

# SystemC Introduction

## Revision History

July 2008: Initial version using FIR Filter (R. Jenkal /P. Franzon)

## 1. Introduction

The aim of this tutorial is to understand the basic language constructs of SystemC. These language fundamentals are going to be used in the description of a simple FIR filter design. This FIR filter consists of a producer that generates data that is needed as an input and an accelerator that computes the FIR result. These two units communicate with each other over nets and also share common input signals for clock, reset and enable. Additionally, we will provide an example of a Transaction Level Model (TLM) of the above system. This is done by introducing the need for a FIFO between the accelerator and the producer to deal with issues of the sub-modules working at different rates. This connectivity is achieved by the use of a `sc_fifo` transaction level channel and its interfaces.

## 2. Learning Objectives

- Understanding of the coding structure for the simple SystemC design
- Understanding the requirements for the simulation of SystemC code in Questa /Modelsim.
- Understand the creation of a simple transaction level model of the design when the system specifications change.

**PLEASE DO NOT CUT AND PASTE COMMANDS FROM THE TUTORIAL. THIS LEADS TO ERRORS. PLEASE TYPE THEM OUT INSTEAD.**

**Please download files associated with this part of the tutorial (Part 1) into a directory of your choice. All the work from this point assumes that you are within this directory.**

1. Header files: `accelerator.h`, `producer.h`, `Top.h`
2. Source Files: `accelerator.cc`, `producer.cc`, `Test_Top.cc`
3. Definition file: `defs.h`
4. Modelsim initialization file: `modelsim.ini`

## 3. Understanding the design example:

The design implements an Nth order FIR filter computation where the **result** =  $x[n] \cdot b_0 + x[n-1] \cdot b_1 + x[n-2] \cdot b_2 + x[n-3] \cdot b_3 + \dots + x[n-N] \cdot b_N$  is performed by an **accelerator**. The data (**x** in the above) is provided to the consumer from a **producer** which also provides the coefficients for the FIR computation (**b<sub>i</sub>** in the above equation). For this example, the order of the filter is fixed at 5.

To implement the above, the design under consideration consists of the following parts:

1. **Producer:** The producer is going to create a controllable number of pieces of data  $x$  that need to be sent to the accelerator for processing. The producer also creates a random array of 6 coefficients (5<sup>th</sup> order filter). These are sent to the accelerator over a dedicated array of ports.
2. **Consumer:** This is going to be timing accurate but not bit accurate representation of the FIR filter which, when enabled, takes in the data input for that cycle and the array of coefficients to compute the result.

The signals of importance in this design are:

- **result:** This is a  $(\text{COEFF\_WIDTH} + \text{DATA\_WIDTH})$  wide bus from consumer to producer. This contains the result of the FIR computation for that cycle.
- **coeff:** This interface is  $6 \times \text{COEFF\_WIDTH}$  (set in defs.h) wide and is represented as an array of ports between the producer and the accelerator. The coefficients  $(b_0 \dots b_5)$  in the above equation are sent over this bus.
- **data:** This represents the data  $x$  that is going to be used for the computation of the result in a given cycle. It is  $\text{DATA\_WIDTH}$  bits wide.

The top level block diagram is shown below:

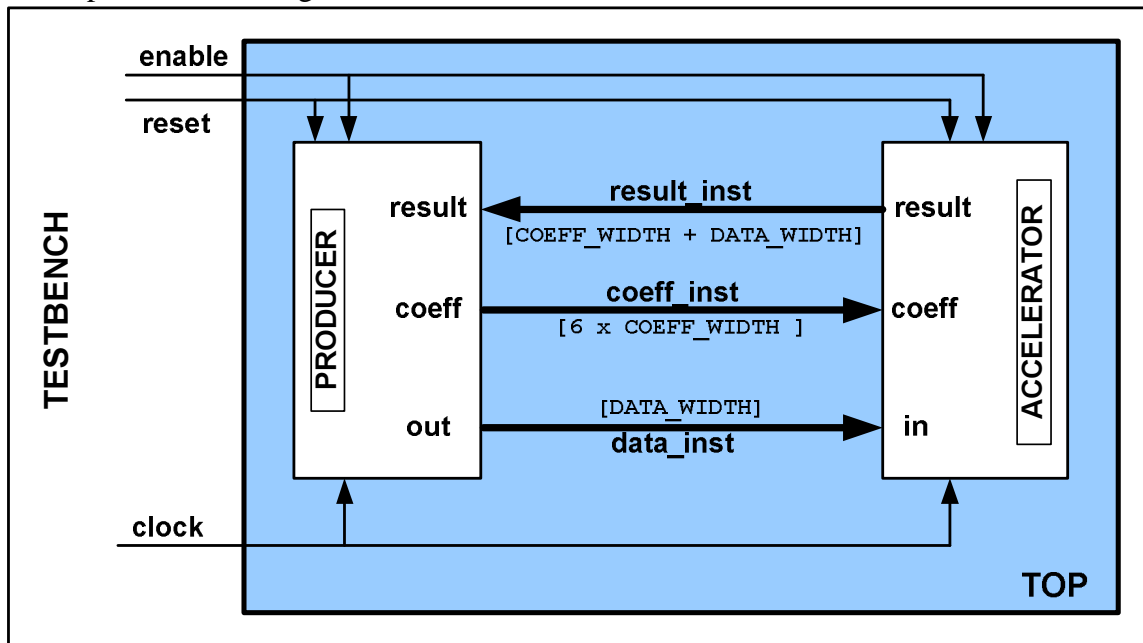


Figure 1: Design organization of SystemC example

The following protocol is followed for setting up the test and computation:

- a. In the constructor of the producer:
  1. Create a user-specified number of data values.
  2. Create 6 coefficients.
- b. Reset the design
- c. As long as enable is 1
  1. At the producer: At each clock edge
    - i. send a new piece of data out to the accelerator
    - ii. read result from the accelerator

2. At the consumer: At each clock edge:
  - iii. compute FIR output result using the incoming FIFO values.

#### 4. Source Code organization

The source code for this example consists of: `accelerator.cc`, `accelerator.h`, `producer.cc`, `producer.h`, `Top.h` and `Test_Top.h`. The `Top.h` is used to integrate the consumer and producer instances. The consumer and producer are described in the `.h` and `.cc` division where the header file is only used to describe the input and output connectivity and the methods to be used to describe the behavior of each functional core (for example `reset_producer()`, `proc_producer()`).

##### 4.1 Default definitions file:

To begin with, we declare all the necessary data types in a definitions header to enable easier iterations on sizes and data types. This practice is great encouraged. The content of the `defs.h` file is shown below:

```
// #####DEFS.H#####
#ifndef DEFS_H
#define DEFS_H

#define COEFF_WIDTH 11
#define DATA_WIDTH 12

typedef sc_int<COEFF_WIDTH >      coeff_type;
typedef sc_int<DATA_WIDTH >      data_type;
typedef sc_int<COEFF_WIDTH + DATA_WIDTH > result_type;

#endif
// #####
```

The definitions here provide widths for the coefficients and data going to the accelerator from the producer and the resulting value from the accelerator.

