermediate component consists of two registers, one is control register and the other is data register. Input output data and control signals are declared as in the front image:

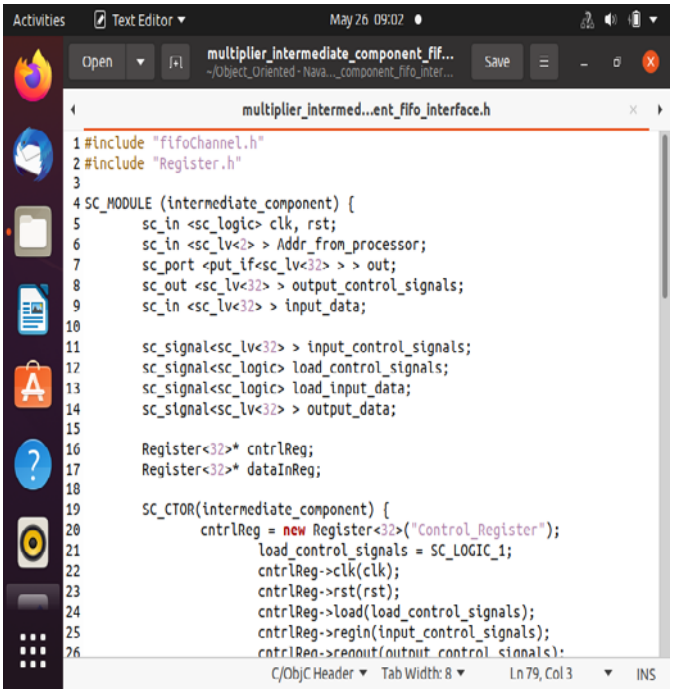

Fig. 1 *Intermediate component module SystemC*

```
26                  cntrlReg->regout(output_control_signals);
27
28              dataInReg = new Register<32>("Data_Register");
29                  load_input_data = SC_LOGIC_1;
30                  dataInReg->clk(clk);
31                  dataInReg->rst(rst);
32                  dataInReg->load(load_input_data);
33                  dataInReg->regin(input_data);
34                  dataInReg->regout(output_data);
35
36              SC_THREAD (put_operands);
37                  sensitive << clk << rst;
38      }
39      void put_operands();
40 };
41
```

Fig. 2 *Intermediate component module SystemC*

"*input_data*" is defined as input signal to receive the incoming data from the processor.

"*output_control_signals*" is defined as output signal to inform the processor that FIFO is empty or not by the first bit of "*output_control_signals*".

"*Addr_from_processor*" is for communication between this module and the processor. Whenever processor wants to communicate with this module, it initialises "*Addr_from_processor*" to "01" (which is intermediate component's identifier).

"*out*" is put interface which is used to send incoming data to the queue channel.

"*put_operands*" method is used to put the incoming data from the processor into the queue channel to store data in FIFO.

"*put_operands*" method is sensitive to clock. So each posedge of clock this method is called. At first we wait until posedge of clock occurs by "wait()". If "*Addr_from_processor*" is equal to "01" then we realise that processor wants to send data to *intermediate component*. First we check if FIFO is empty. Then we must inform the processor that FIFO is empty by initialising the first bit of "*input_control_signals*"(which is registered with "*output_control_signals*") . So processor will send its data. In the end we put the incoming data into FIFO about 32 times by a loop. Each time data is put into the FIFO, we must wait about a clock (which has 100 NS period) to put data clock by clock.

```
#include "multiplier_intermediate_component_fifo_interface.h"

void intermediate_component::put_operands() {
    while(true){
        wait();
        if(Addr_from_processor.read() == "01"){
            if(out->is_fifo_empty()){
                input_control_signals = "00000000000000000000000000000001";
                wait(300, SC_NS);
                for (int i = 0; i < 32; i++)
                {
                    wait(100, SC_NS);
                    out->put(output_data);
                    cout << "Data: (" << output_data << ") was transmitted
                        << sc_time_stamp() << '\n';
                }
            }
            else
                input_control_signals = "00000000000000000000000000000000";
        }
    }
}
```

Fig. 3 *Intermediate component put_operands method*

B. *16-bit Multiplier in SystemC*

The 16-bit multiplier consists of two registers, one is control register and the other is data register. Input output data and control signals are declared as in the below image:

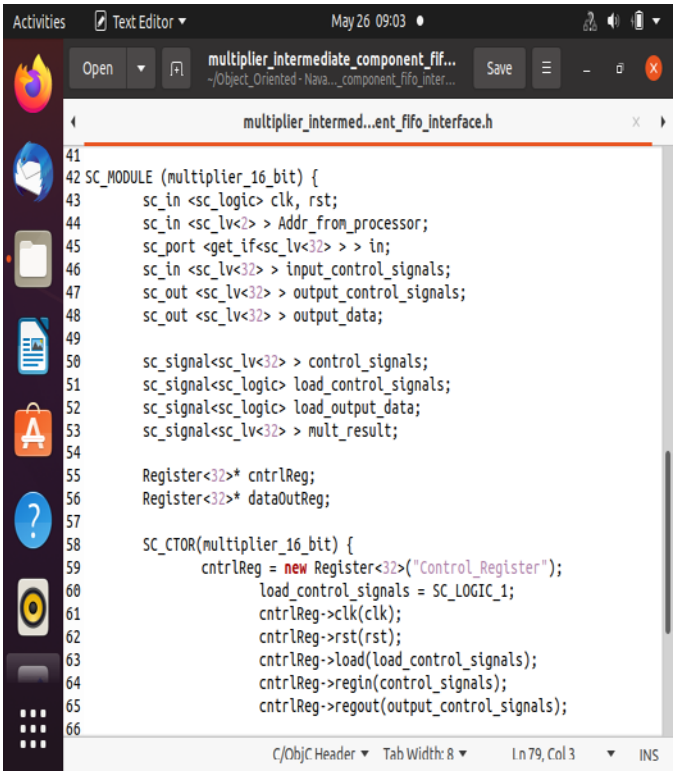

Fig. 4 *16-bit Multiplier module SystemC*

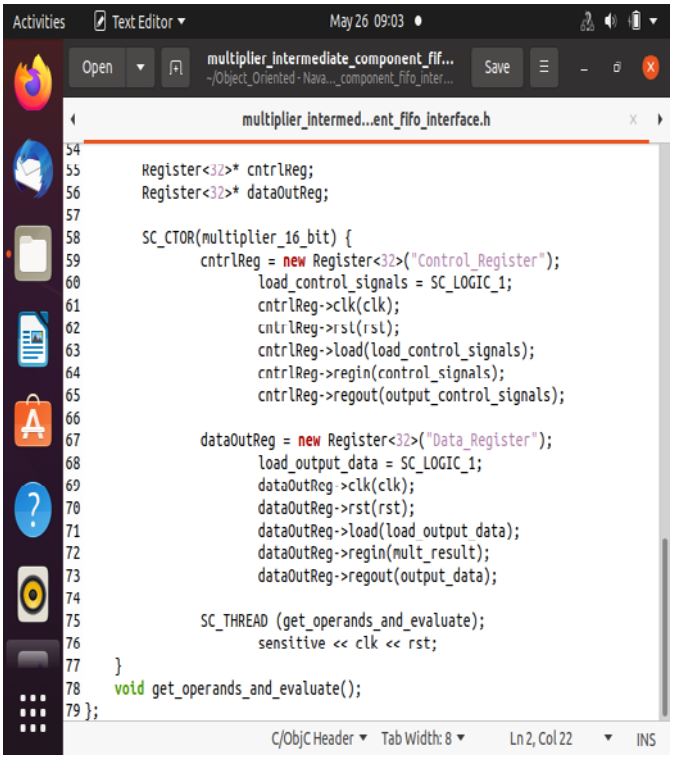

Fig. 5 *16-bit Multiplier module SystemC*

“*output_data*” is defined as output signal to send the multiplication result to the processor.

“*input_control_signals*” is defined as input signal to inform the multiplier to start the multiplication by the “*start*” bit (which is the first bit of “*input_control_signals*) which is initialised by the processor.

“*output_control_signals*” is defined as output signal to inform the processor that multiplication result is ready by the “*ready*” bit (which is the second bit of “*output_control_signals*) which is initialised by the multiplier.

“*Addr_from_processor*” is for communication between this module and the processor. Whenever processor wants to communicate with this module, it initialises “*Addr_from_processor*” to “10” (which is 16-bit multiplier’s identifier).

“*in*” is get interface which is used to get the operands from FIFO queue channel to calculate the multiplication.

“*get_operand_and_evaluate*” method is used to get operands from FIFO and evaluate the multiplication of the operands.

“*get_operand_and_evaluate*” method is sensitive to clock. So each posedge of clock this method is called. At first we wait until posedge of clock occurs by “wait()”. If “*Addr_from_processor*” is equal to “10” then we realise that processor wants to communicate to *16-bit multiplier*. First we check if start is issued by the processor (which is the first bit of “*input_control_signals*”) after that we get the operands by “*get*” method which belongs to “*in*” port. By “*range*” method, data from FIFO is split into two 16 bits data, which are two operands of multiplication. In the end two operands are multiplied and stored in “*mult_result*” (which is registered with “*output_data*”) to send multiplication result to the processor. At least 0 SC_NS must be passed to see updated value of “*mult*_result”, so we wait (100 NS) to store the updated value of “*mult_result*” into “*output_data*”. To inform the processor that the multiplication result is ready, the “ready” bit (which is the second bit of “*control_signals*”, which is registered with “*output_control_signals*”) is initialised.

Fig. 6 *16-bit Multiplier get_operands_and_evaluate meyjod*

C. FIFO Queue channel in SystemC

We simply used ”*FIFO put get interface queue channel*” provided in Channel_codes.zip. The “*is_fifo_empty*” method is added, which simply counts number of elements stored in FIFO to check if FIFO is empty or not.

