2019

ERTS Assignment 1

SYSTEM LEVEL MODELING USING SYSTEMC

JONAS KJÆR RASK

MADS KRABSEN

Description

In this assignment, you will learn about modeling with SystemC at the system level. The purpose of the assignment is to learn about the SystemC modeling library. How to model different system designs at different abstraction levels?

Goals

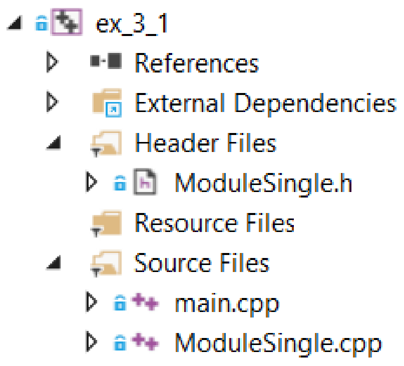
When you have completed this assignment, you will have learned about the SystemC modeling elements like:

- Modules, methods, threads and events
- Signals and Ports
- Communication using events, signals and channels
- Modelling at different abstraction levels using SystemC
- Using SystemC models for modelling of different system designs

Exercise 1

Create a module (ModuleSingle) with a single thread and a method. The thread should notify the method each 2 ms by use of an event and static sensitivity. The method should increment a counter of the type sc_uint<4> and print the value and current simulation time. Limit the simulation time to 200 ms. Describe what happens when the counter wraps around?

Solution



```
ex_3_1
  References
  External Dependencies
  Header Files
    ModuleSingle.h
  Resource Files
  Source Files
    main.cpp
    ModuleSingle.cpp
```

Figure 1 Ex3.1 class overview

ModuleSingle.h

The class definition of ModuleSingle is done is this file. This is done using the systemC macro for module definition: SC_Module(SingleModule).

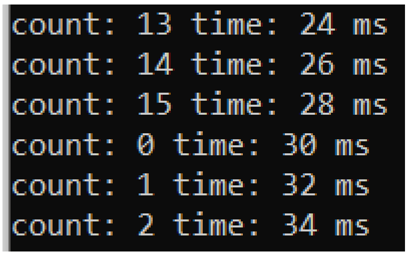
ModuleSingle.cpp

The thread and method used in this exercise are implemented here. Using an event, *event_a,* for which the method is declared statically sensitive the thread function activated the method that prints out every 2 milliseconds.

```
void ModuleSingle::moduleSingleThread()
{
   wait(SC_ZERO_TIME);
   while (true)
   {
      event_a.notify();
      wait(2, SC_MS);
   }
}

void ModuleSingle::moduleSingleMethod()
{
      // increment counter
      counter++;

      // print current count and simulation time
      std::cout << "count: "
         << counter
         << " time: "
         << sc_time_stamp()
         << std::endl;
}
```

Figure 2 Code Snippet from ModuleSingle.cpp

Results



```
count: 13 time: 24 ms
count: 14 time: 26 ms
count: 15 time: 28 ms
count: 0 time: 30 ms
count: 1 time: 32 ms
count: 2 time: 34 ms
```

Figure 3 screen dump from terminal

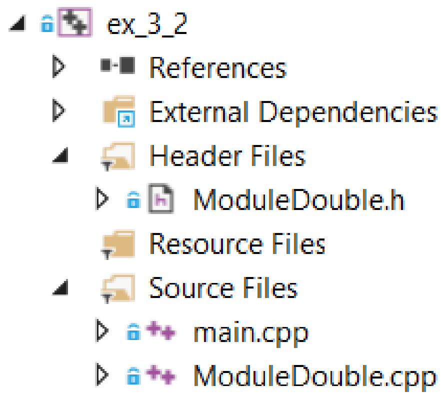
Describe what happens when the counter wraps around?

As can be seen under results the counter, which is declared as a sc_uint<4>, wraps around and increments from 0.

