Participants:

Jacob Munkholm, 20140479

Jens Jakob Mikkelsen, 201506215

Yang Hanshou, 201902791

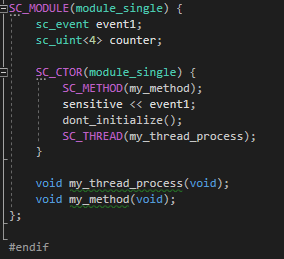
**Embedded Real Time Systems – Assignment 1**

**System level modeling using SystemC**

# Modules, threads, methods and events

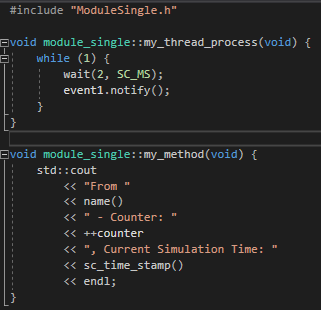
## 3.1:

The module ’moduleSingle’ is created:

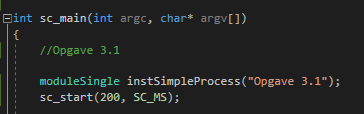


The thread to continuously notify the method, and the method to count, is initialized in the constructor. Static sensitivity is used, my\_method is called everytime the event is raised.

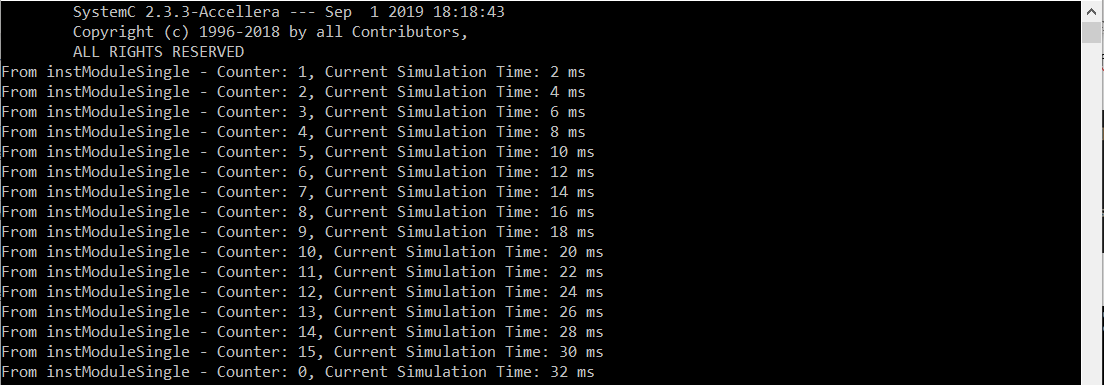
The functions looks like this:



’Trigger thread’ is an unending loop, calling counter\_event.notify() every other ms. In the main function, the simulation is limited to 200 ms:



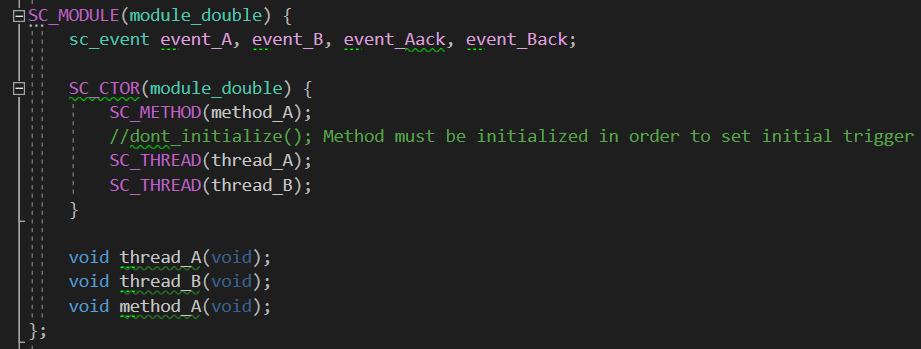
The following is the result:



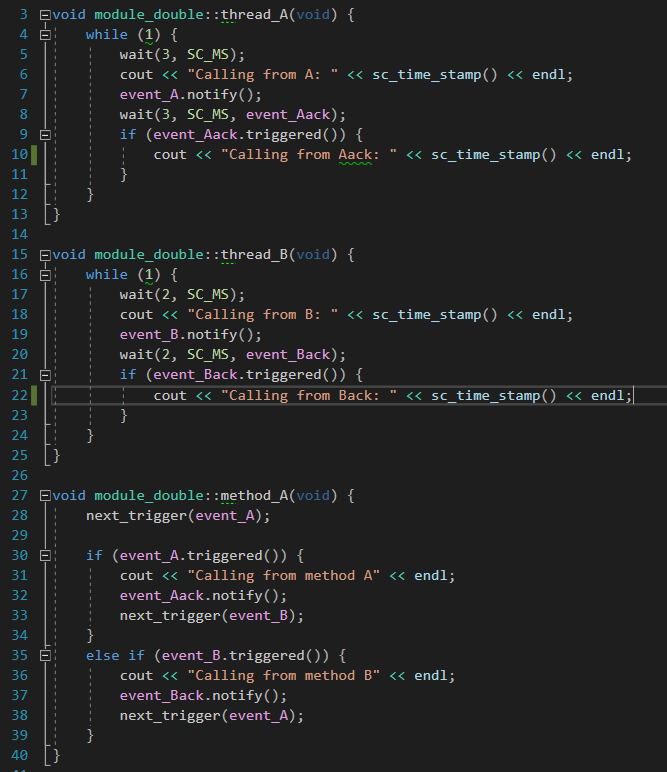
When sc\_uint<4> counter overflows, the program will count from zero again.

## 3.2.

The different threads and events and the method are created. It’s worth noting, that since the trigger function of method A is dynamically sensitive, it needs to be initialized in order to call next\_trigger() the first time.



The threads and the method look like this:

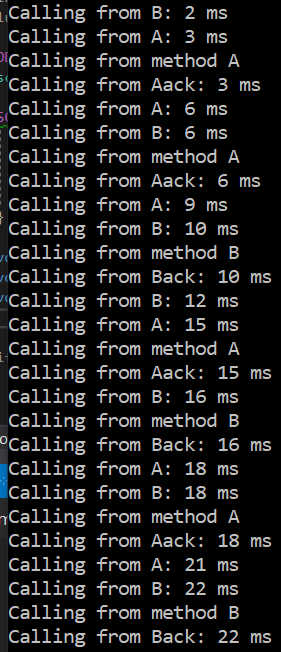


In method\_A(), the method is initialized to be triggered by event\_A, when the program is running. After that, when the method has been triggered it shifts between setting event\_B and event\_A as the next trigger. When either of the these has triggered the method, their corresponding acknowledgement event (event\_Aack and event\_Back) is notified, and the other event is set as the next trigger.

There is one problem with the current solution. The method determines which event triggered the method, by calling triggered() on the event. If both events are triggered at the same time, the method will assume that the triggering event was event\_A, as this is the first to be checked. If the program should work so that methodA knows exactly which event is called, signals would need to be used.

Both thread\_A and thread\_B first notifies their respective event, whereafter they wait for the respective acknowledgement event. Thread\_A notifies its event every third second and waits on the acknowledgement for three seconds. The same goes for thread\_B but the timespan is to two seconds. This it done in the wait function by specifying event\_Aack and event\_Back as parameters. If the acknowledgement event is not notified, the thread will timeout and restart. This is checked by calling triggered() on the event.