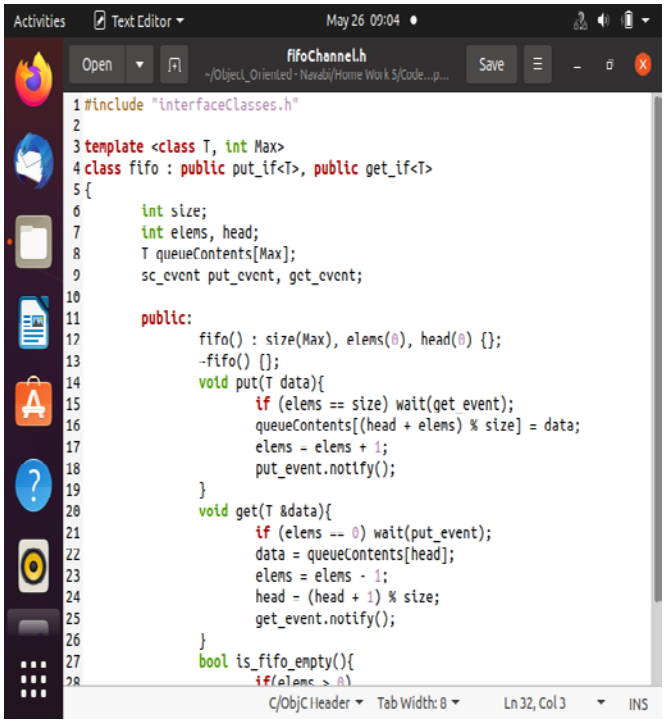

Fig. 7 FIFO Queue channel *module SystemC*

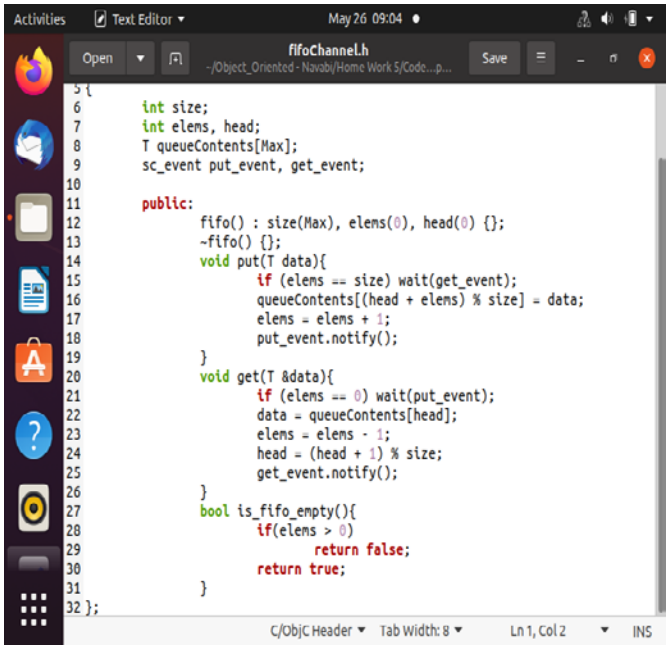

Fig. 8 FIFO Queue channel *module SystemC*

D. Intermediate component, Multiplier and FIFO Test Bench

Simply instances of Intermediate component, 16-bit multiplier and FIFO queue channel are taken. *Intermediate component* must be attached to ”*out*” (which is put interface of queue FIFO channel), to put processor’s data into FIFO, whereas *16-bit Multiplier* must be attached to “*in*” (which is get interface of queue FIFO channel) to get data(operands) from FIFO. “*inputting*”, “*clocking*” and “*resetting*” methods are used to generate test case, generate clock signal and reset respectfully.

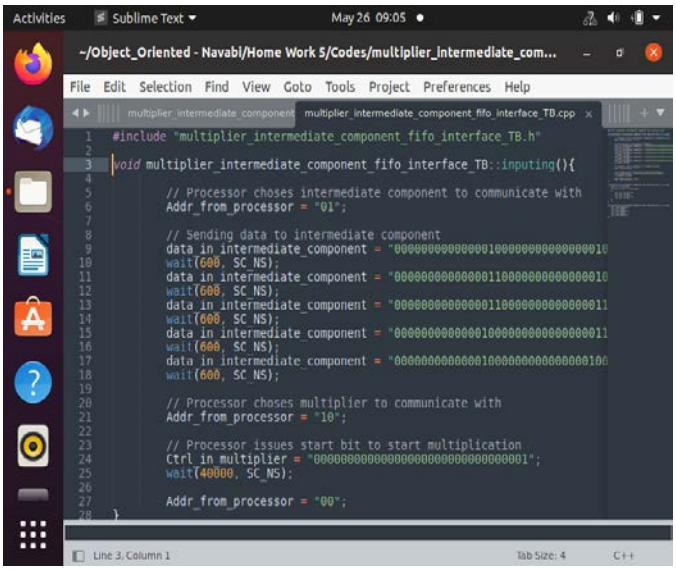

Fig. 9 Intermediate, Multiplier and FIFO *Test Bench inputting method*

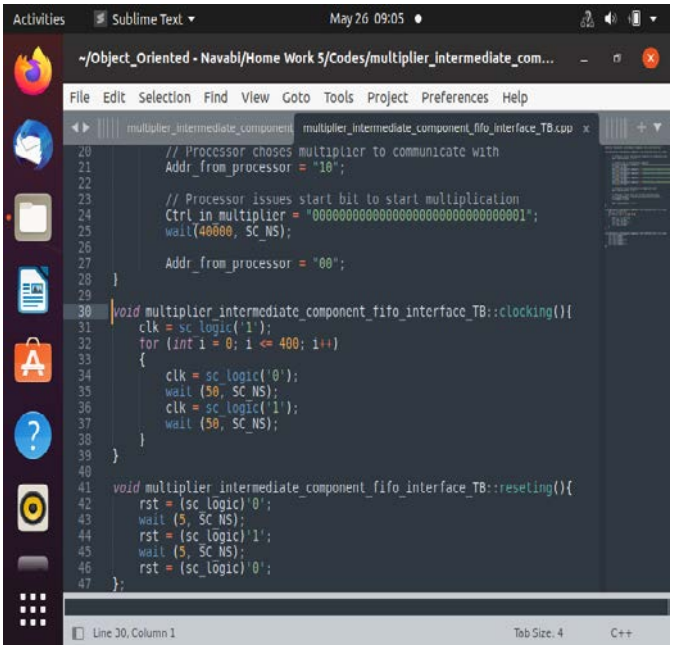

Fig. 10 Intermediate, Multiplier and FIFO *Test Bench clocking resetting method*

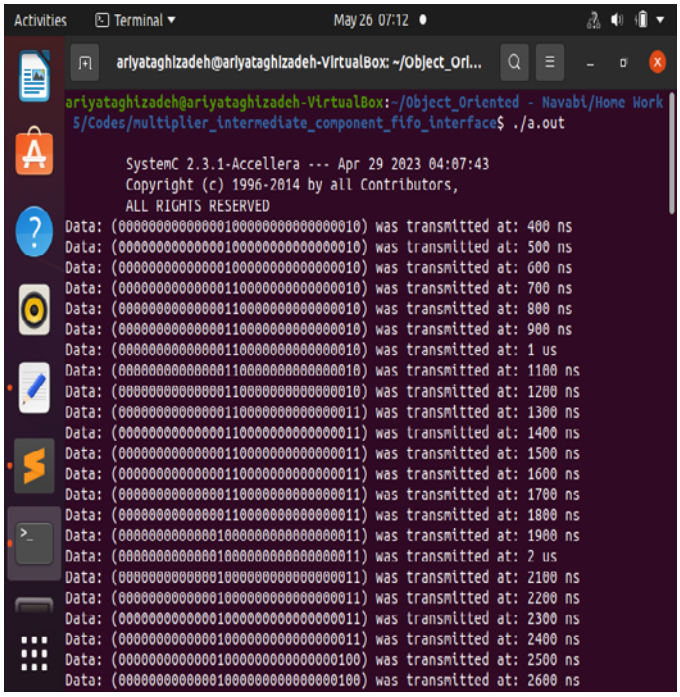

Fig. 11 *Intermediate Component Multiplier FIFO interface Test Bench results*

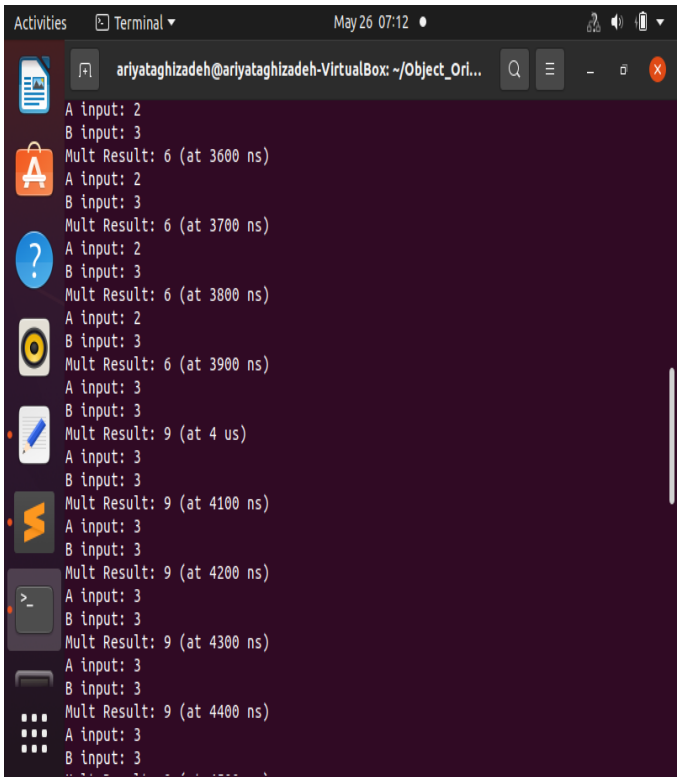

Fig. 12 *Intermediate component Multiplier FIFO interface Test Bench results*

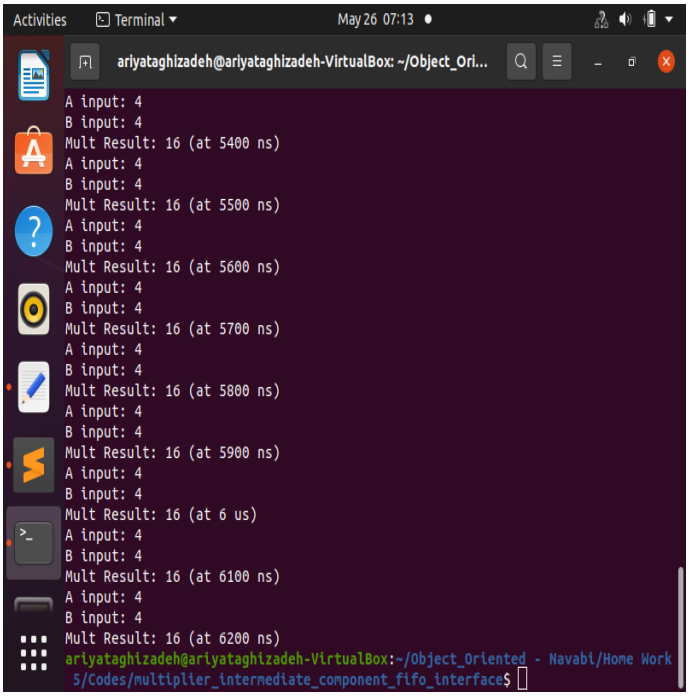

Fig. 13 *Intermediate component Multiplier FIFO interface Test Bench results*

E. Memory in SystemC

We simply used "*Memory Export*" provided in Channel_codes.zip. First a memory with 256 32 bit elements is defined and initialised by a loop:

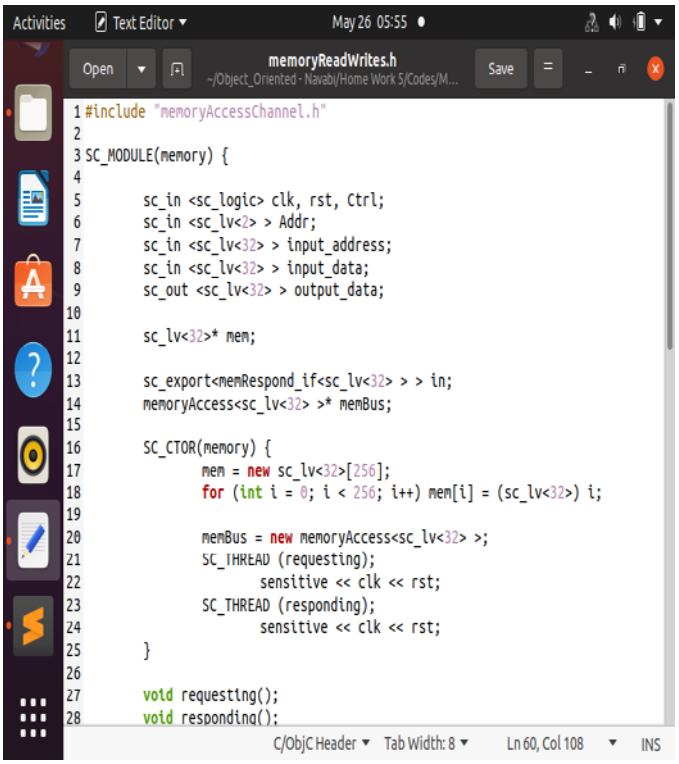

Fig. 14 *Memory module SystemC*

"*requesting*" method is used to request to the memory to load/store data, which uses memBus interface's "*requestMem*" method to request to memory. Whenever "*Addr_from_processor*" is equal to "00" (which is Memory's identifier). Processor wants to send/receive data from memory so "*requesting*" method is called to inform the memory that processor has a request from memory.

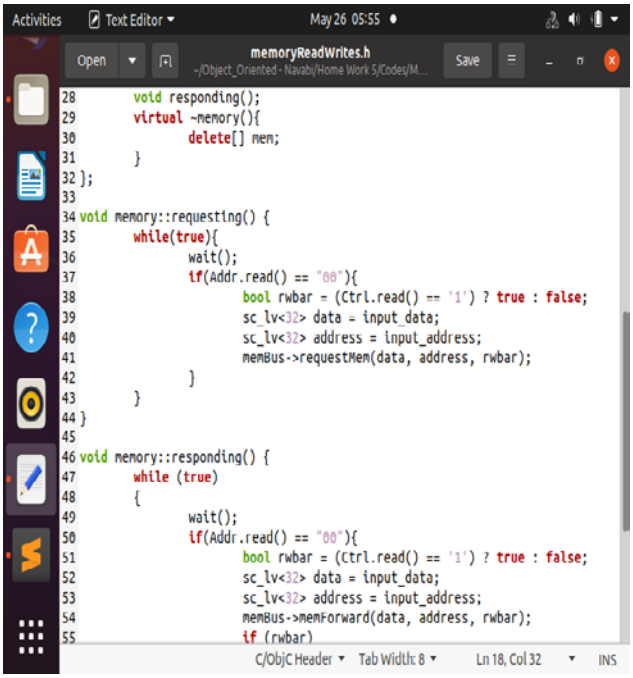

Fig. 15 *Memory requesting method*

"*responding*" method is used to respond to processor. If "*rwbar*" (which is "*Ctrl*" input signal equivalent Boolean value that determines which of read or write is requested by processor) is "1" so processor wants to read from memory else processor wants to write some data in memory. The below images shows how read and write from memory is handled by "*responding*" method:

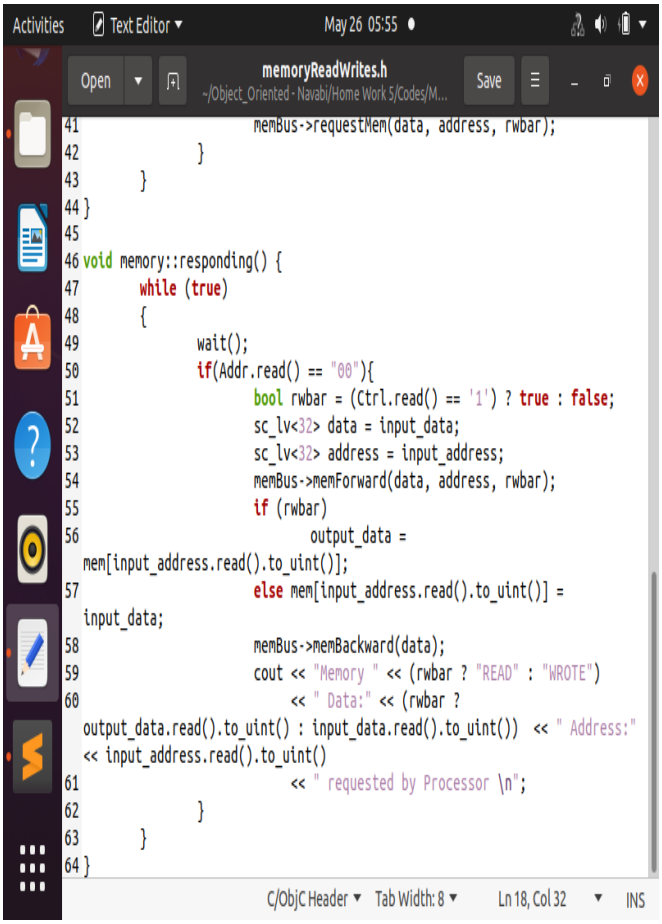

Fig. 16 *Memory responding method*

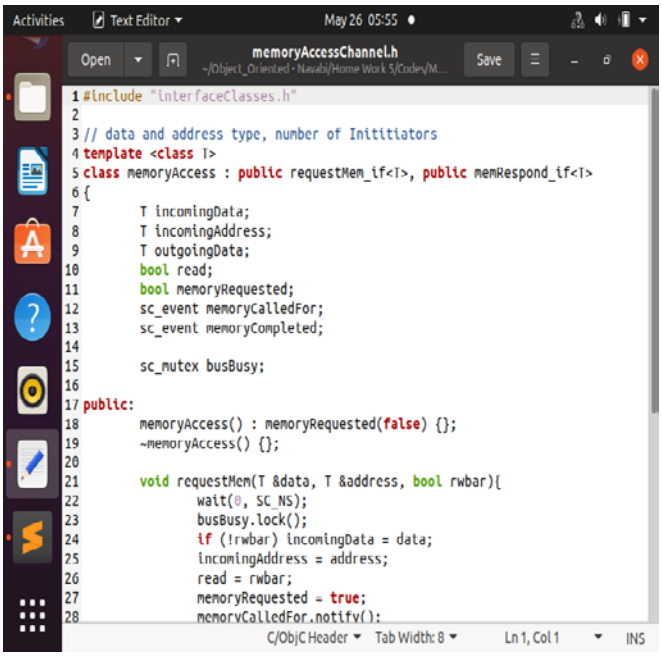

Fig. 17 *Memory memroyAccess interface*

Fig. 18 *Memory memroyAccess interface* methods

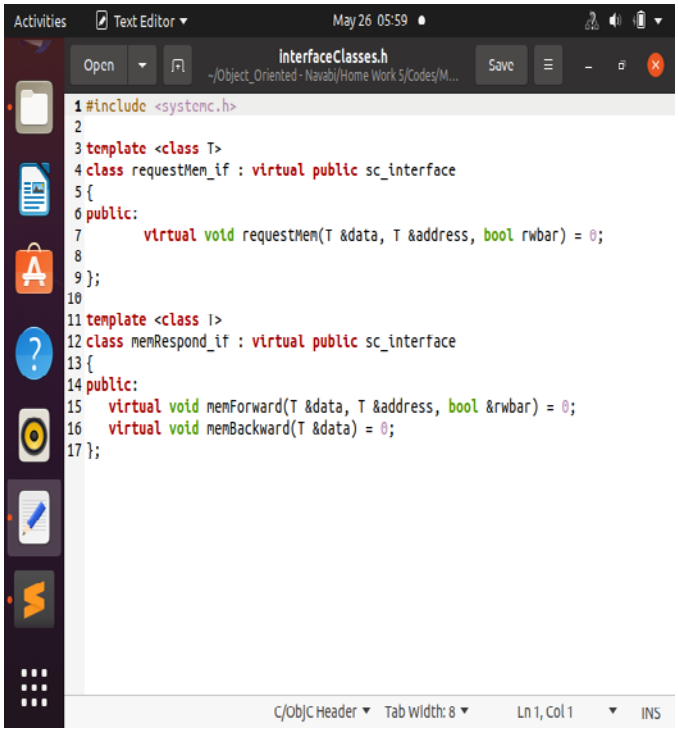

Fig. 19 *Memory requestMem and memRespond interfaces*

F. *Memory Test Bench*

Simply instance of memory is taken. "*inputting*", "*clocking*" and "*resetting*" methods are used to generate test case, generate clock signal and reset respectfully.