##### 4.2 Closer look at the producer description:

```
// ##### PRODUCER.H #####
```

```
#ifndef PRODUCER_H
#define PRODUCER_H
```

← *Compilation Directive to prevent double invocation of this header file*

```
#include "systemc.h"
```

← *NEEDED FOR ALL HEADERS: Inclusion of SystemC header*

```
#include "defs.h"
```

← *Inclusion of definitions file specific to this example*

```
class producer : public sc_module {
```

← *Definition of producer as a SystemC module (done as a class)*

`public:`

```
    sc_in  <bool >      clock;
    sc_in  <bool >      reset;
    sc_in  <bool >      enable;
    sc_in  <result_type > result;

    sc_out <coeff_type > coeff[6];
    sc_out <data_type >  out;
```

← *Definition of inputs (sc\_in) and outputs (sc\_out) of module.*

```

void proc_producer();
void reset_producer();
void create_input();
void create_coefficients();

```

Definition of methods specific to this module. It is suggested that the reset behavior be separated from the functional behavior. The functional behavior here is `proc_producer()`

```

sc_signal <int> data_count;

```

Definition of internal signals. These signals show up on the waveform viewer and represent internal wires of the module

The above code segment profiles the first part of the producer header. As explained before, this is just a bare-bones definition of the module under consideration. To begin, the necessary headers are included and the producer is declared as a SystemC module: `class producer : public sc_module {}`. Next, the input and output signal definitions of the module are enumerated as shown. The declaration of the ports is in the format:

```

<port direction>      <data type>      <port name>

```

To capture the behavior of the producer, we can divide up its response into a set of methods. Here the methods of consequence are `reset_producer()` and `proc_producer()` which encode the behavior of the producer when reset and when enabled. We also have other methods for generation of the requisite number of coefficients (here, 6) and data. The rest of the `producer.h` code is shown below:

```

SC_HAS_PROCESS(producer);
producer(sc_module_name name, int input_size){

```

Constructor (called on making an instance of producer class). Requires "name" and "input size" values on instantiation

```

SC_METHOD(proc_producer);
sensitive_pos << clock;

SC_METHOD(reset_producer);
sensitive_pos << clock;

```

Declaration of the reset and behavioral methods as **SC\_METHOD** (SystemC specific). These methods are made sensitive to the positive edge transition of the clock signal

```

input_array = new data_type[input_size];
coeff_array = new coeff_type[6];

input_limit = input_size;

create_input();
create_coefficients();

```

Creation of default data values and allocation of memory for internal variables

} Creation of the input data array and the coefficients in the constructor. Thus, the values are ready on instantiation of this class.

```
private:
```

```
    data_type *    input_array;  
    coeff_type *   coeff_array;  
    int           input_limit;
```

←  
*Local variables that can only be  
accessed within the class.*

```
};
```

```
#endif
```

```
// #####
```

We begin by declaring a user-defined constructor for the producer. To enable the generation of a controlled number of data points, a necessary input to the constructor is the "input\_size" variable. Within the constructor, we make the `proc_producer()` and `reset_producer()` methods sensitive to the positive edge of the clock and declare them to be methods of the type `SC_METHOD()`. This implies that the method will execute once for a positive edge of the clock and can not be stalled during execution using wait statements. To enable stalling or waiting for events to occur, the method would have to be a `SC_THREAD()`.

Allocation of memory is also performed in the constructor. At this point, we are ready to look at providing meaning to the skeletal representation of the producer module.

```
// ##### PRODUCER.CC #####
```

```
#include "producer.h"
```

```
void producer:: create_input() {  
    int i;  
    data_type value;  
    cout << sc_simulation_time() << " [PRODUCER] In create_input" << endl;  
    for (i =0; i <= input_limit; i++) {  
        value = (data_type) rand()%1024;  
        input_array[i] = value;  
    }  
    cout << sc_simulation_time() << " [PRODUCER] create_input finished" << endl;  
}
```

```
void producer:: create_coefficients() {  
    int i;  
    coeff_type value;  
    cout << sc_simulation_time() << "[PRODUCER] In create_coeffs" << endl;  
    for (i =0; i <= 5; i++) {  
        value = (coeff_type) rand()%512;  
        coeff_array[i] = value;  
    }  
    cout <<sc_simulation_time()<< "[PRODUCER]create_coeffs finished"<<endl;  
}
```

The most important part of the description is the system on reset and during normal operation. The reset behavior is shown below:

```
void producer::reset_producer() {
    int i;
    if(1 == reset.read()) {
        cout << sc_simulation_time() << " [PRODUCER] In Reset" << endl;
        data_count.write(0);
        cout << sc_simulation_time() << " [PRODUCER] Reset Finished" << endl;
    }

    for(i=0; i <= 5; i++) {
        coeff[i].write(coeff_array[i]);
    }
}
```

*Declaration that this is the reset\_producer() method in the*

*SystemC function that provides current simulation time*

As seen above, the reset requirements of the producer call for the initialization of all the coefficients to their generated values. Also, the internal signal data\_count is reset to 0. We also note that the coeff[i], reset and data\_count signals are now interfaced with using the write() and read() methods. This is because they are declared as sc\_in / sc\_out / sc\_signal which makes it important to use read() and write() to ensure fidelity of data. A useful internal SystemC method is sc\_simulation\_time() which provides the present simulation time of the execution.

For the normal operation of the producer, reset should be 0 and enable needs to be 1. This leads to the description shown below for proc\_producer(). During normal operation, a piece of data is sent out from input\_array[data\_count] to the port out of the producer. Also, data\_count is incremented to keep tabs on the number of data values sent out. Finally, the result bus from the accelerator is read as result.read().

```
void producer:: proc_producer() {
    if((0 == reset.read()) && (1 == enable.read())) {
        if (input_limit != data_count.read()) {
            out.write(input_array[data_count.read()]);
            data_count.write(data_count.read() + 1);
        }
        else {
            out.write(0);
        }
        cout << sc_simulation_time() << " Resulting value from Consumer " <<
            result.read() << endl;
    }
}
```

*read() and write() methods are methods within classes like sc\_in, sc\_out and sc\_signal that are used to read from and write to the respective signals and ports*

*#####*

At this point, we have captured the intent of the producer. The accelerator follows the same pattern in the encoding of the `accelerator.h` and `accelerator.cc` files. The most important aspect of this block is the computation of the result. This is done in the form shown below:

```
// ##### ACCELERATOR.CC #####
    temp_result = 0;
    for (i= 5; i >= 1; i--) {
        input_array[i] = input_array[i-1];
        temp_result = temp_result + input_array[i] * coeff[i].read();
    }
    input_array[0] = in.read();
    temp_result = temp_result + coeff[0].read() * in.read();
    result.write(temp_result);
// #####
```

The body of the accelerator contains a continuous shifting of the values in `input_array` in keeping with the memory of the FIR filter description. Thus, at any given point, we use the present value at the input `in` of the accelerator and 5 previous values which are multiplied with `coeff[0] .. coeff[5]`.