Exercise 2

Create a module (ModuleDouble) with two threads (A, B), one method (A) and four events (A, B, Aack, Back) as shown in Figur 1. Thread A notifies event A every 3 ms and thread B notifies event B every 2 ms. After notification, the thread waits for an acknowledge (event Aack and Back). If acknowledge is not received after a timeout period (A = 3 ms and B = 2 ms) the threads continue notifying event A or B. The method A alternates between waiting on event A and B. Use dynamic sensitivity in the method by calling next_trigger() to define the next event to trigger the method. Let the method print the current simulation time and the notified events

Solution



```
ex_3_2
    References
    External Dependencies
    Header Files
        ModuleDouble.h
    Resource Files
    Source Files
        main.cpp
        ModuleDouble.cpp
```

Figure 4 Ex3.2 class overview

ModuleDouble.h

A class definition of a module containing two threads and one method.

ModuleDouble.cpp

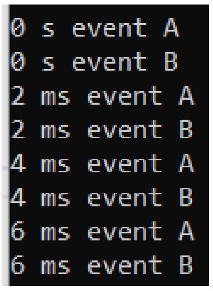
The implementation of ModuleDouble is in this file. Some of the implementation is shown below. The two threads: *A_Thread()* and *B_Thread()* are implemented identically. As described in the exercise description: first a notifies event A, then waiting 3 seconds before waiting for an acknowledge from the method.

The method is implemented as shown below. First the method dynamically awaits the event A using: *next_trigger()* then the acknowledge signal is given.

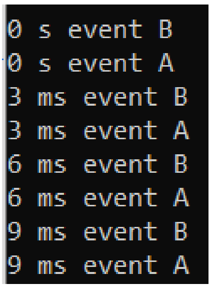
```
void ModuleDouble::A_Thread(void)
{
   wait(SC_ZERO_TIME);
   while (true)
   {
      // notify every 3 ms
      event_A.notify(3, SC_MS);
      wait(3,SC_MS, event_Aack);
   }
}
/*********************************************/

void ModuleDouble::a_method(void)
{

   // wait for next event A
   next_trigger(event_A);
   std::cout << sc_time_stamp() << " event A" << std::endl;
   event_Aack.notify();


   // wait for next event B
   next_trigger(event_B);
   std::cout << sc_time_stamp() << " event B" << std::endl;
   event_Back.notify();

}
```

Figure 5 Code Snippet from ModuleDouble.cpp

Results



```
0 s event A
0 s event B
2 ms event A
2 ms event B
4 ms event A
4 ms event B
6 ms event A
6 ms event B
```



```
0 s event B
0 s event A
3 ms event B
3 ms event A
6 ms event B
6 ms event A
9 ms event B
9 ms event A
```

Figure 6 screen dump of terminal (left: the method first waits for A then B. Right: the method first waits for B then A)

Discussion

Looking at the results it is clear that given the way in which the method waits for the two events, first one then the other. Only one of the delays are in reality followed. The event waited on is the one last called using *next_trigger()* i.e. in the implementation shown above it is the b delay. The reason is that the function *next_trigger()* creates a tempura sensitivity list that overwrites any previous ones each time it is called. Therefore in reality using the shown way of implementing is not really using two events but only one. An alternate implementation would be: *next_trigger(event_A | event_B)* but there would be no way to determine which event triggered the method.

Exercise 3

Create 2 modules that realize a producer and a consumer thread. The modules should be connected together using a sc_fifo channel. Use the structure of a TCP package to simulate the data transmitted over the transmission (fifo) channel. The producer transmits a new TCP package with a random interval between 2-10 ms. The consumer thread must print the simulation time and sequence number each time a new TCP package is received. Use the TCP Header structure as described below with a total package size of 512 bytes. Inspiration can be found in the FifoFilter (Fork.h, when adding two consumers) example project

Extend your model to have two fifo channels and consumers receiving TCP packages on port 1 and 2. The producer must be rewritten to connect to more ports. As illustrated below:

Solution

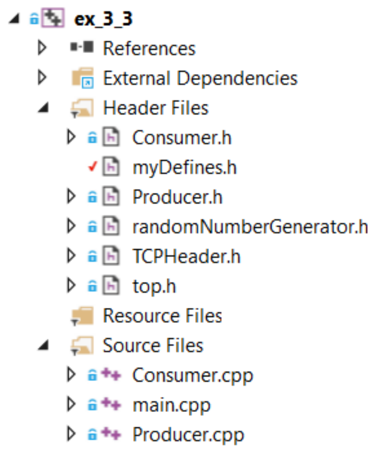


Figure 7 Ex3.3 class overview

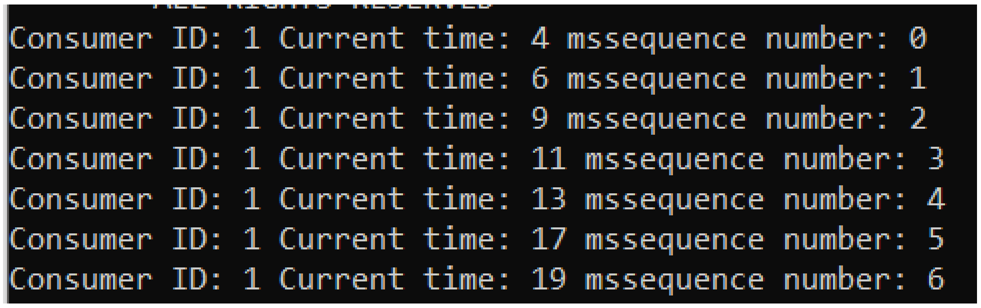
There are several files in this project. Most of which are not of interest to the implementation of the exercise.

Producer.cpp

This file implements the functionality of the producer. Given a random wait time between 2-10 ms. A TCP header is dynamically created. And send to a consumer using the sc_fifo which both are connected to in the top module. If more than one consumer is made then it is essential that as many TCP packets are created. Because in this implementation the consumer assumes ownership of the packet and therefore deletes the memory then finished.

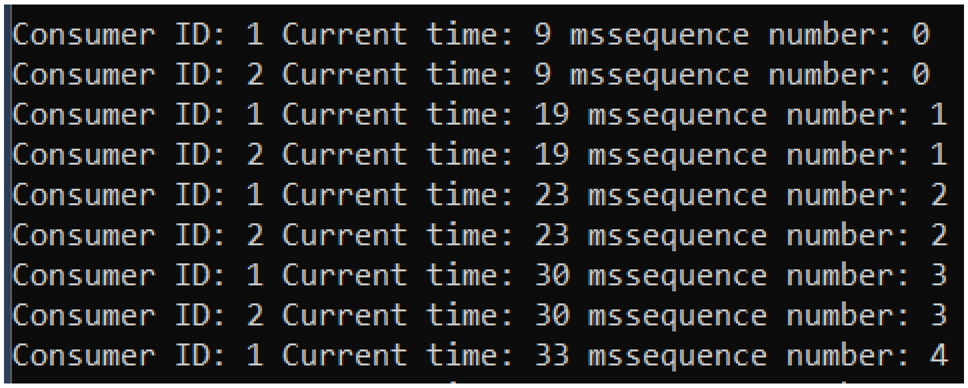
```
     /**************Producer send packets  ***************/
#ifdef EX_3_3_2
     for (int i = 0; i < _fifo.size(); i++)
     {
        TCPHeader* packet = new TCPHeader;
        packet->SequenceNumber = sequence_number;
        _fifo[i]->write(packet);
     }
#else

     TCPHeader* packet = new TCPHeader;
     packet->SequenceNumber = sequence_number;
     _fifo.write(packet);

#endif // EX_3_3_2
```

Results



```
Consumer ID: 1 Current time: 4 mssequence number: 0
Consumer ID: 1 Current time: 6 mssequence number: 1
Consumer ID: 1 Current time: 9 mssequence number: 2
Consumer ID: 1 Current time: 11 mssequence number: 3
Consumer ID: 1 Current time: 13 mssequence number: 4
Consumer ID: 1 Current time: 17 mssequence number: 5
Consumer ID: 1 Current time: 19 mssequence number: 6
```

Figure 8 screen dump from terminal (one comsumer)



```
Consumer ID: 1 Current time: 9 mssequence number: 0
Consumer ID: 2 Current time: 9 mssequence number: 0
Consumer ID: 1 Current time: 19 mssequence number: 1
Consumer ID: 2 Current time: 19 mssequence number: 1
Consumer ID: 1 Current time: 23 mssequence number: 2
Consumer ID: 2 Current time: 23 mssequence number: 2
Consumer ID: 1 Current time: 30 mssequence number: 3
Consumer ID: 2 Current time: 30 mssequence number: 3
Consumer ID: 1 Current time: 33 mssequence number: 4
```