Fig. 20 *Memory Test Bench*

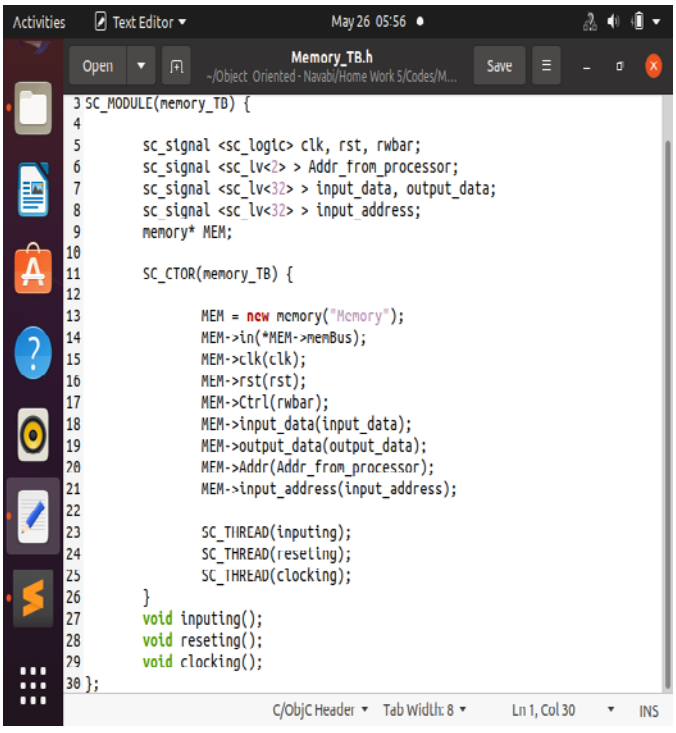

Fig. 21 *Memory Test Bench*

In “*inputting*”, address (which we want to read from or write to) is increased in each iteration of a loop, so by the end of this loop all data are read/write from/to addresses 0000_0008 till 0000_00087. In this test bench memory input data starts from 0 and is increased in each iteration of a loop, so data will be from 0 to 127. we must wait about a clock (which has 100 NS period) to read/write data clock by clock and be synchronous.