The next step is the creation of a top level integration file which, keeping with Figure 1,

- instantiates the accelerator and producer modules
- attempts to connect up the ports: `[coeff - coeff]`, `[result - result]` and `[in - out]` of the accelerator and the producer respectively.
- sends the `reset`, `clock` and `enable` signals in as inputs to the top level integration file since they go to both the accelerator and producer.

```
// ##### TOP.H #####
```

```
#ifndef TOP_H
#define TOP_H

#include "systemc.h"
#include "producer.h"
#include "accelerator.h"
using namespace std;
```

*Typical inclusions of header and compiler directives (BE SURE TO INCLUDE THE HEADERS OF THE MODULES TO BE INSTANTIATED)*

```
SC_MODULE(Top) {
public:
```

*Alternate method for declaration SystemC module (here we declare the class TOP)*

```
    producer *    prod_inst;
    accelerator *  acc_inst;
```

*pointers to producer and accelerator classes*

```
    sc_in      <bool>  clock;
    sc_in      <bool>  reset;
    sc_in      <bool>  enable;
```

*clock, reset and enable are still inputs at the level that need to go to the sub-modules*

```
    sc_signal   <result_type>  result_inst;
    sc_signal   <coeff_type>   coeff_inst[6];
    sc_signal   <data_type>    data_inst;
```

*result\_inst, coeff\_inst and data\_inst are the nets that will connect up the ports of the producer with the accelerator*

```
    int i;
```

```

SC_HAS_PROCESS(Top);
// Creating a custom Constructor for the Top level Instance
Top(sc_module_name name, int input_size) : sc_module(name) {

    // Custom Constructor where an instance of top will have to be
    // called with a "name" and "input_size" input. The name provided
    // will be assigned as the name of the module instance and input_size
    // will be passed on to the producer class. This will lead to the
    // creation of a data array of length "input_size" in the producer

    prod_inst = new producer("Prod", input_size);
    // instance of producer with name "Prod"

    prod_inst->out(data_inst);
    // Connecting port out of producer with the
    // signal data_inst

    prod_inst->clock(clock);
    prod_inst->reset(reset);
    prod_inst->enable(enable);
    prod_inst->result(result_inst);
    // Miscellaneous connectivity

    for (i=0; i <=5; i++) {
        prod_inst->coeff[i](coeff_inst[i]);
    }
    // Connection to an
    // array of ports

    acc_inst = new accelerator("Accel");
    // instance of Accelerator with name "Accel"

    acc_inst->in(data_inst);
    acc_inst->clock(clock);
    acc_inst->reset(reset);
    acc_inst->enable(enable);

    for (i=0; i <=5; i++) {
        acc_inst->coeff[i](coeff_inst[i]);
    }
    // Instantiation of accelerator
    // with the name "Accel" and
    // creation of the requisite
    // connectivity to the Top level
    // inputs and internal signals.

    acc_inst->result(result_inst);

}
};
#endif
// #####

```

An important point to note from the above is the fact that all the connectivity was performed within the constructor of the module Top. Moreover, there is no need for a Top.cc file here given that this module does not perform any particular function other than to act as a means of stitching together the accelerator and producer modules.



### 4.3 Top level File coding requirements for testbench:

Of all the files provided, particular attention must be paid to the design of the testbench file (Test\_Top.cc). To enable the use of Questa (or Modelsim for that matter) for simulation, this testbench SystemC code structure must conform to following set of requirements:

1. `SC_MODULE()` is used in place of `sc_main()` to provide the testbench name. For example, in this case we use `SC_MODULE(Test_Top) { }`
2. All the inputs that go into the design (here `Top`) will be sent in through a `SC_THREAD`. Here we use `SC_THREAD(stimulus_test())` in conjunction with the method `stimulus_test()`. As you will note within the `stimulus_test()` method, all the inputs to the DUT are initialized and the times at which they are sent into the design are determined using `wait()` statements.
3. `SC_CTOR` usage is a must for the creation of the constructor for the testbench code.
4. The clock is initialized within the constructor. In this example this is done as:  
`SC_CTOR(Test_Top):clock("CLK", CLKPERIOD, SC_NS, 0.5, 0.0, SC_NS, false)`

5. `SC_MODULE_EXPORT` is used to export the top level instance name of the testbench. In this case it is done as `SC_MODULE_EXPORT(Test_Top);`

// ##### TEST\_TOP.H #####

```
#include "systemc.h"
#include "Top.h"
#define CLKPERIOD 10
```

← Inclusion of `Top.h` and declaration of variable `CLKPERIOD` as 10

```
SC_MODULE(Test_Top) {
public:
```

```
    Top * Top_inst;
```

```
    void stimulus_test();
```

```
    sc_clock clock;
```

← declaration of testbench clock using `sc_clock`

```
    sc_signal<bool> reset;
    sc_signal<bool> enable;
```

```
    SC_CTOR(Test_Top):clock("CLK", CLKPERIOD, SC_NS, 0.5, 0.0, SC_NS, false) {
```

```
        Top_inst = new Top("TopInst", 50);
```

← Testbench constructor with a default clock of 10ns and 50% duty cycle starting at 0

```
        Top_inst->clock(clock);
        Top_inst->reset(reset);
        Top_inst->enable(enable);
```

← Instantiation of `Top` with the name "TopInst" and with requirement for the creation of 50 data values

```
        SC_THREAD(stimulus_test);
```

```
    };
```

← Declaration of `stimulus_test()` as a `SC_THREAD` which implies it does not terminate after one execution and also allows for stalling using `wait()`

```

void Test_Top::stimulus_test() {
    reset = 0;
    enable = 0;
    wait(CLKPERIOD, SC_NS);

    reset = 1;
    wait(CLKPERIOD, SC_NS);

    reset = 0;
    wait(CLKPERIOD, SC_NS);
    enable = 1;

}
SC_MODULE_EXPORT(Test_Top);
// #####

```

*Initialize the inputs going to design under test*

*Send in reset = 0 for 1 clock cycle and then assert enable = 1 till the end of simulation*

*Exporting Test\_Top as a simulation module which leads to it as being registered as a design unit which can be exported to a viewer*

## 5. Establishing the Design Environment for compilation

### 1. One time setup for a given directory used for simulation:

Each time you create a directory for simulations (here Lab1) you would have to do the following

- prompt%> add questasim63 **OR** prompt%> add modelsim
- Copy the modelsim.ini file that comes with this tutorial into the directory. This file sets up the necessary defaults for the Modelsim tool.
- Create the library into which all the design units (SystemC files) will be compiled. Do this by performing  
prompt%> vlib mti\_lib  
(Note that the name “mti\_lib” corresponds to the variable “work” within the modelsim.ini file and is the library to which all the source code would be compiled to create a design single entity).