The result looks like this:

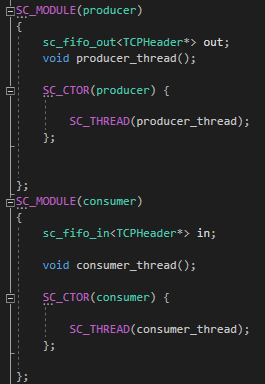


It can be seen, that even though thread\_B is the first to trigger an event, event\_A is the first to trigger the method. The aforementioned problem of determining the triggering event, can be seen at 6 ms, where event\_B triggered the method, but the method assumed it was event\_A, so that only event\_Aack was notified. After that, the method steadily shifts between notifying event\_Aack and event\_Back.

# Channels, signals, hierarchy, communication

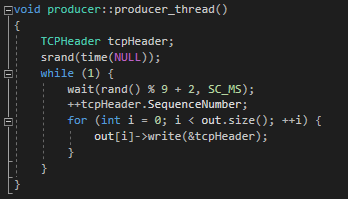
## 3.3

A producer and consumer-thread is created:

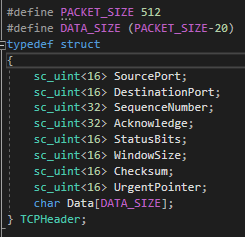


The main object to be used is the sc\_fifo\_in/out, through which data is transferred.

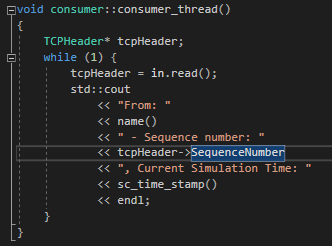
The producer thread is written like this:



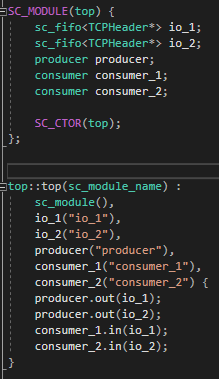
The thread will wait a random amount of time between 2 and 10 ms, increment the sequence-number (the whole TCP segment structure is not used) and write the sequence to the queue. The TCP-struct is defined, but only the sequenceNumber is used to demonstrate the concept:



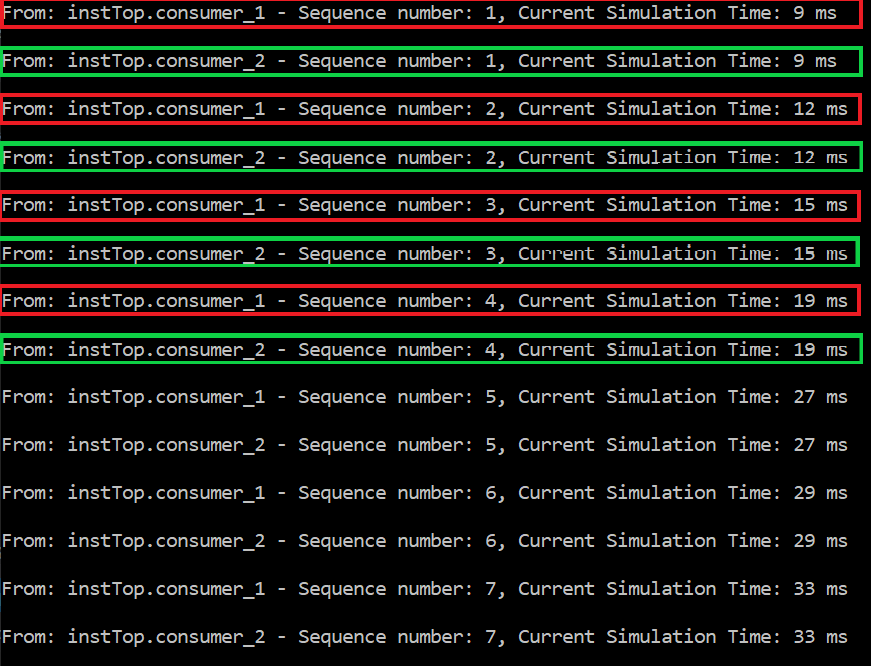
The consumer-thread read the queues continuously and then writes out the sequenceNumber:



In the top-module, the producer, consumer and fifos are created. More consumers and producers could be added like this, if need be.