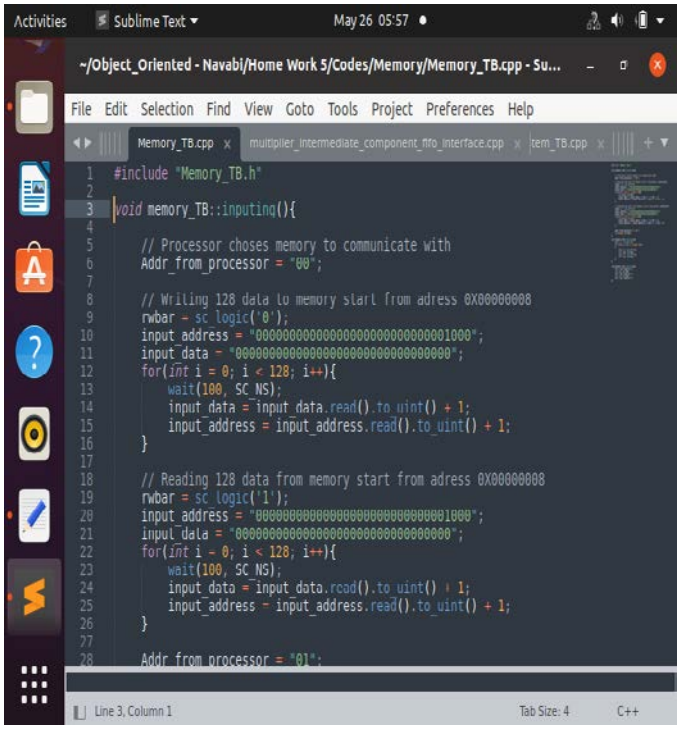

Fig. 22 *Memory Test Bench inputting method*

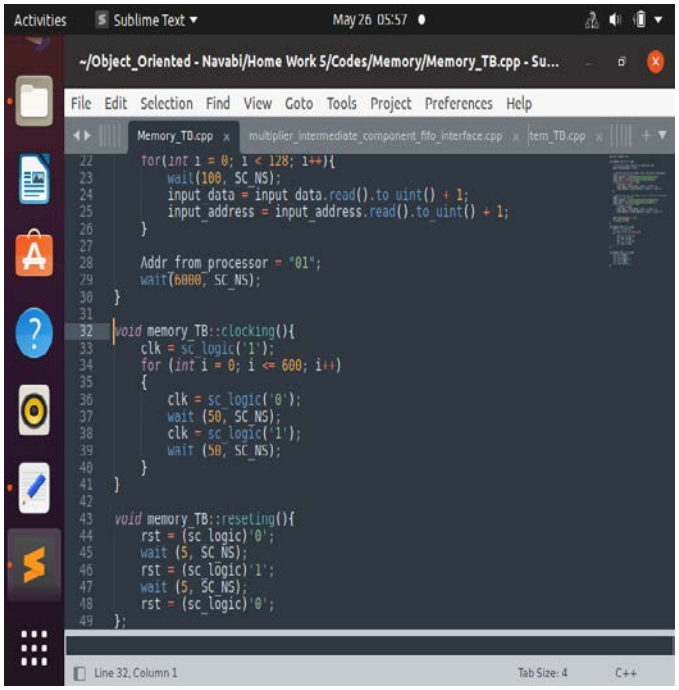

Fig. 23 *Memory Test Bench clocking and resetting methods*

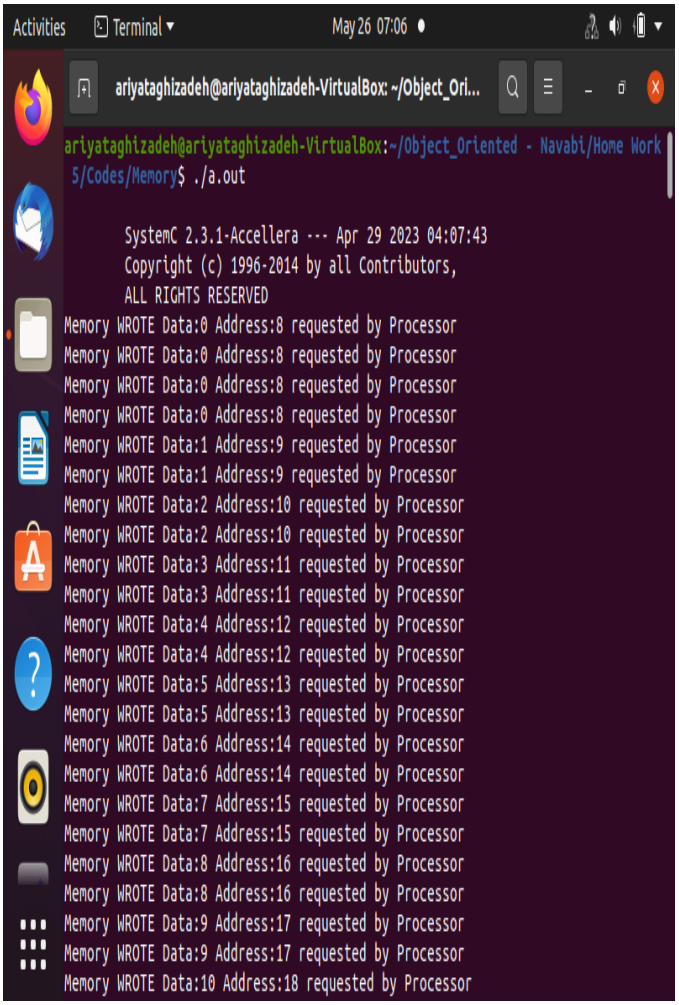

Fig. 24 *Memory Test Bench results*

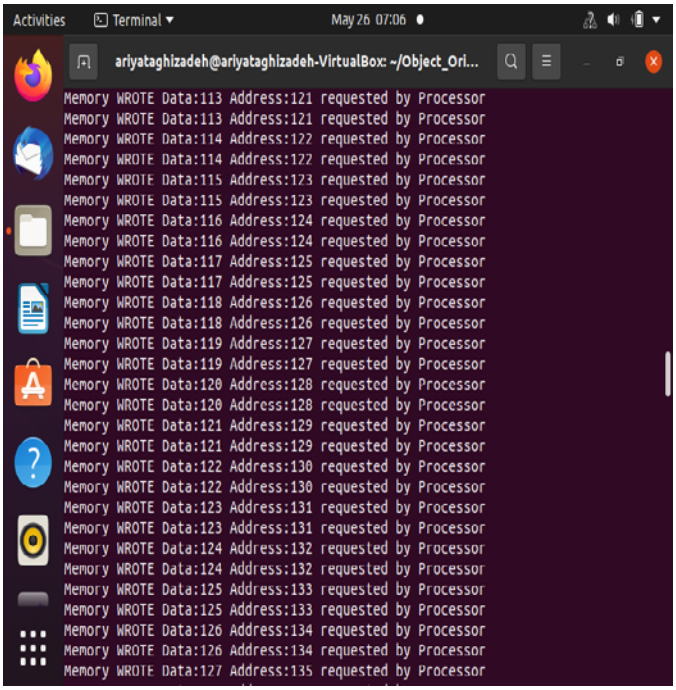

Fig. 25 *Memory Test Bench results*

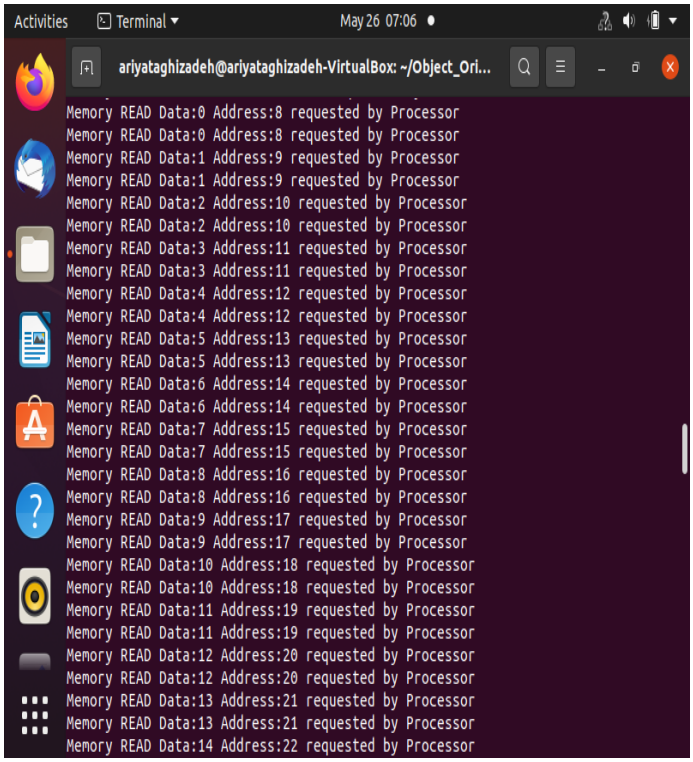

Fig. 26 *Memory Test Bench results*

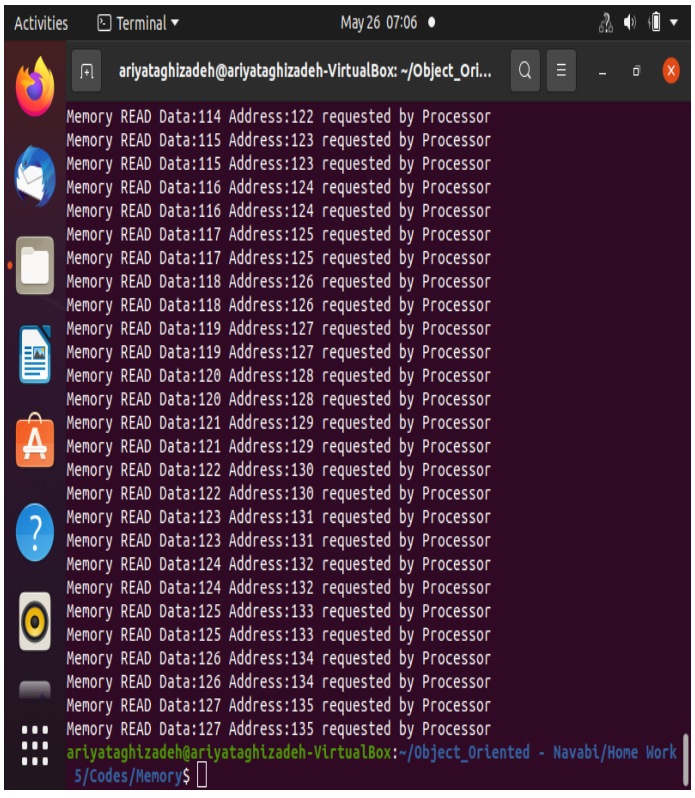

Fig. 27 *Memory Test Bench results*

G. *Bus Iinterface*

All components are in communicate with Bus interface, so all input/output of modules are declared in "*Bus*" as shown in the below images:

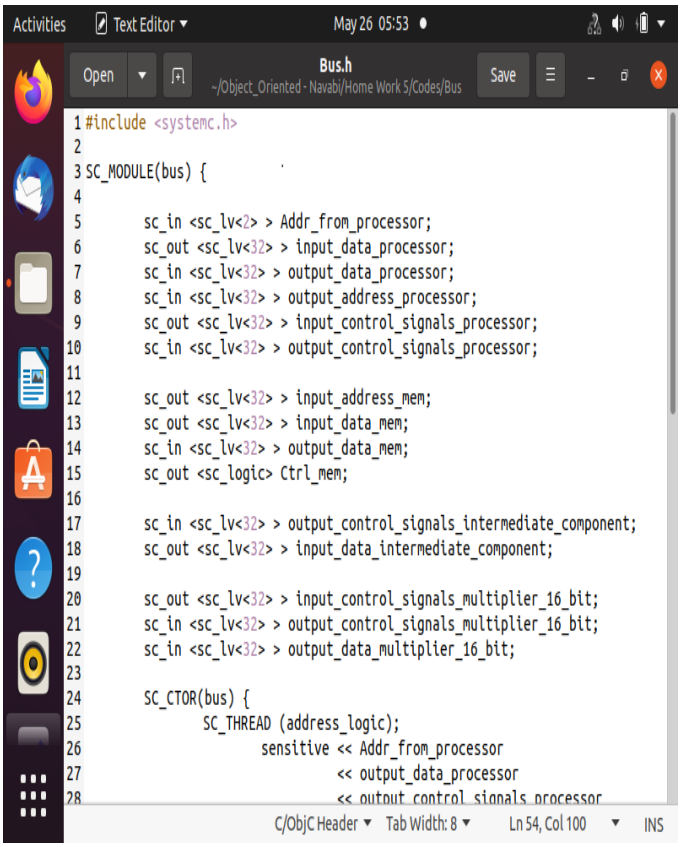

Fig. 28 *Bus Interface module SystemC*

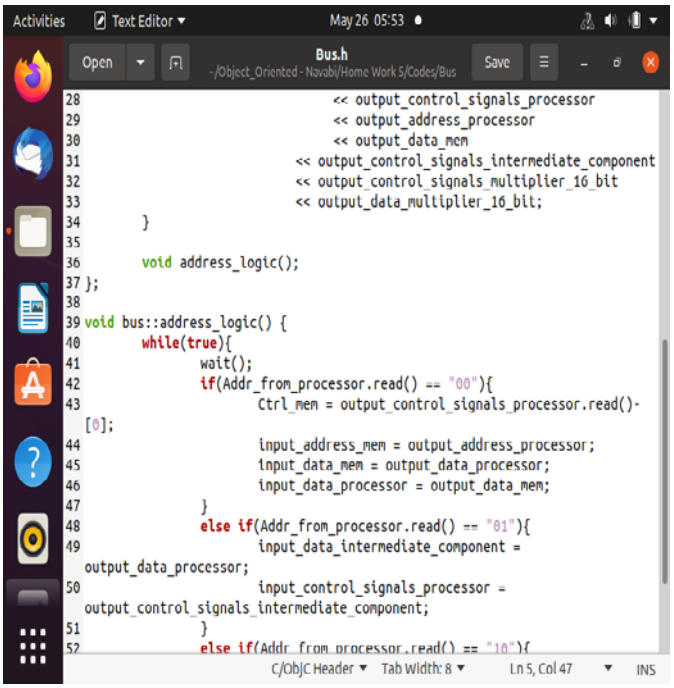

Fig. 29 *Bus Interface SystemC*

"*address_logic*" method is used to decode "*Addr_from_processor*" to identify which module does processor wants to communicate with ("00" means that processor wants to communicate with "*memory*", "01" is for "*intermediate component*" and "10" is for "*16-bit Multiplier*"). After we realised that which module does processor wants to communicate, we simply connect data and control input/output signals of that module to processor data and control input/output signals. So first we identify which module does processor wants to communicate with, then we connect that module to the processor. So the processor and that module can send data to and handshaking with each other.

Fig. 30 *Bus Interface address_logic method*

H. Processor

To store read data from memory an embedded Register File (depth = 128, word = 32 bit) is defined in processor. "*processor_function*" method is the main program that processor is going to execute:

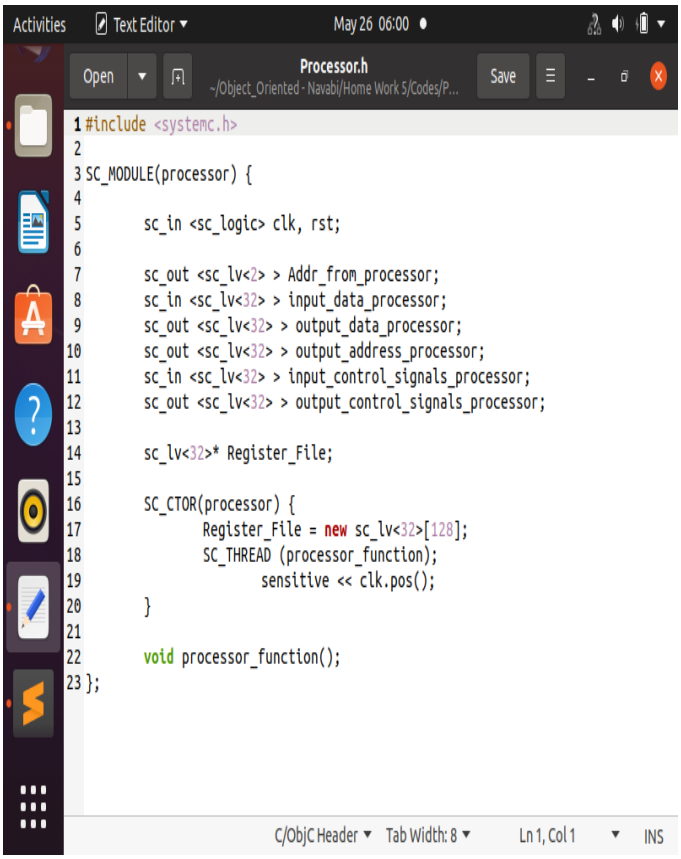

Fig. 31 *Processor module SystemC*

"*rwbar*" (the first bit of "*out_put_control_signal_processors*") must be '0' for writing to memory and '1' for reading from memory:

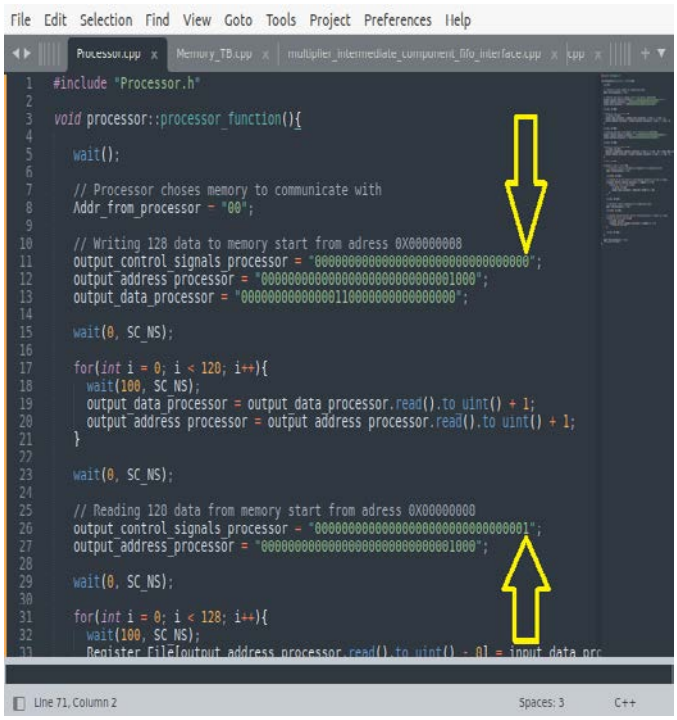

Fig. 32 *Processor processor_function method*

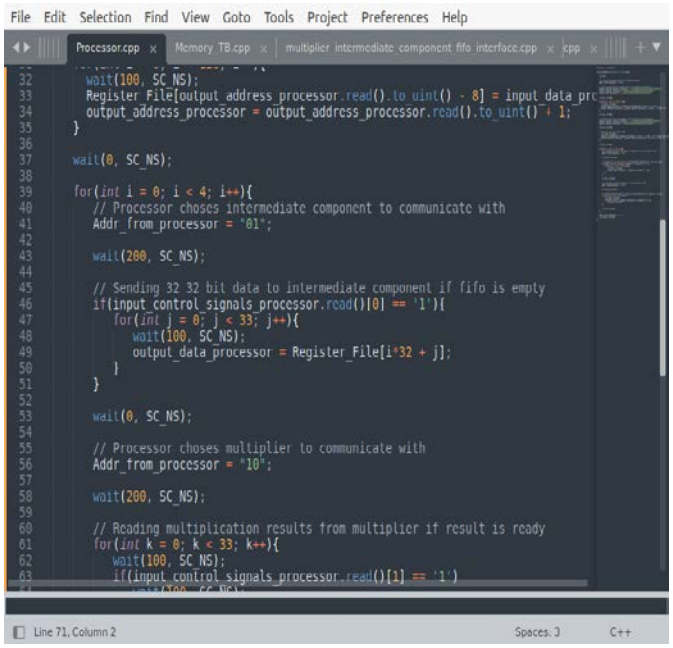

Fig. 33 *Processor processor_function method*

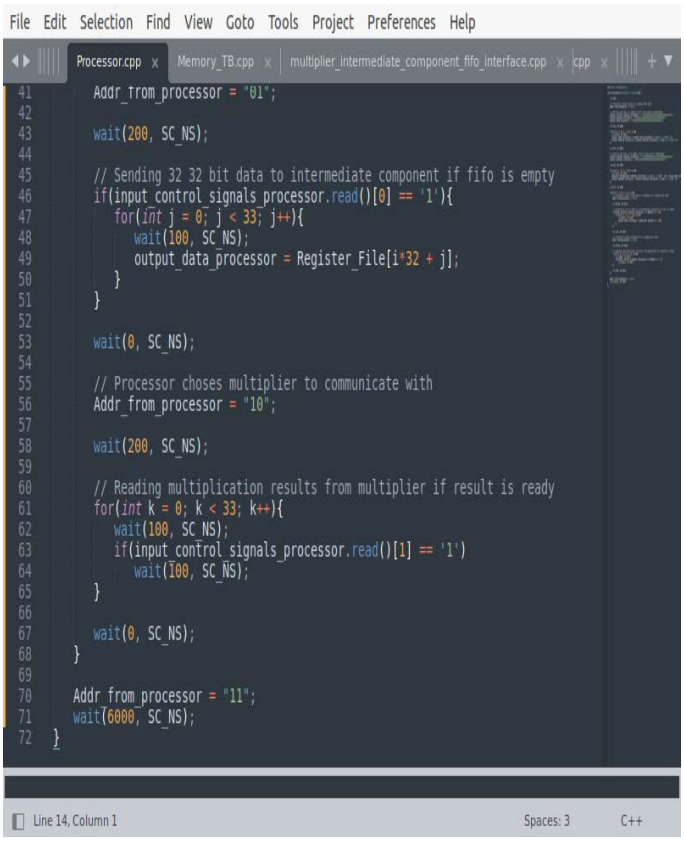

Fig. 34 *Processor processor_function method*

I. *Digital System*

Here the entire digital system is made by putting all above components together. Simply we just take some instances from all above modules and connect them to each other properly.