**Note that in some cases, if the compilation seems to crash for a reason you think is incorrect, it would be advisable to delete the “mti\_lib” directory (Use: \rm -rf mti\_lib OR vdel -all) and re-create it as shown above.**

### 2. One time setup for a set of simulations within a directory:

We assume step 1 has already been followed. Let us assume a directory has been setup up correctly and you come into this directory for a future simulation. You would still need to run the following commands each time you start a set of simulations for a given design within a directory.

- set paths to the Modelsim tool:
- prompt%> add questasim63 **OR** prompt%> add modelsim
- set environment variable MODELSIM to modelsim.ini  
prompt%> setenv MODELSIM modelsim.ini

At this point we have all the relevant variables and directories needed for a successful simulation setup.

## 6. Compilation and Simulation of design

### 6.1. Compilation of source Code:

We state again, that to be able to compile the source code within a directory after logging on for the first time, Step 5.2 should be followed i.e.

```
prompt%> add questasim63 OR prompt%> add modelsim
prompt%> setenv MODELSIM modelsim.ini
```

We assume that Step 5.1 has already been followed and both the `mti_lib` library has been created and the `modelsim.ini` file is already present in the simulation and compilation directory.

For the present example, the compilation of the code is done as (making it generic to the case where the files are in a different directory as the compilation):

```
prompt%> sccom <path to source files>/accelerator.cc
prompt%> sccom <path to source files>/producer.cc
prompt%> sccom <path to source files>/test_Top.cc
```

Here, `sccom` is the compiler for SystemC descriptions (the equivalent of `vlog` for verilog). After the compilation of the files, you will receive the following message from the compiler,

```
Exported modules:
    Test_Top
```

This implies that the top level instance i.e. the testbench module `Test_Top` in this case, has been successfully exported as a design unit. We now need to link the multiple design units and module declarations using :

```
prompt%> sccom -link
```

At any stage of the above compilation or simulation, errors in coding would be displayed as messages with the respective error code.

### 6.2. Simulation of design:

We are now poised to simulate the design example using the `vsim` simulation engine. The invocation of the engine is performed as:

```
prompt%> vsim -novopt Test_Top
```

which, in the generic case it would be:

```
prompt%> vsim -novopt <top level module> &
```

**This invokes the GUI based simulation engine. The `-novopt` command line option is used to direct the simulator that no optimizations should be performed in the invocation of the design unit. This is done to ensure that we have visibility to as much of the design as possible. It you are interested in just having the simulator run**

as fast as possible while not needing to see all the design intricacies, this option can be removed. It must also be noted that, if you are interested in having the simulator run without invoking the GUI, you can do so using the `-c` option (i.e. `vsim -c -novopt <top level module>`) **WITHOUT** the “&”.

[ASIDE] All of the above `sccom` commands (and **ONLY** the `sccom` commands) can be run either on the command prompt or after the invocation of the simulation engine. Thus the prompt would either be the command prompt on the UNIX/Linux terminal or “`VSIM#>`” which is the command prompt of the simulator. Another useful option is the use of `.do` files to prevent having to type all the commands. This will be talked about in Appendix A.

The result of the above invocation is the GUI shown in Figure 2. with the `Test_Top` design unit loaded. Note, that `create_coefficients()` and `create_input()` have already run given that they occur on the instantiation (and hence initialization) of the producer class.

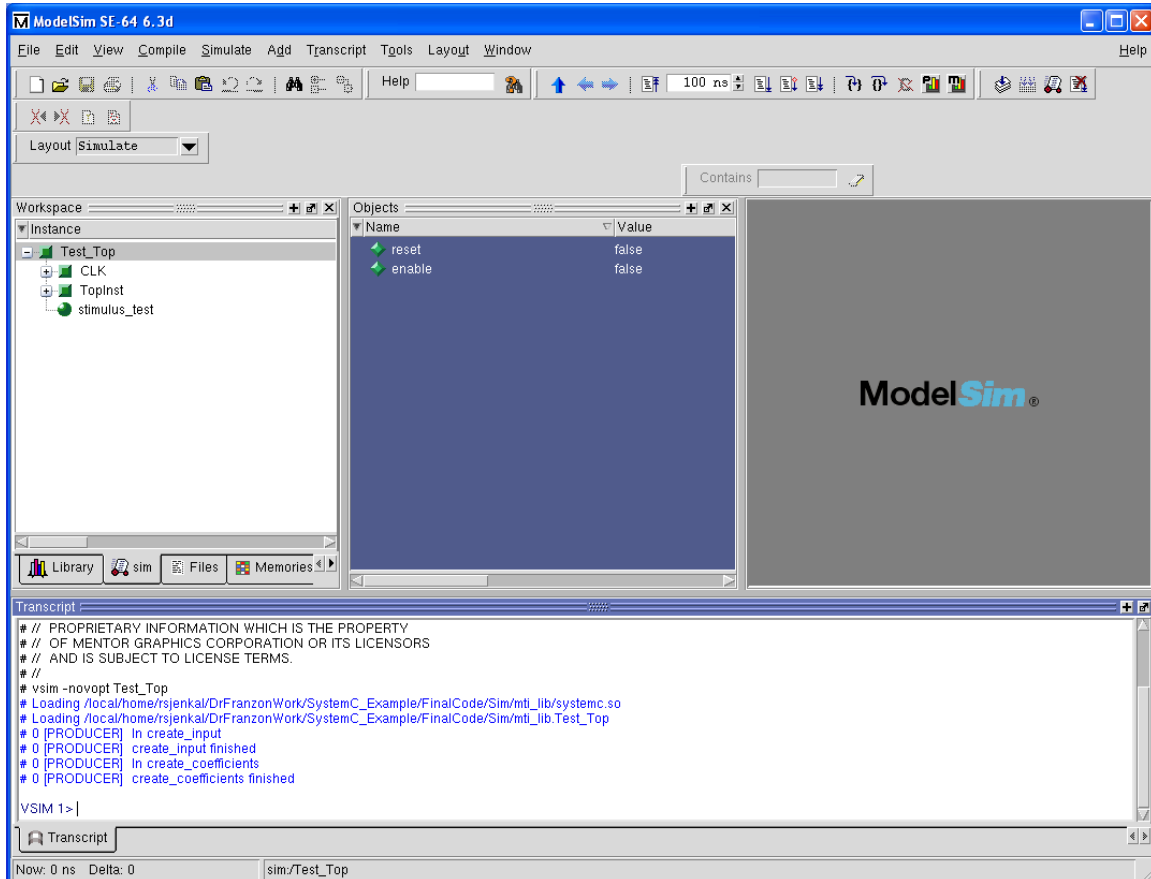


Figure 2: GUI after invocation of design unit

## Design Hierarchy

Verilog allows you to build modules that contain other modules. You might recall that we instantiated the modules `producer` and `consumer` within the module `Top` (look

within `Top.h`), to enable us to test this architecture. The module `Top` is itself instantiated with the name (`sc_module_name`) "`TopInst`" within `Test_Top.cc` (view code for conformation) as

```
Top_inst = new Top("TopInst", 50);
```

You might recall that we had declared the following signals inside `Test_Top`

```
sc_signal      <bool >      reset;
sc_signal      <bool >      enable;
```

Thus within the instance, `Test_Top` we should be able to find the signals declared within that module and a reference to the `TopInst` instance of the module `Top`.