Figure 9 screen dump from terminal (two comsumers)

Exercise 4

Create a cycle accurate communication model of a master and slave module that uses the Avalon Streaming Bus interface (ST). Simulate that a master are transmitting data to a slave module as illustrated in the figures 5-2 and 5-8. The slave should store received data from the master in a text file. Include in the model a situation where the data sink signals ready = '0'. The simulated result should be presented in the GTK wave viewer, so a VCD trace file must be created. It should be possible to configure the channel, error and data size define in a separate header file as illustrated in the below code snippet.

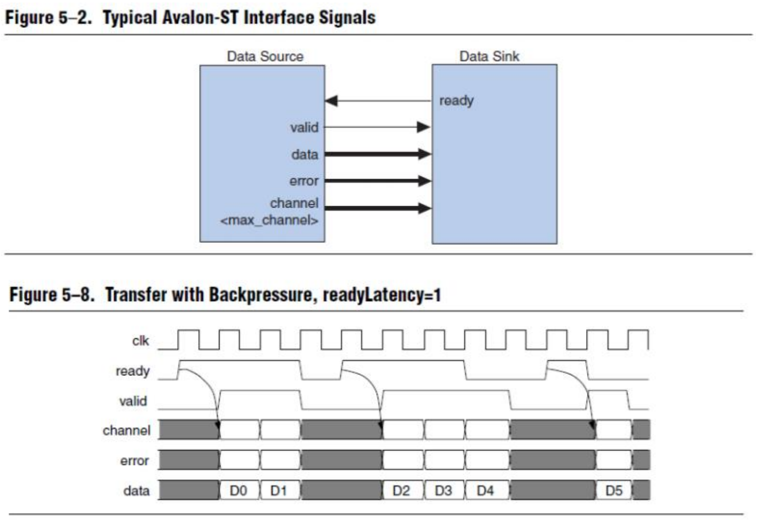
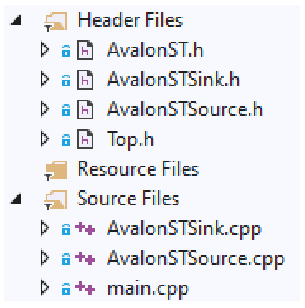


Figure 5–2. Typical Avalon-ST Interface Signals
Data Source
Data Sink
ready
valid
data
error
channel
<max_channel>
Figure 5–8. Transfer with Backpressure, readyLatency=1
clk
ready
valid
channel
error
data
D0
D1
D2
D3
D4
D5

Figure 10 - Figures from assignment

Solution

For this exercise several files have been created as seen on Figure 11 - Ex3.4 class overviewFigure 11.



Header Files
AvalonST.h
AvalonSTSink.h
AvalonSTSource.h
Top.h
Resource Files
Source Files
AvalonSTSink.cpp
AvalonSTSource.cpp
main.cpp

Figure 11 - Ex3.4 class overview

AvalonST.h

This file contains the definitions provided in the assignment for configuration of the bus.

AvalonSTSink

This class implements the Avalon ST Data Sink (Slave). It has four input ports for valid, data, error and channel. One output port for ready. Two input ports for clock and reset. The class has one SC_THREAD that runs the function read(), that is implemented as follows:

```
void AvalonSTSink::read()
{
   sc_logic fake_ready = SC_LOGIC_1;
   sc_int<DATA_BITS> result;

   // Open text file
   output_file = new ofstream(m_file_name.c_str());
   if (!output_file)
   {
      std::cout << "ERROR: Problem opening " << m_file_name << " for output." << endl;
      return;
   }

   // Implements Avalon ST Sink
   while (true)
   {
      if (reset == SC_LOGIC_0)
      {
         wait(clock.posedge_event());
         if (valid == SC_LOGIC_1)
         {
            // Read data to file
            result = data.read();
            *output_file << result << endl;

            // Change ready every 3 samples
            if (result % 3 == 0)
               fake_ready = SC_LOGIC_0;
         }
      }
      else wait(clock.posedge_event());

      // Change ready
      ready.write(fake_ready);
      fake_ready = SC_LOGIC_1;
   }
}
```

Figure 12 - Implementation of AvalonSTSink.read()

The function has two local variables fake_ready and result. The fake_ready variable is used to simulate situations where the data sink is not ready to receive further data.
The function starts by opening a stream to an output file, where the received data is stored. Then the actual implementation is run in an infinite while loop.