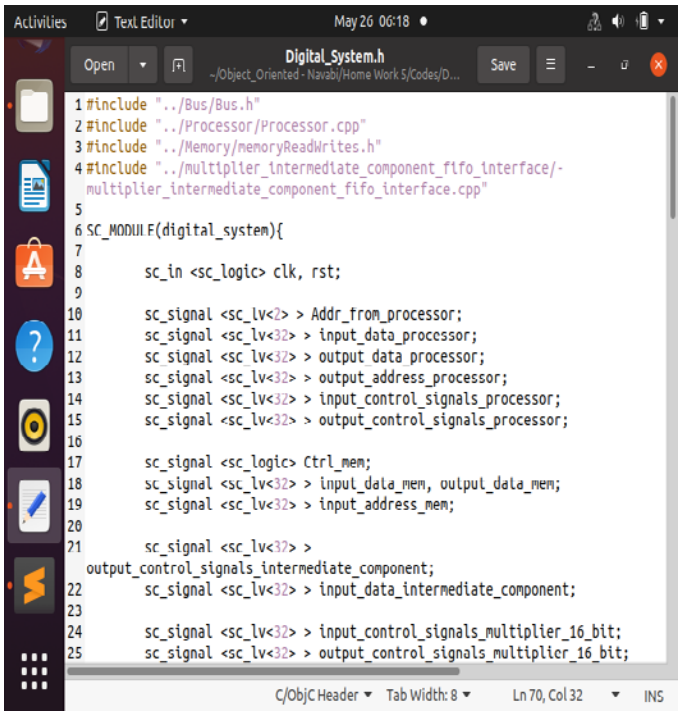

Fig. 35 *Digital System SystemC*

Fig. 37 *Digital System SystemC*

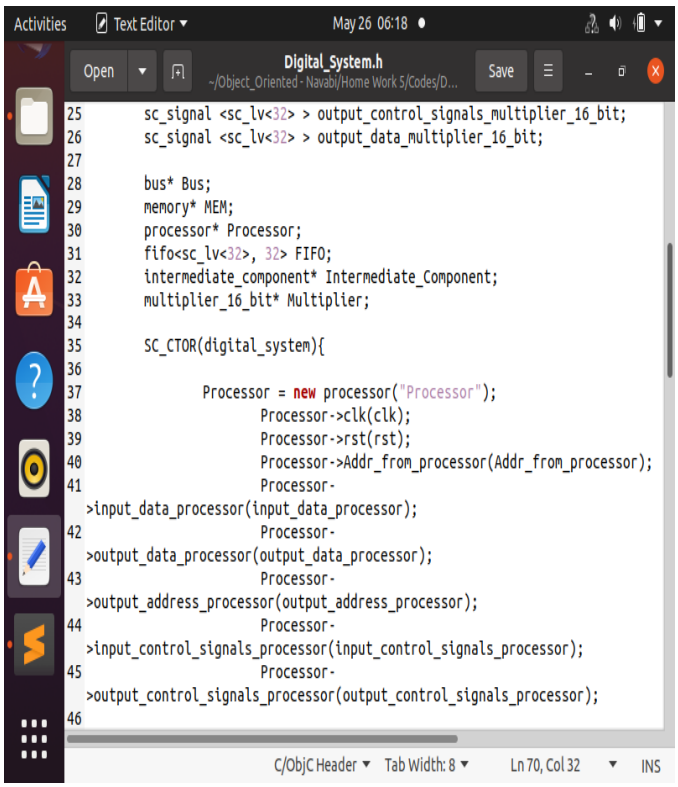

Fig. 36 *Digital System SystemC*

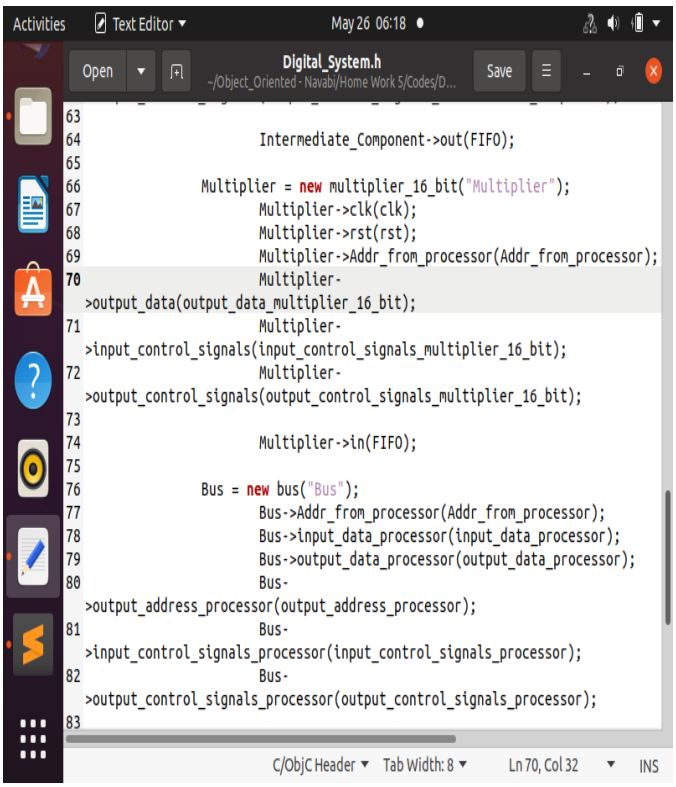

Fig. 38 *Digital System SystemC*

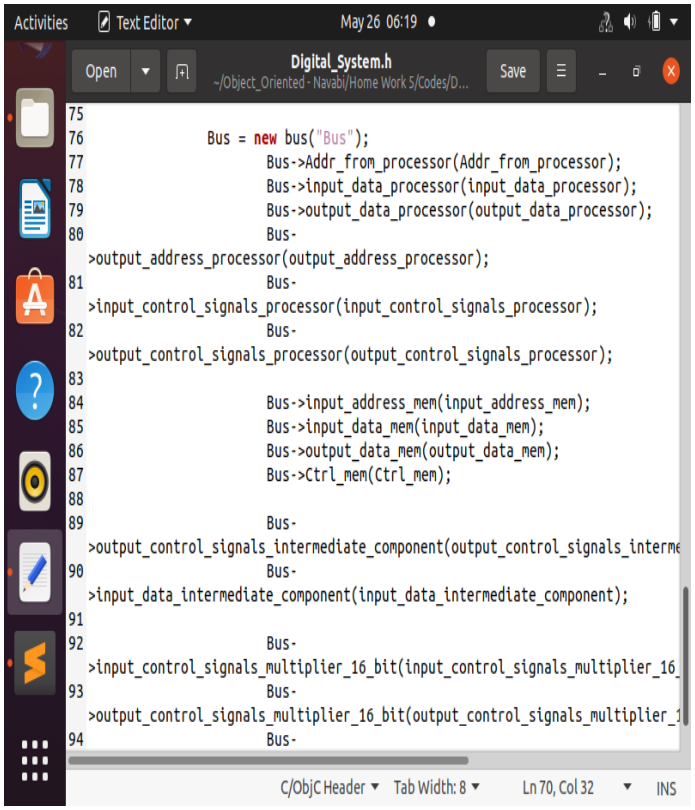

Fig. 39 *Digital System SystemC*

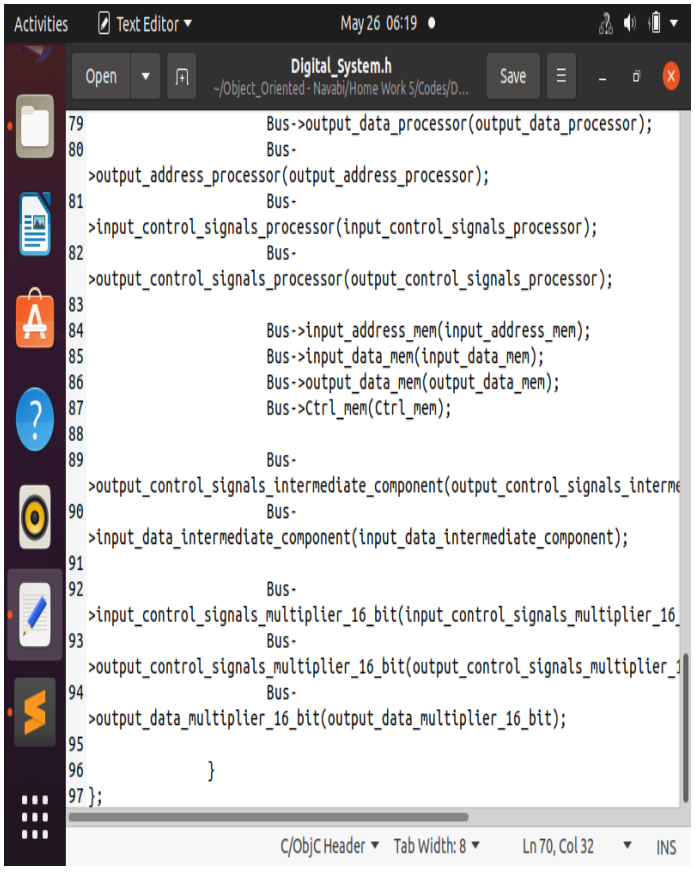

Fig. 40 *Digital System SystemC*

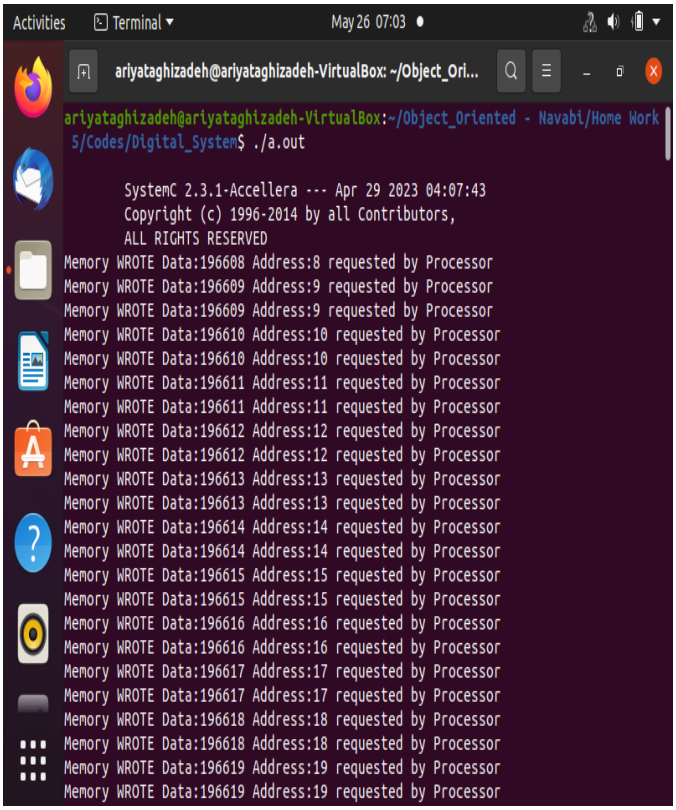

Fig. 41 *Digital System Test Bench results*

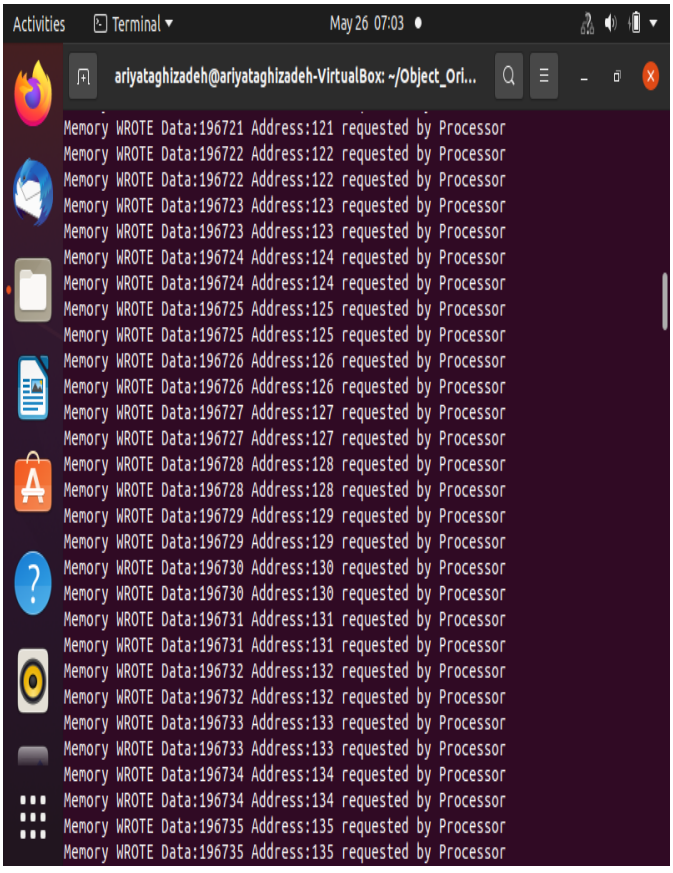

Fig. 42 *Digital System Test Bench results*