To this end, let us look at the contents of the GUI window. Note the presence of the workspace window and the objects window. The workspace contains all the design units within the `mti_lib` working library. At the top of the workspace window you will see the `Test_Top` design unit. This is the logical top level of the simulation. Given that the `Top` module is instantiated within the `test_fixture` as `TopInst`, you will notice the presence of the instance `TopInst` under `Test_Top`. The hierarchy of the design is captured by virtue of a series of buttons with `+` on them. To go down the hierarchy you would need to click on the `+` to expand it and view the contents of the design unit.

The objects window lists the objects i.e. inputs, outputs, internal nets, parameters etc, corresponding to the design unit that is highlighted in the workspace window. In the above case, the instance `TopInst` has been highlighted by clicking on it. It is interesting to note that when `Test_Top` is highlighted we see `reset`, and `enable` while on highlighting `TopInst` we see `clock`, `enable`, `reset` and a slew of other signals that correspond to the connectivity between the producer and consumer. Moreover, on expanding the `TopInst` tab, you will also notice the instances of the producer and consumer as "`Prod`" and "`Accel`" in keeping with their instance within `Top.h` as:

```
prod_inst = new producer("Prod", input_size);
acc_inst = new accelerator("Accel");
```

We now need to use the GUI to enable us to view the waveforms at the level of the instance `TopInst` to determine correctness of the design operation. One of two things can be done now to view waveforms:

- a. We can either display all the nets in the objects window by right clicking on the objects window and doing: `add to wave` → `Signals in Region` in the list that pops up.
- b. You can highlight all signals that are of relevance to you by keeping the `ctrl` signal depressed and clicking all the signals of interest once. Once this is done, you can do `add to wave` → `Selected Signals` in the list that pops up.

The preview of case a. is shown below:

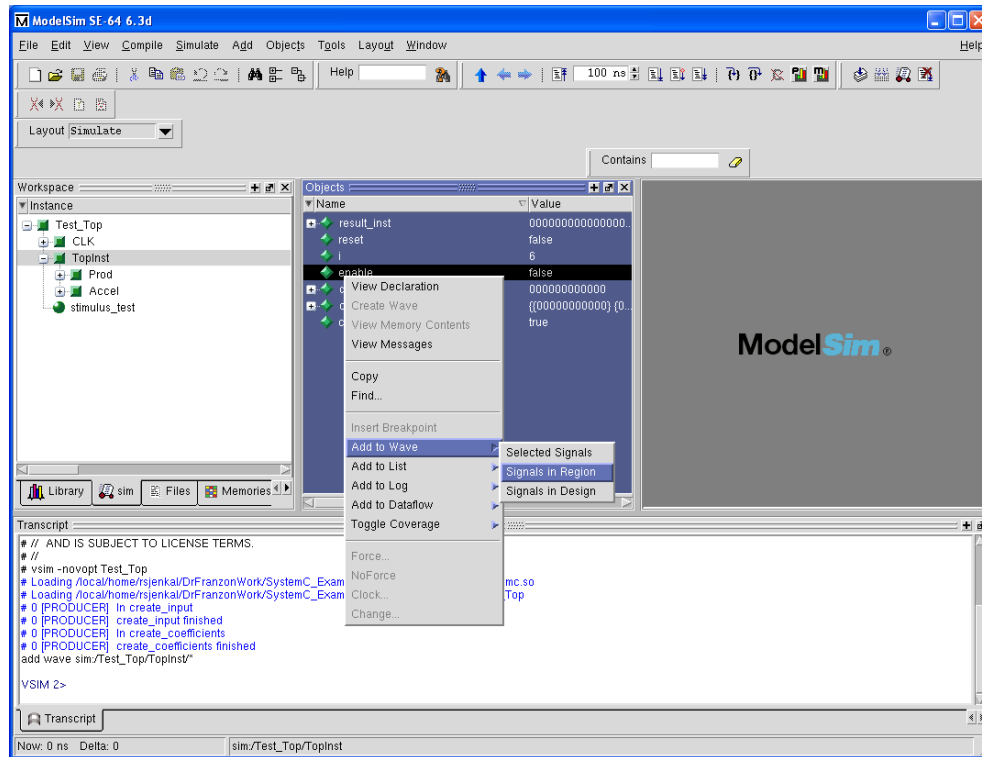


Figure 3: Addition of signals to waveform viewer

The result of the above command is the creation of the waveform window. This is shown below (with the integer i removed). The window would generally result in a window that would be docked with the rest of the GUI. It is easiest to view the wave in a free moving window for the waves. To do this, “undock” the waveform window by hitting the button at the top of the waveform window.

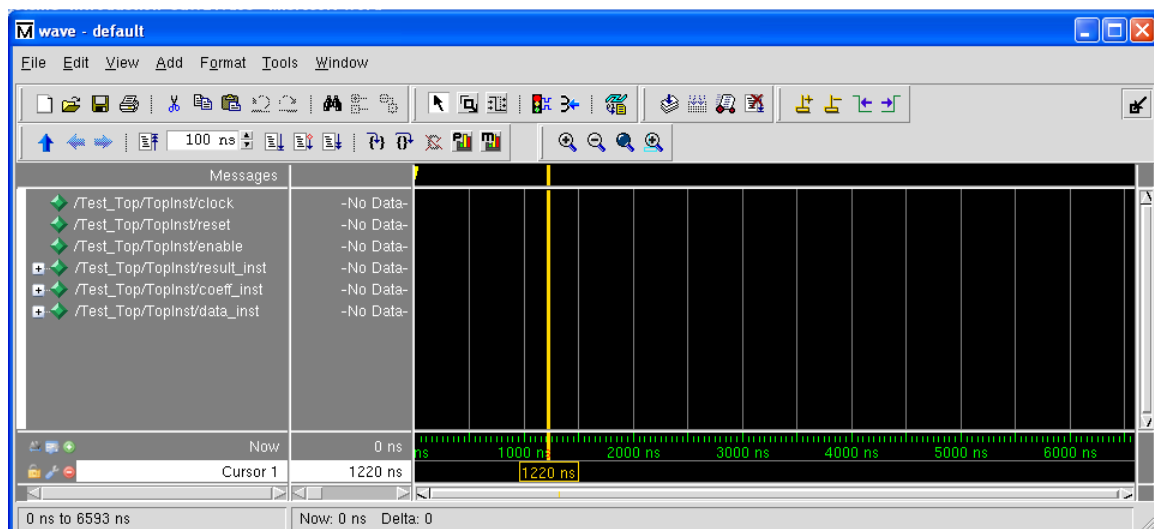


Figure 4: Waveform Window after invocation and undocking

We are now poised to run the simulation for this design. To this end, we would need to do the following at the GUI command prompt (say we are running only for 200ns).

```
VSIM #> run 200ns
```