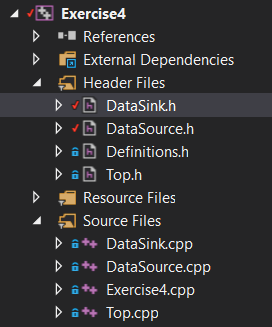


In the console output, it can be seen that each consumer receives the same package (marked by the sequence number) at the same time. Consumer 1 is marked in red, consumer 2 is marked in green.



## 3.4

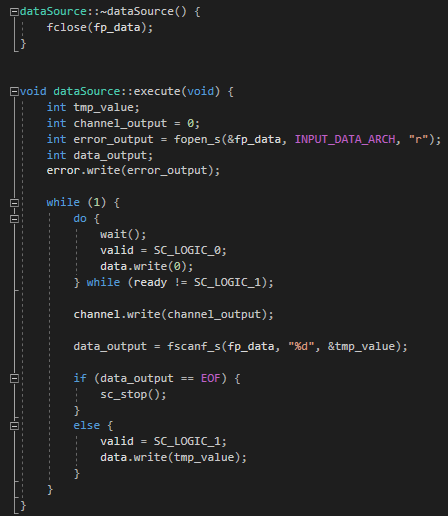
The program consists of the following files:



As usually, the top module initializes all the signals mm. It also creates the clock and creates the traces on the signals. SC\_METHOD(clock\_signal) uses static sensitivity to be called every time the clock raised.

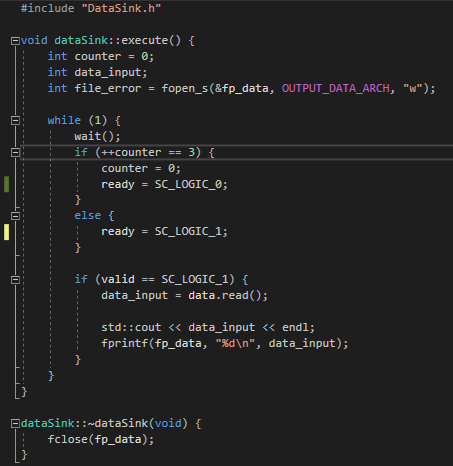


The dataSource looks like this:



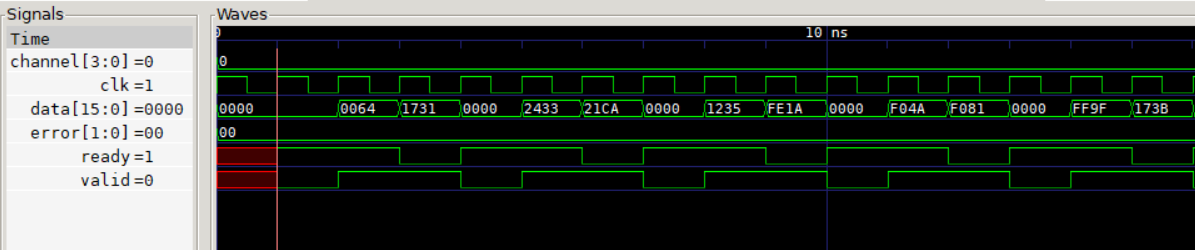
The loop runs until the end of the file fp\_data points to has been read. The loop follows the protocol, so it only sends data when the valid signal is high, and 1 with a 1 clock cycle delay from when the ready signal is high. If not, it only sends zero’s to simulate not receiving data.

The sink then looks like this:



It waits one clock cycle, and pulls the ready signal high for the source to react on. Then it itself expects the valid signal, after which it will read the data from the queue. On the third cycle, it sets the ready-signal low, which is the cycle in which it will receive zero’s from the source.