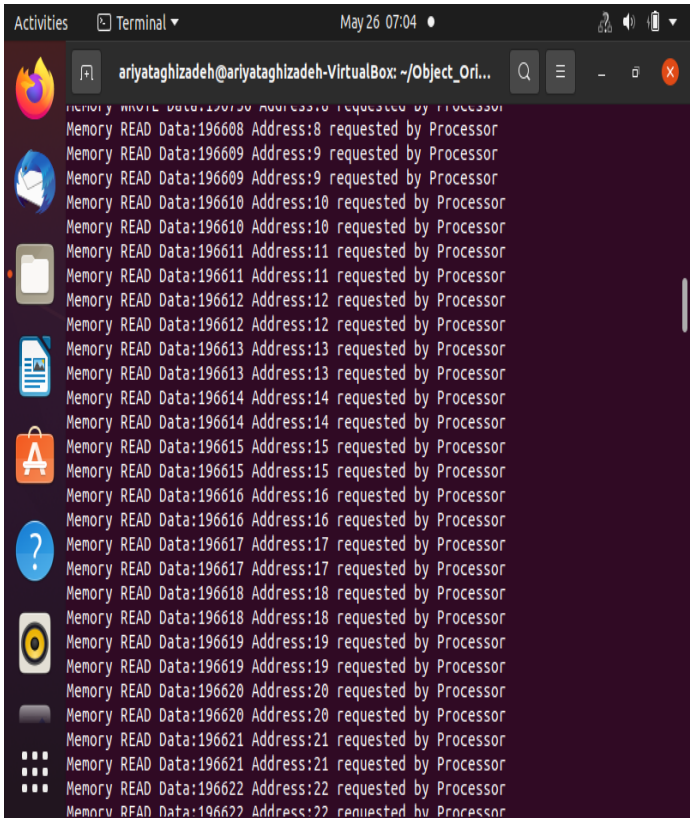

Fig. 43 *Digital System Test Bench results*

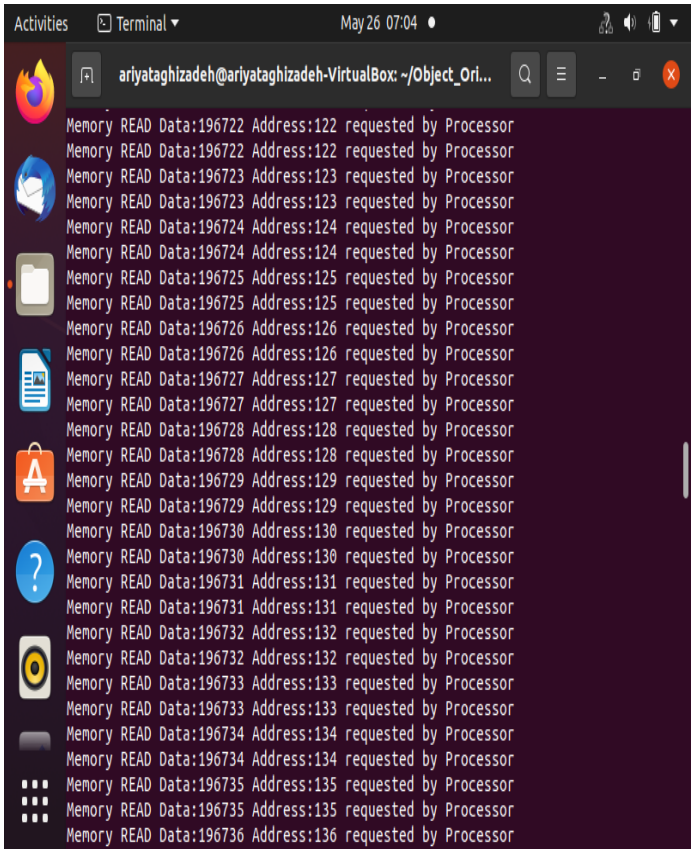

Fig. 44 *Digital System Test Bench results*

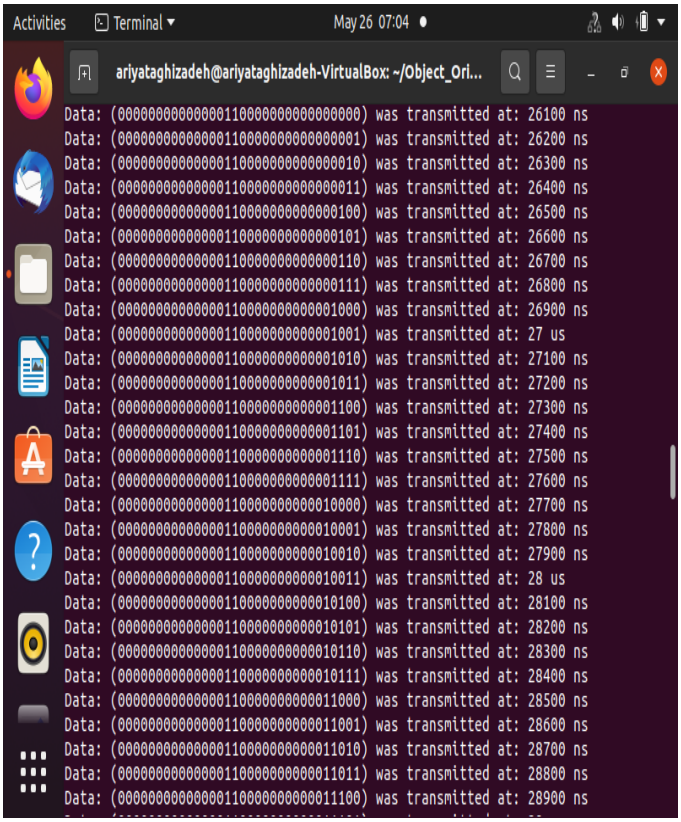

Fig. 45 *Digital System Test Bench results*

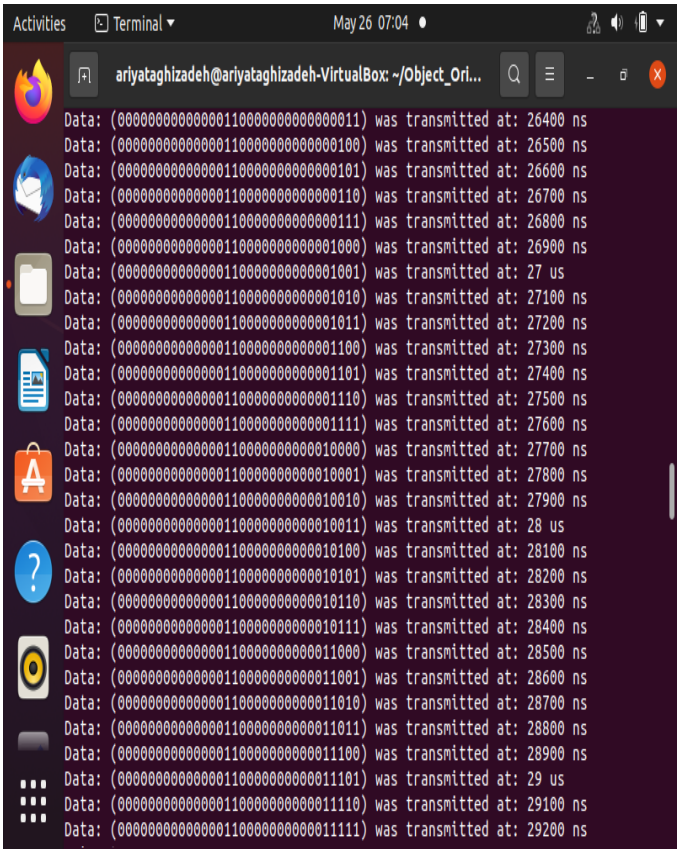

Fig. 46 *Digital System Test Bench results*

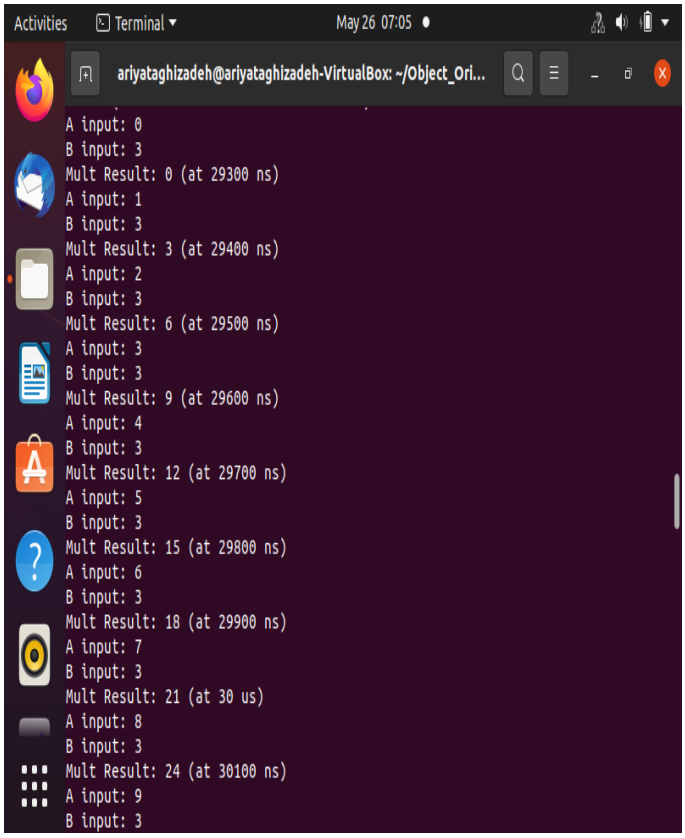

Fig. 47 *Digital System Test Bench results*

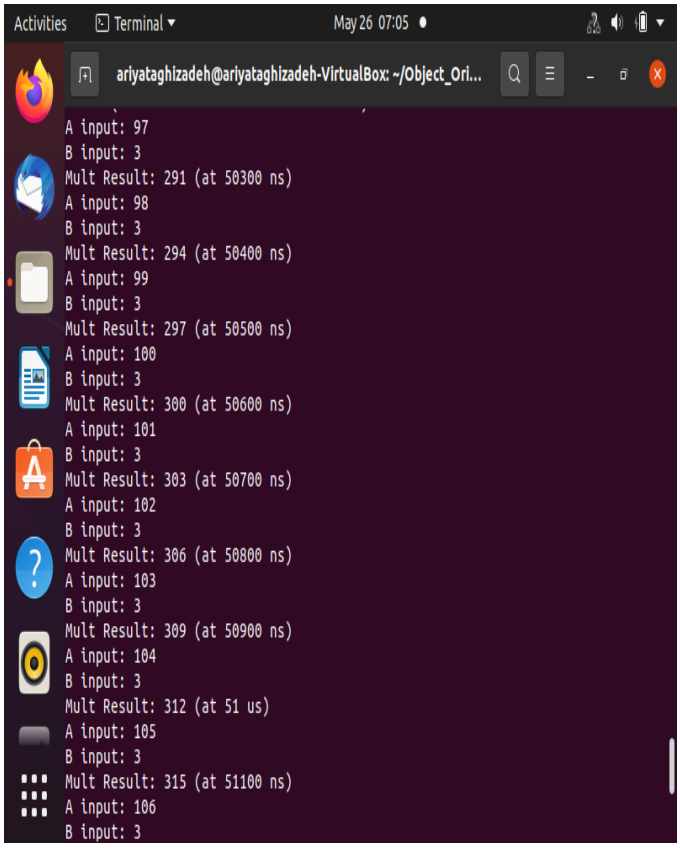

Fig. 48 *Digital System Test Bench results*

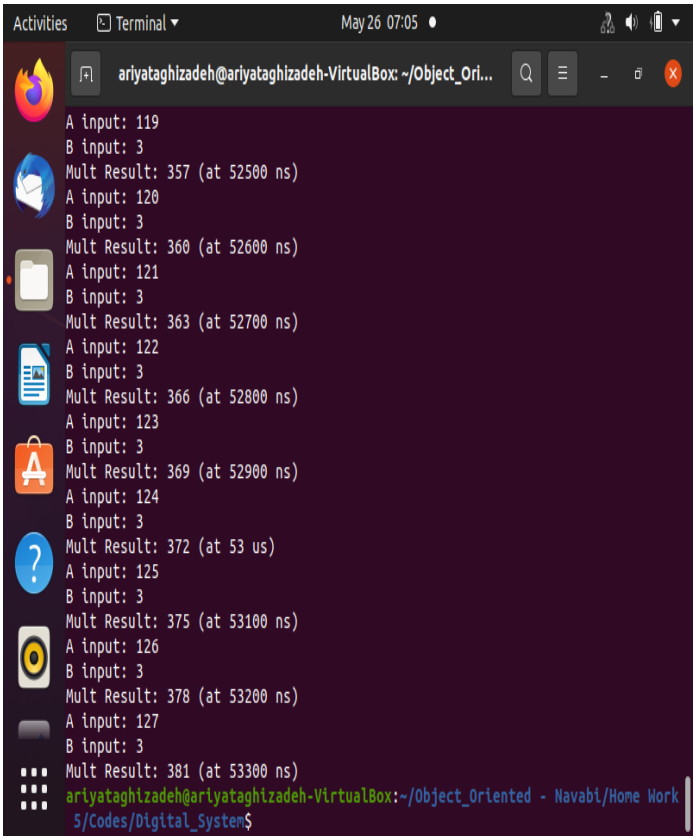

Fig. 49 *Digital System Test Bench results*