The result of the above command can be seen in the waveform window where we can note simulation progress. A preview of the waveforms is shown below:

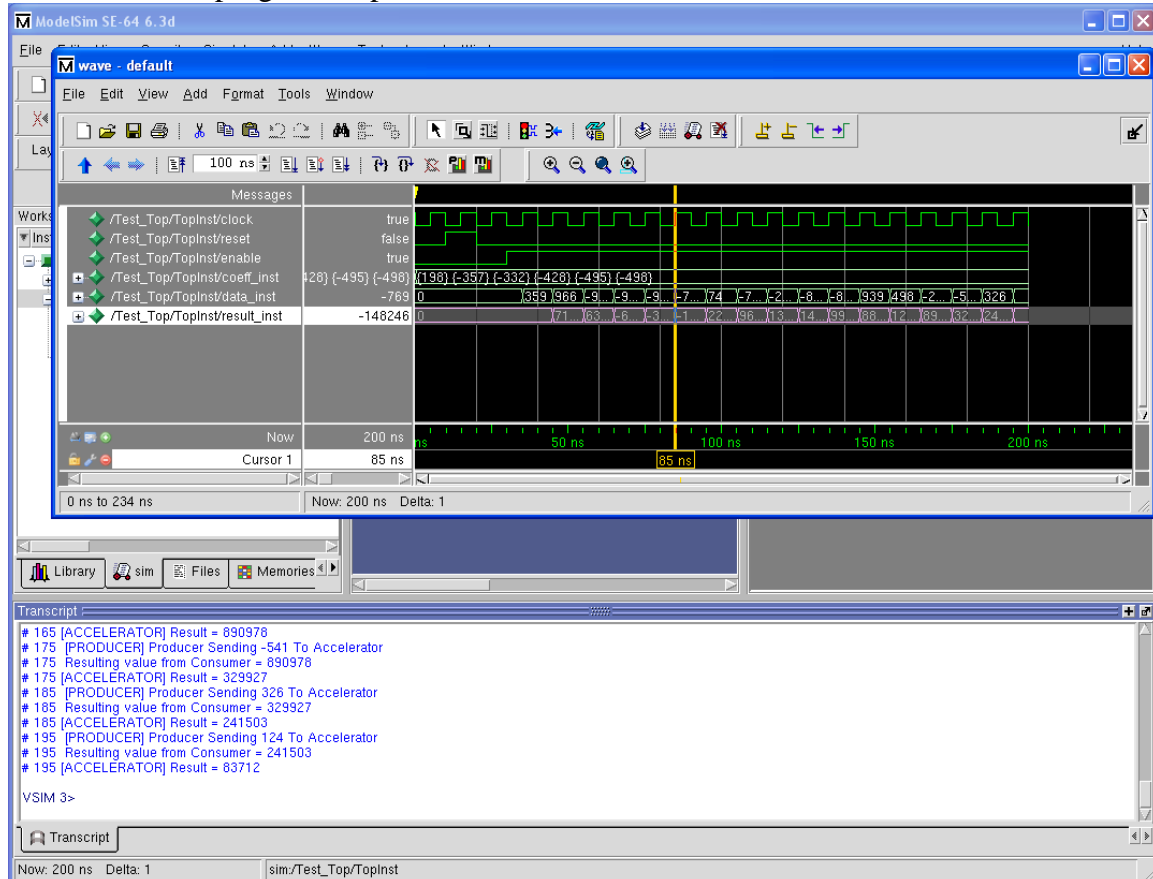



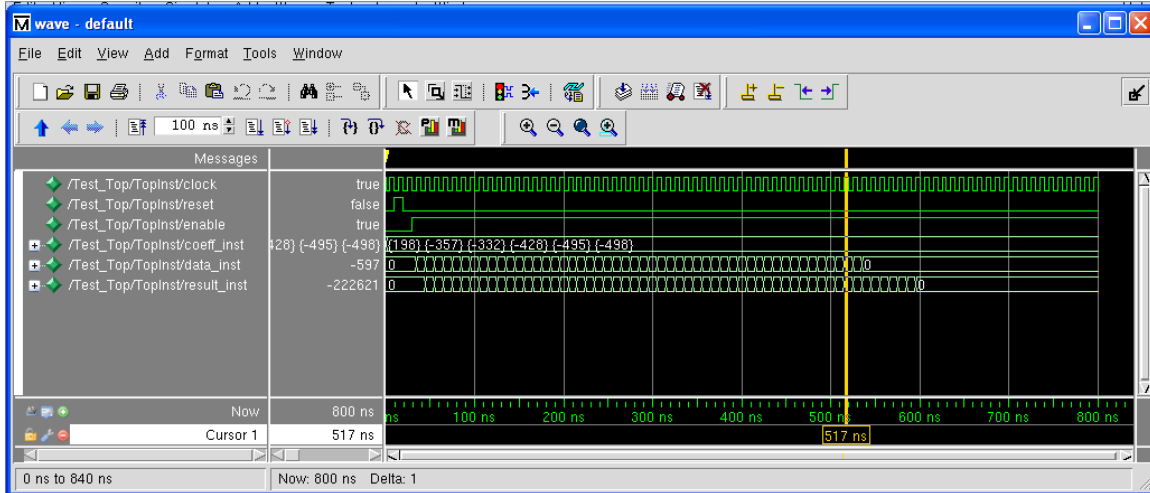
Figure 5: Initial Simulation run for 200ns

To view the waveform in its entirety you would need to fit the waveform in the window by either hitting the 'f' key on the keyboard or by hitting the  key to zoom to fit the waveform in the window. At this point we can ensure that the design works correctly by visual inspection. Something that might prove useful immediately is the tools support for changes in radix. To be able to change the radix of a signal of choice (here the coeff\_inst, data\_inst and result\_inst buses), right click in the signal name on the waveform window and then perform: radix -> decimal (or anything else you might need). The view of the window above assumes a conversion to decimal representation.

**[ASIDE]:** To run this simulation to its completion we would need to do `VSIM#> run -all` where it will take the simulation onwards till a `sc_stop()` is found. When a `sc_stop()` is hit, the tool prompts a "Are you sure you want to finish?" to

which you hit No. It is always advised to run the simulation to a controlled amount of time.