If reset is ‘1’ nothing is to happen, therefore we immediately wait for the next clock event. If reset is ‘0’ we check if valid is ‘1’ which means that data is available from the Master. If data is available, it is read from the data port and saved in the output file.

Next the fake ready signal is handled stating that for every 3 data points read the slave is not ready for a clock cycle.

AvalonSTSource
This class implements the Avalon ST Data Source (Master). It has four output ports for valid, data, error

and channel. One input port for ready. Two input ports for clock and reset. The class has one SC_THREAD that runs the function write(), that is implemented as follows:

```
void AvalonSTSource::write()
{
   sc_int<DATA_BITS> counter = 0;

   while (true)
   {
      if (reset == SC_LOGIC_0)
      {
         // Output sample data on negative edge of clock
         while (ready == SC_LOGIC_0)
         {
            wait(clock.posedge_event());
            valid.write(SC_LOGIC_0);
         }

         wait(clock.posedge_event());
         data.write(counter++);
         valid.write(SC_LOGIC_1);  // Signal valid new data
         channel.write(0);         // Channel number
         error.write(0);           // Error
      }
      else wait(clock.posedge_event());
   }
}
```

Figure 13 - Implementation of AvalonSTSource.write()

The data that is transferred to the Slave is a simple counter, that increments every time data is successfully received. The Master waits for the Slave to signal that it is ready and sets the valid signal to '0' as the data is no longer valid after one clock cycle. When the Slave is ready we wait one extra clock cycle, this is to implement the readyLatency = 1 as depicted in the assignment.
Then the counter is written to the data port and valid is set to '1' to indicate new data. The ports 'channel' and 'error' is not used therefore they're set to '0'.

Top.h
Here the top design is described, initializing the modules Sink and Source and providing a clock and reset signal. The modules are connected using sc_signals corresponding to their ports.

The top design also contains the setup of the WaveForm trace, containing all the Top signals.

Main.cpp
Initializes Top and starts the simulation for 200 ns.

Results
Running the solution creates two files "output.txt" and "WaveForm.cvd".

output.txt
Contain the data received by the sink, in this case the number 0-6 as the simulation is only run for 200 ns.

WaveForm.cvd

Contain the trace of the signals from the simulation as seen on Figure 14.

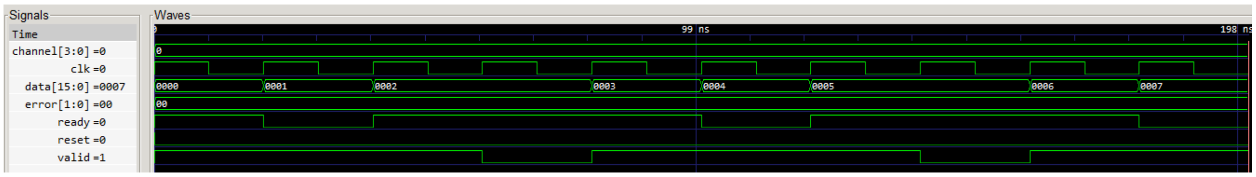


Figure 14 - WaveForm from simulation

From the waveform it is seen that the streaming interface is implemented correctly. When ready is high data is transferred one clock cycle later, indicated by valid going high. When ready goes low data stops being transferred one clock later as there is a slight delay before the Master can react on the change, as expected.

Exercise 5

Implement a model that demonstrates a system design that transfer data at the TLM level refined to BCAM level. Use the sc_fifo to model communication at the TLM level and refine it to BCAM using adapters as inspiration study the example project **SmartPitchDetector** (InAdapter.h and OutAdapter.h). Here a master sends data to a slave using a sc_fifo and an adapter that converts to the bus cycle accurate interface on the receiving slave. Use the model from exercise 3.4 for the interface at the Avalon-ST sink interface for the slave as illustrated below.

