ERTS Assignment 1

System level modeling using systemc

Jonas Kjær Rask

Mads Krabsen

2019

# 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 and print the value and current simulation time. Limit the simulation time to 200 ms. Describe what happens when the counter wraps around?

## Solution

## Results

# 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.

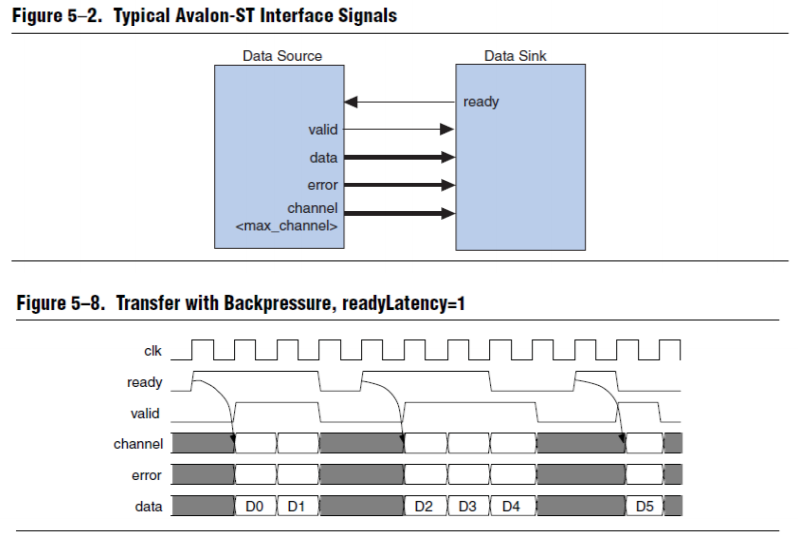


Figure - Figures from assignment

## Solution

For this exercise several files have been created as seen on Figure 2.

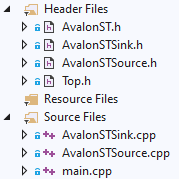


Figure - Ex3.4 class overview

**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 - 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 - 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 5.

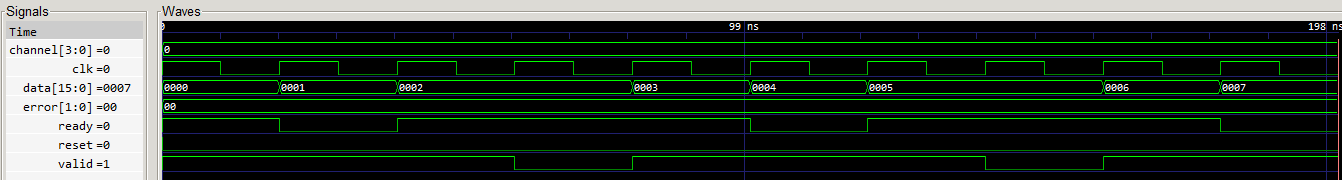


Figure - WaveForm from simulation

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.

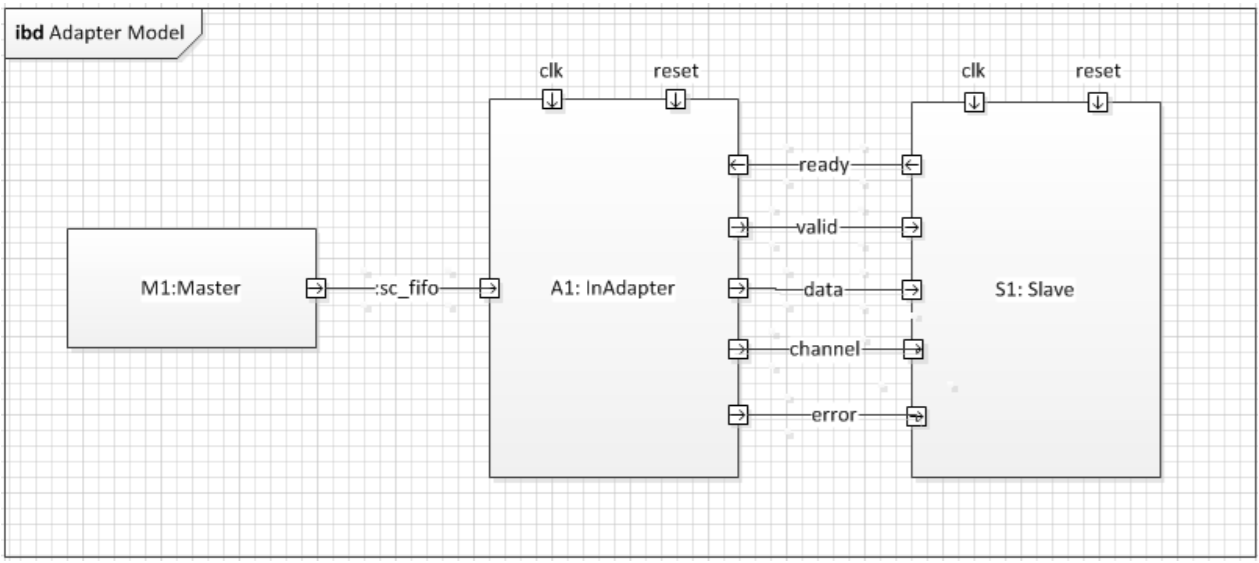


Figure - Figure from assignment

## 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 - Code snippet from Top.h

## Results

From the simulation a waveform is generated as seen on Figure 8.

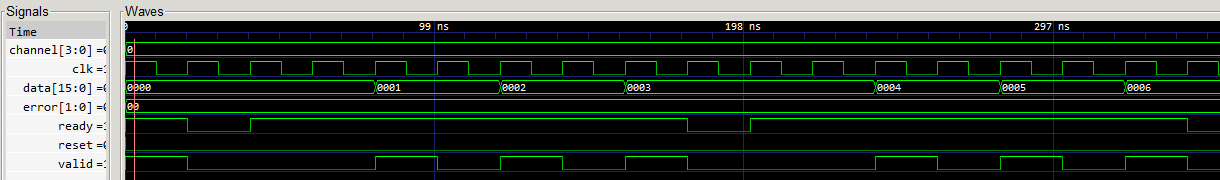


Figure - WaveForm from simulation

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.