Let us now, run for another 600ns to see how the system reacts. Therefore we do a VSIM#> run 600ns which moves simulation to a total of 800ns. This results in the following waveform result



**Figure 6: System progress after 800ns. [Note data = 0 after 50 data points are sent]**

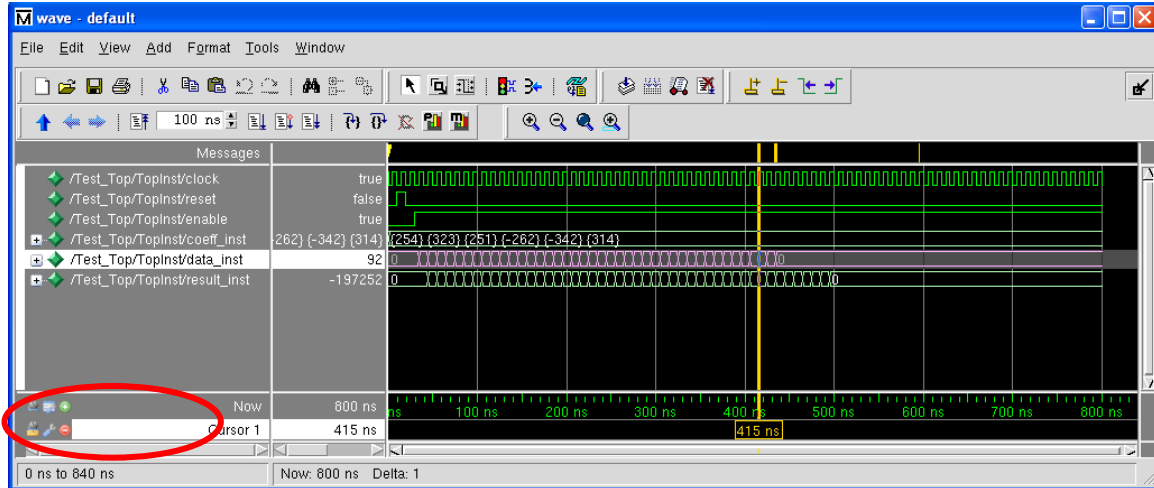
Note that the producer begins sending 0's after all the 50 data values created have been sent in. This manifests itself in the result going to 0 as well after 6 clock cycles.

Let us assume that you want to change a few parameters in your simulation environment, say the number of inputs. This would be done in Test\_Top.cc by changing the line `Top_inst = new Top("TopInst", 50);` where the second entry call corresponds to the number of inputs that are going to be sent into the accelerator. The idea here is to note that we do not need to leave the GUI based environment to do the necessary compilation after the modification of the code. To recompile and simulate, do the following

- Modify `Top_inst = new Top("TopInst", 50);` to `Top_inst = new Top("TopInst", 40);`
- In the vsim GUI type the following : `VSIM#> sccom <path to source file>/Test_Top.cc`. This compiles the edited file within the GUI environment.
- Link the modified design using `VSIM#> sccom -link`
- Restart the simulation that is presently running by doing a `VSIM#> restart -f`  
`VSIM#> run 800ns`

The resulting waveform characteristic is shown below. As can be seen the change in the number of data values created causes reduction in 10 clock cycles in the time taken for the data values to go to 0. This is what we expect.





**Figure 7: Simulation after modification of top level constructor (@800ns)**

Also note the presence of a set of tools (circled in the previous figure), which can be used to create cursors (we have added a cursor in the above figure), lock the cursors, remove them, change the names of the signals from the full path to the short signal name etc.

## 7. Transaction Level Models (TLM)

Transaction level models are used to represent designs in an abstracted manner. What this means is that certain details of representation are abstracted away in the interest of increased simulation speeds and simpler representations of the problems of interest. An example of this would be removal of exact bit level details of a read from memory (enables, control signals, address values etc) and its replacement with just a single function call like `read_memory()`. We will be following a similar example with the representation of a FIFO.

### 7.1 Overview of the system and modifications from base example:

Let us consider for this example that the producer and the accelerator are shared resources and therefore perform their computations only when they are “available”. This availability is modeled using `(rand() % m > n)` to create a  $n$ -in- $m$  probability of availability. Given that these systems are of different rates, it becomes necessary to communicate between them using FIFO’s to ensure lack of data loss. If we were to spend our time modeling the FIFO to exact details, a good deal of design time would be taken away from possible optimizations for the producer and accelerator units. As a way around this, we separate the computation (within the producer and accelerator) and the communication to and from these units. We do this by using a SystemC FIFO channel called `sc_fifo` for communication and interface to this channel using the `sc_fifo_out_if` and `sc_fifo_in_if` channel interfaces. By doing this we gain access to internal methods of the channel (through the interfaces) such as `num_available()`, `nb_read()`, `nb_write()` etc. which can be used to represent required behavior

without all the detail associated with bit accurate representations. Using the FIFOs to communicate the result and the data portions of the communication and assuming the coefficient bus remains unaffected, the system takes the form shown below

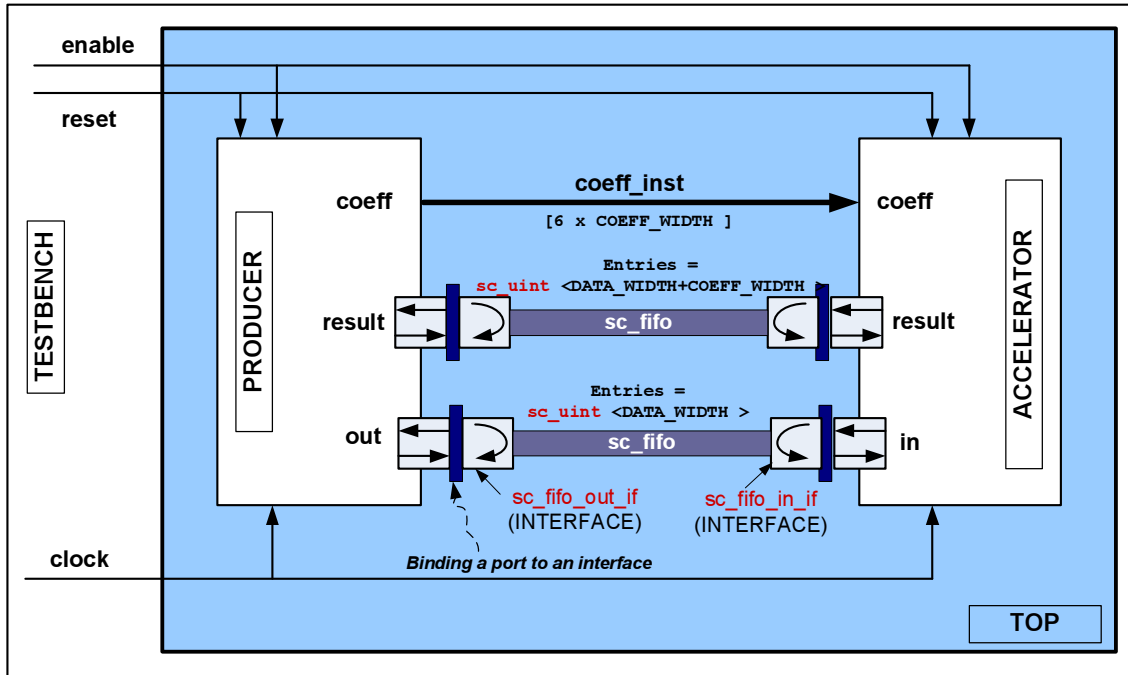


Figure 8: System block diagram with bit accurate signals replaced with `sc_fifo` channels

The design shares the `reset`, `clock`, `enable` and coefficient interface with the previous design example. What changes by virtue of the random “availability” of the producer and the accelerator is the interface for the result and the data busses. The result bus is written to by the accelerator when it is available and will only be read when the producer is ready. The data values are sent to the accelerator over another FIFO.