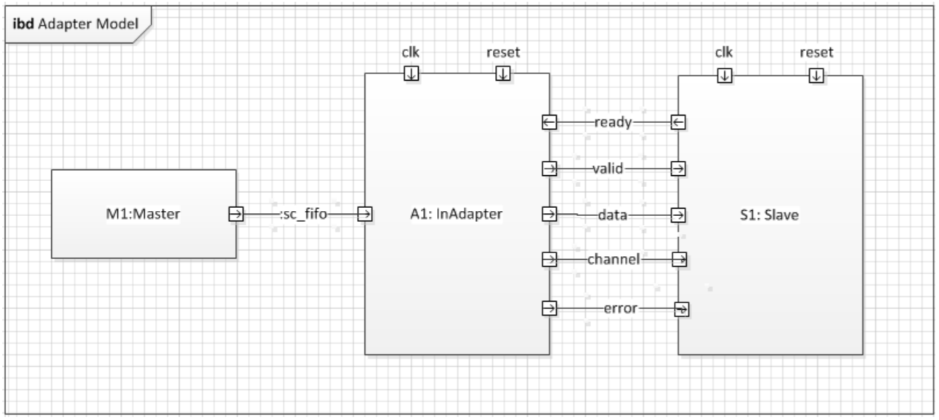


Figure 15 - Figure from assignment

Solution

For this exercise the files from Exercise 4 are reused, except the AvalonSTSource. New files are: “Master” and “InAdapter”. To use the adapter changes has been made to “Top.h”

InAdapter.h

This class implements the sc_fifo_out interface, which allows the model to be refined from using a sc_fifo into using a cycle accurate streaming bus, in this case the Avalon ST bus. The implementation is copied from the assignment.

Master.h

Implements a simple Master that has a sc_fifo_out port and a write() SC_THREAD. In the write() function the Master transmits an incrementing counter using the sc_fifo port.

Top.h

As before Top implements the top design, that initializes the modules and connects them. The InAdapt class is used by assigning it to the Master sc_fifo_out port as seen below:

```
    Top(sc_module_name name) :
       sc_module(name),
       clock("clock", sc_time(CLK_PERIODE, SC_NS)),
       master("master"),
       inAdapt("inAdapt"),
       sink("sink", std::string("output.txt"))
    {
       // Master
       master.out(inAdapt);

       // Adater
       inAdapt.clock(clock);
       inAdapt.reset(s_reset);
       inAdapt.ready(s_ready);
       inAdapt.valid(s_valid);
       inAdapt.channel(s_channel);
       inAdapt.error(s_error);
       inAdapt.data(s_data);

 . . . .
```

Figure 16 - Code snippet from Top.h

Results

From the simulation a waveform is generated as seen on Figure 17.

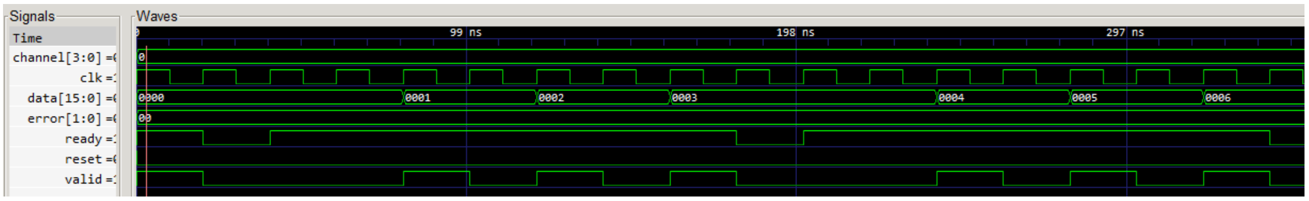


Figure 17 - WaveForm from simulation

From the waveform is seen that the InAdapt implements a Avalon ST bus where data is sent every two clock cycles, hence the oscillation on the valid signal. Also it has a Ready Latency of 1 clock.