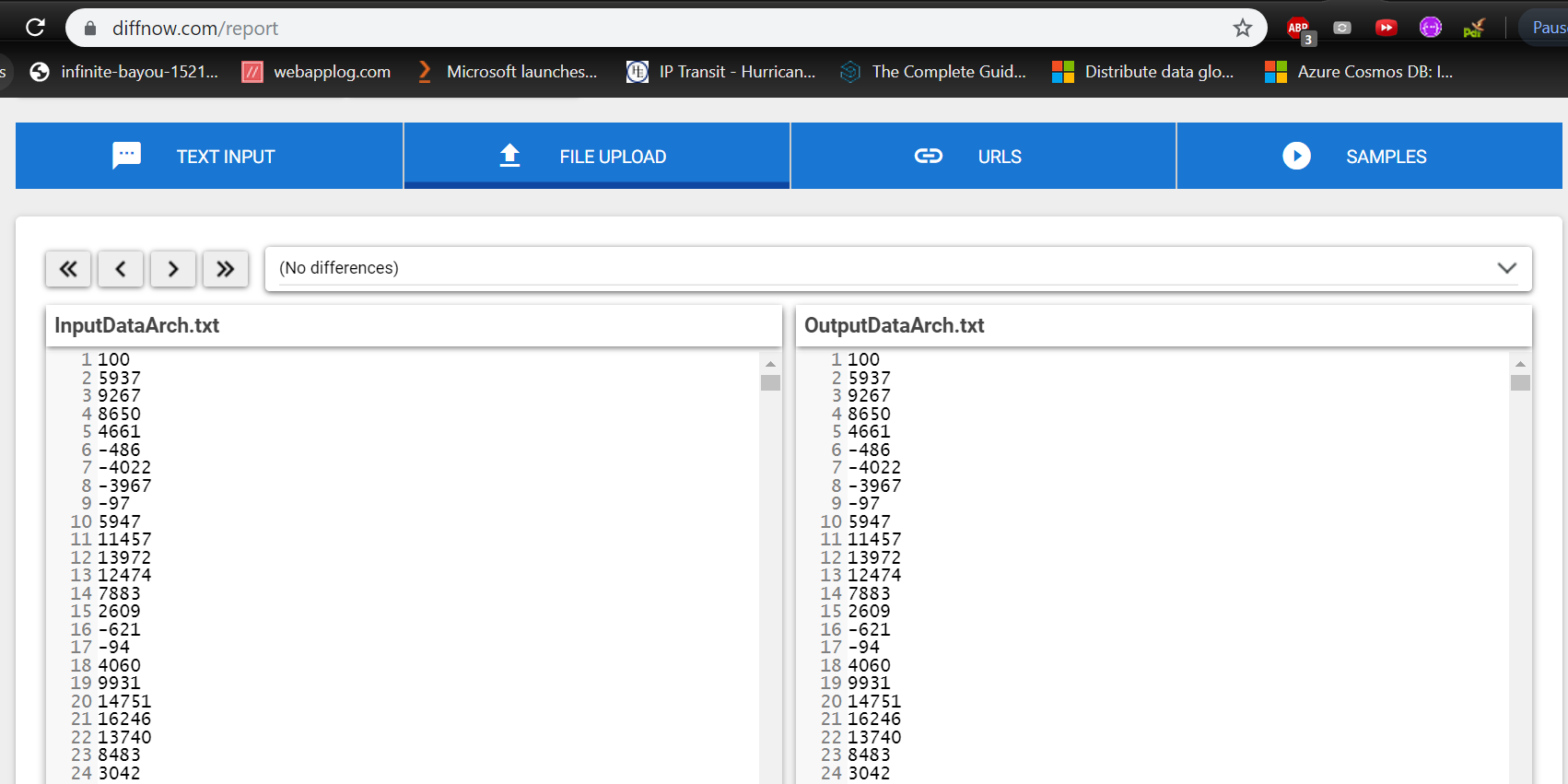
The tracefile in the end, looks like this:



It is seen that then one data byte is send out every clock-cycle when the valid signal is high, delayed by one clock cycle from the ready signal.