**At this point, please download files associated with this part of the tutorial (Part 2) into a different directory of your choice. All the work from this point assumes that you are within this directory.**

To explain a description of this nature, we will again use the example of the producer. We will focus specifically on the modifications from the producer in Part 1 of this tutorial.

```
// ##### PRODUCER.H #####
sc_in  <bool>      clock;
sc_in  <bool>      reset;
sc_in  <bool>      enable;
sc_port <sc_fifo_in_if<result_type>> result;
sc_out  <coeff_type> coeff[6];
```

*result is declared as a port of type `sc_port` which inherits its behavior from the interface `sc_fifo_in_if` of the channel `sc_fifo`*

```
sc_port <sc_fifo_out> if<data_type> > out;
```

*out is declared as a port of type sc\_port which inherits its behavior from the interface sc\_fifo\_out\_if of the channel sc\_fifo*

```
// #####
```

From the above, we note that the only major modification is the use of `sc_port` in place of `sc_in` or `sc_out` for the result and out ports of the producer. What we are aiming to achieve here is the connection of these ports to the channel `sc_fifo` using specific interfaces `sc_fifo_in_if` for an input and `sc_fifo_out_if` for an output. The interfaces to a channel determine the methods of the `sc_fifo` to which the ports gain access to. This becomes clearer when we look at the `producer.cc` file. Again, we will be focusing on the part of `producer.cc` that contains a major difference from Part 1.

```
// ##### PRODUCER.H #####
```

```
void producer:: proc_producer() {
    if((0 == reset.read()) && (1 == enable.read())) {
        if(rand()%5 < 2) { Modeling "availability" of resource
            // a 2 in 5 chance of resource for computation being ready
            if ((out->num_free() > 0) && (input_limit !=
                data_count.read())) { Determining number of free locations in FIFO
                                    connected to output port out

                out->nb_write(input_array[data_count.read()]);
                data_count.write(data_count.read() + 1);
            } Writing data to the FIFO connected to port out using the nb_write() method
              which simulates a non-blocking write to the FIFO
            else if(input_limit != data_count.read()){
                out->nb_write(0);
            }
            else {
                cout << sc_simulation_time() << " [PRODUCER] FIFO Full
                    .. need to try later" << endl;
            }
            if (result->num_available() > 0) { Determining number of
                                                available pieces of data in the
                                                FIFO connected to input port
                result->nb_read(temp_result); in
            }
        } Reading data from the FIFO connected to input port in using the
          nb_read() method which simulates a non-blocking read from the FIFO
    }
}
```

The accelerator follows similar trends in terms of modifications from Part 1 and we leave to the reader to note the difference between the description in Parts 1 and 2.

The next major modification takes place in the `Top.h` where we need to specify the connectivity between the accelerator and the producer and provide sizes for the FIFOs

connecting the result ports and the data ports of the two modules. To this end, some of the important modifications in Top.h are:

```
// ##### TOP.H #####
producer *      prod_inst;
accelerator *    acc_inst;

sc_fifo<data_type >    data_fifo_inst;
sc_fifo<result_type >  result_fifo_inst;
```

*// declaration sc\_fifo channels for the data and result busses*

```
SC_HAS_PROCESS(Top);
Top(sc_module_name name, int result_fifo_size,
    int data_fifo_size, int input_size) :
    sc_module(name), result_fifo_inst(result_fifo_size),
    data_fifo_inst(data_fifo_size) {}
```

*Making FIFO instances using sizing values*

*Constructor where the inputs include "name", the number of data points to be generated and the sizes for the data and result FIFOs. The FIFO sizes are then used to make instances of the result and data FIFOs.*

```
prod_inst = new producer("Prod", input_size);
// instance of producer with name "Prod"

prod_inst->out(data_fifo_inst);

prod_inst->clock(clock);
prod_inst->reset(reset);
prod_inst->enable(enable);

prod_inst->result(result_fifo_inst);

for (i=0; i <=5; i++) {
    prod_inst->coeff[i](coeff_inst[i]);
}
```

*Connecting the data and result fifo channels to the relevant ports of the producer. REMEMBER: the ports have the relevant interfaces sc\_fifo\_in\_if and sc\_fifo\_out\_if to the sc\_fifo channel*

```
acc_inst = new accelerator("Accel");
// instance of Accelerator with name "Accel"

acc_inst->in(data_fifo_inst);

acc_inst->clock(clock);
acc_inst->reset(reset);
acc_inst->enable(enable);

for (i=0; i <=5; i++) {
    acc_inst->coeff[i](coeff_inst[i]);
}

acc_inst->result(result_fifo_inst);
```

*Connecting the data and result fifo channels to the relevant ports of the accelerator.*

```
// #####
```

The next modification occurs in the Test\_top.cc where we have to modify the instance of the Top module with the inclusion of data and result FIFO sizes. Thus the only modification happens to be:

```
// ##### TEST_TOP.CC #####

Top_inst = new Top("TopInst", 10, 8, 50);
    ↑
    Create an instance of Top with a) the name TopInst b) result FIFO size of 10 c) data FIFO
    size of 8 and d) 50 input samples of data to be evaluated.

// #####
```

## APPENDIX A: ADDITIONAL TOOL INFORMATION

1. **Working on the command prompt:** It is useful to note that, if the GUI is to be avoided (debug using only text commands/generation of files/checking for compilation correctness.), the simulation can be invoked in the non-GUI mode by doing a `vsim -c`. This causes a Modelsim prompt to appear where you compile (`vlog`), invoke the simulation (`vsim`) and run the entire simulation (`run`). This is sometimes a good alternative when X-windows proves troublesome or you do not have a connection strong enough to support a GUI based operation.
2. **Compiling selectively:** If you make a modification in just a few .cc files, then you can compile just those files with a `sccom` (instead of the entire set of files you are working with). Be sure to do a "`sccom -link`" after the modified file and just do a `VSIM#> restart -f` to restart the simulation. There is no need to quit the GUI.
3. **Using .do files:** An extremely useful feature in Modelsim is the presence of do files. These are the equivalent of shell scripts for Modelsim. All the necessary compilation commands, simulation commands, waveforms etc can be stored within a .do file. An example .do file is provided on the EDA page. This allows you to avoid typing in all the commands manually. Instead, you can call this file within the vsim environment ( GUI / no GUI) by doing a `VSIM #> do <filename>.do`. Of particular importance in working with do files is working with waveforms.
  - a. **Saving Waveforms:** Once you have set up all the relevant waves on the waveform window, it is feasible in Modelsim to store the format of these waves in a .do file and restore the same set of waves the next time you come back to the simulator. This is done by saving all the waveforms by doing a `File→Save →` and saving the format as a .do file. The next time you invoke the simulator, **MAKE SURE THAT THE DESIGN HAS BEEN LOADED** using a `vsim <modulename>` for eg, `vsim -novopt test_fixture`, after which you can open the waveform window and do a `File → Load → <filename>.do` to get the same set of waveforms as before.