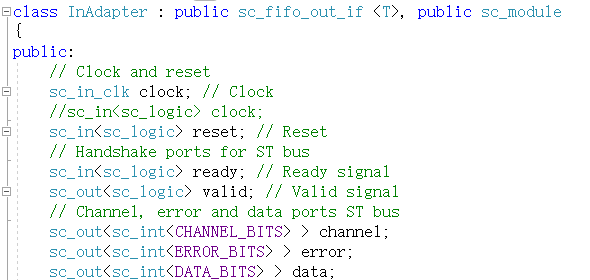
The error byte and data signals are made so they can be chosen manually, but does not affect the code in the source or sink for now.

The data is collected from a text file “InputDataArch.txt” in DataSource and saved in another text file “OutputDataArch.txt” in DataSink. The content has been determined to be identical using an online comparison tool “diffnow.com” to compare the two text files and look for differences:

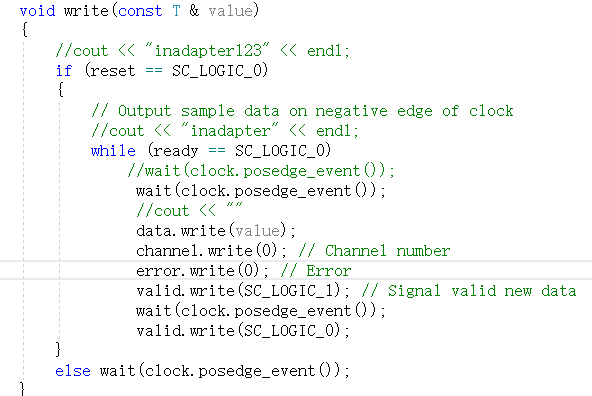


As the files are identical, all data has been transferred from the source to the sink successfully.

## 3.5



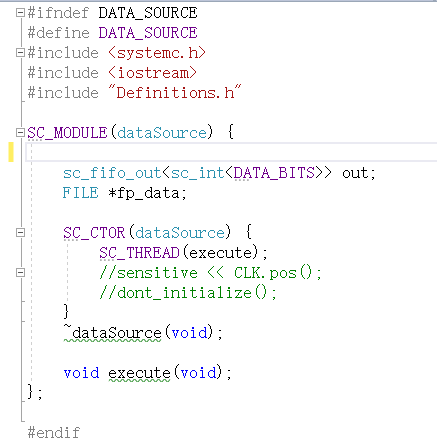
Definition of the InAdapter. The job of the InAdapter: Receiving data from Master, receiving ready signal from Slave, transfer the data to the Slave, output the signal error, channel to the Salve, and also receive the command reset and clock.

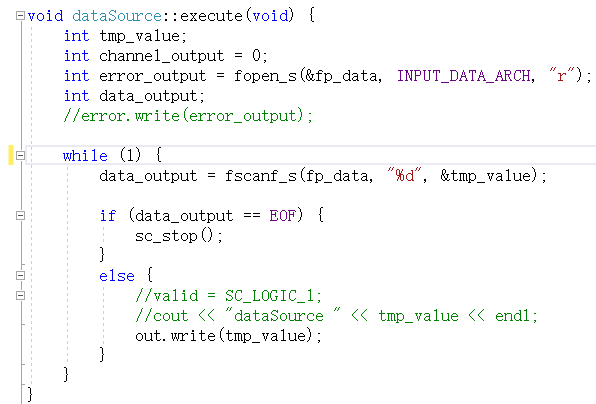


The write function of the InAdapter:

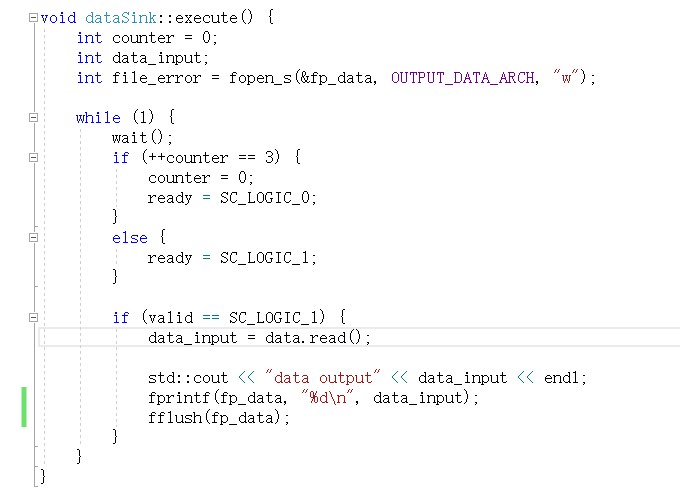
If it receives the command “reset”, then it would be idle and wait for next clock;

If it does not receive the “reset”, it waits for the “slave” to be ready, and when the slave is ready, sends the “data”, “channel” and “valid” to the Slave.

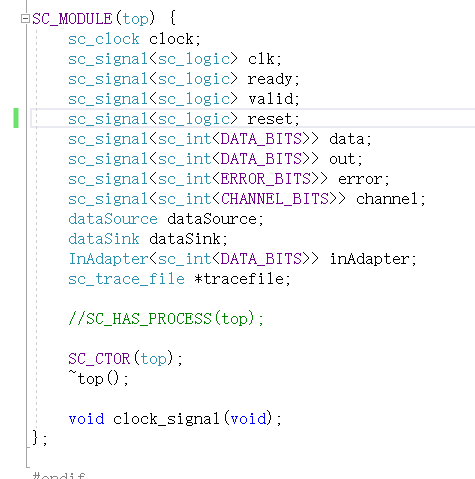


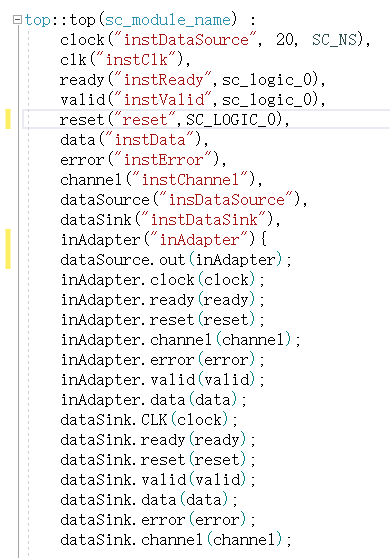


The definition and implement of “master”: it only does the job of reading data from input file and sending data to the InAdapter.



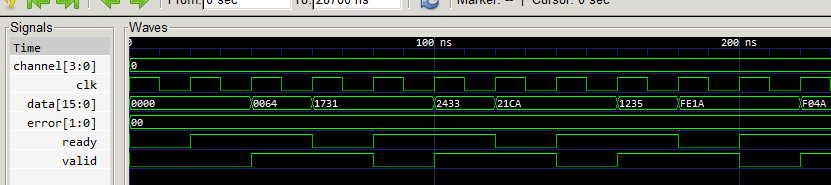
The “Slave”, which is “dataSink” is the same as the on in ex 3.4, but with a new line fflush(fp\_data); just clean up the space cache for the new data waiting to be written to the output file.





The definition and implement of the Top: Since InAdapter inherits from the class sc\_fifo\_out\_if (as stated in the definition) and the “master” has a sc\_fifo\_out, they can be connected to each other directly (dataSource.out(inAdapter));

And then, InAdapter does the same job as the “master” did in the ex3.4: connecting with the “slave”.



Simulation result:

As the code stated in InAdapter, when the InAdapter receives the “ready” signal, it would sent signal “valid” with value 1 after another clock. As stated in the simulation, the “valid” is one clock lagged behind the “ready”.

The InAdapter has to wait for the “Slave” to be ready to transfer data to it, so in the simulation picture, some data`s transfer period is long while the others` is short, because it have to wait the “slave” to be ready.

The “channel” and the “error”, as stated in the code in InAdapter, always has the same value.