Aarhus Universitet

ERTS - Group 2

Rapport

Assignment 1

Gruppemedlemmer:	Studienumre:
Daniel Tøttrup	201509520
Johan Vasegaard Jensen	201505665
Stinus Lykke Skovgaard	201401682

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

16. september 2019

Kapitel 1

ASSIGNMENT 3.1

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?

As stated in the problem a module named "ModuleSingle" are created. In this module a thread and a method were defined. The thread must notify the method every 2 ms, using an event and static sensitivity. So, an event called "incrementEvent" and static sensitivity on this event were implemented. The method should increment a counter, so a counter was also implemented.

Below is the code for the ModuleSingle.h, comments in the code will show how it works.

1.1 ModuleSingle.h

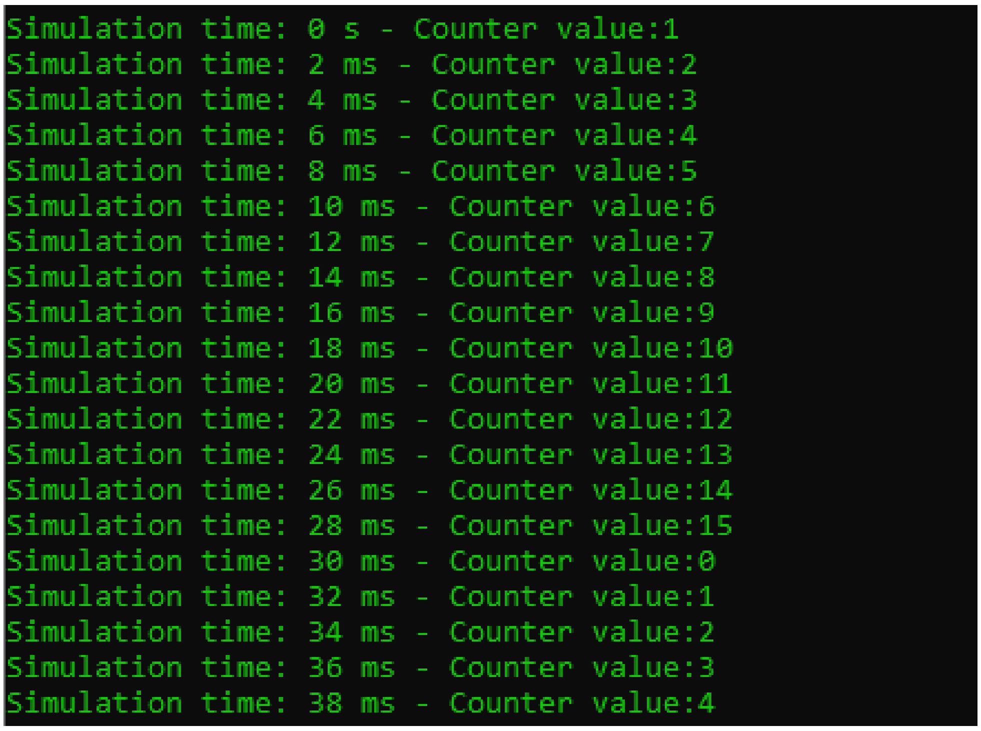
```
#pragma once
#include <systemc.h>
SC_MODULE(ModuleSingle) {
// Module has an event, which is notified in the thread.
sc_event incrementEvent;

sc_uint<4> counter;

SC_CTOR(ModuleSingle)
{
// Define process/thread
SC_THREAD(ModuleSingleProcess);

// Define method triggered in thread.
SC_METHOD(ModuleSinglePrintFunction);

// Make sure that method doesn't run when module is initialized.
dont_initialize();

// Set up static sensitivity.
sensitive << incrementEvent;
}

// Function prototypes
void ModuleSinglePrintFunction(void);
void ModuleSingleProcess(void);
};
```

1.2 ModuleSingle.cpp

The implementation of the two functions used by the thread and the method is showed below. The ModuleSinglePrintFunction is used by the method, which increment the counter and prints out a time stamp and the counter value. The ModuleSingleProcess is used by the thread. It just notifies the incrementEvent, which triggers the method, and waits for 2 ms.

```
#include "ModuleSingle.h"

void ModuleSingle::ModuleSinglePrintFunction(void)
{
// Increment counter
counter++;

// Print simulation time and counter value
cout << "Simulation time: " << sc_time_stamp() << " - Counter
    value:" << counter << endl;
}

void ModuleSingle::ModuleSingleProcess(void)
{
// Set start value for counter.
counter = 0;

// While 1 to ensure continuity
while (1)
{
// Notify event
incrementEvent.notify();

// Wait for 2 MS (simulation time)
wait(2,SC_MS);
}
}
```

The main.cpp just creates an instance of the ModuleSingle and calls sc_start(200, SC_MS); to make it run for 200ms. Below is a screenshot of the Console.



```
Simulation time: 0 s - Counter value:1
Simulation time: 2 ms - Counter value:2
Simulation time: 4 ms - Counter value:3
Simulation time: 6 ms - Counter value:4
Simulation time: 8 ms - Counter value:5
Simulation time: 10 ms - Counter value:6
Simulation time: 12 ms - Counter value:7
Simulation time: 14 ms - Counter value:8
Simulation time: 16 ms - Counter value:9
Simulation time: 18 ms - Counter value:10
Simulation time: 20 ms - Counter value:11
Simulation time: 22 ms - Counter value:12
Simulation time: 24 ms - Counter value:13
Simulation time: 26 ms - Counter value:14
Simulation time: 28 ms - Counter value:15
Simulation time: 30 ms - Counter value:0
Simulation time: 32 ms - Counter value:1
Simulation time: 34 ms - Counter value:2
Simulation time: 36 ms - Counter value:3
Simulation time: 38 ms - Counter value:4
```

Figur 1.1: A screenshot of the console

Kapitel 2

ASSIGNMENT 3.2

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.

For this exercise two threads, a method and four events were created. Below is a snippet code from ModuleDouble.h.

2.1 ModuleDouble.h

```
#pragma once
#include <systemc.h>
SC_MODULE(ModuleDouble) {

// Events used by used by method.
sc_event A, B, Aack, Back;

// Bool used to keep track of method notified.
bool next_trigger_from_A = true;

SC_CTOR(ModuleDouble)
{
// Threads in module.
SC_THREAD(Thread_A);
SC_THREAD(Thread_B);

// Method
SC_METHOD(Method_A);
// Static sensitivity used to start method. Then overwritten by
    dynamic sensitivity.
sensitive << A;
// To make sure that method isn't called when module is initialized.
dont_initialize();
}

// Function prototypes.
void Thread_A(void);
void Thread_B(void);
void Method_A(void);
};
```

2.2 ModuleDouble.cpp

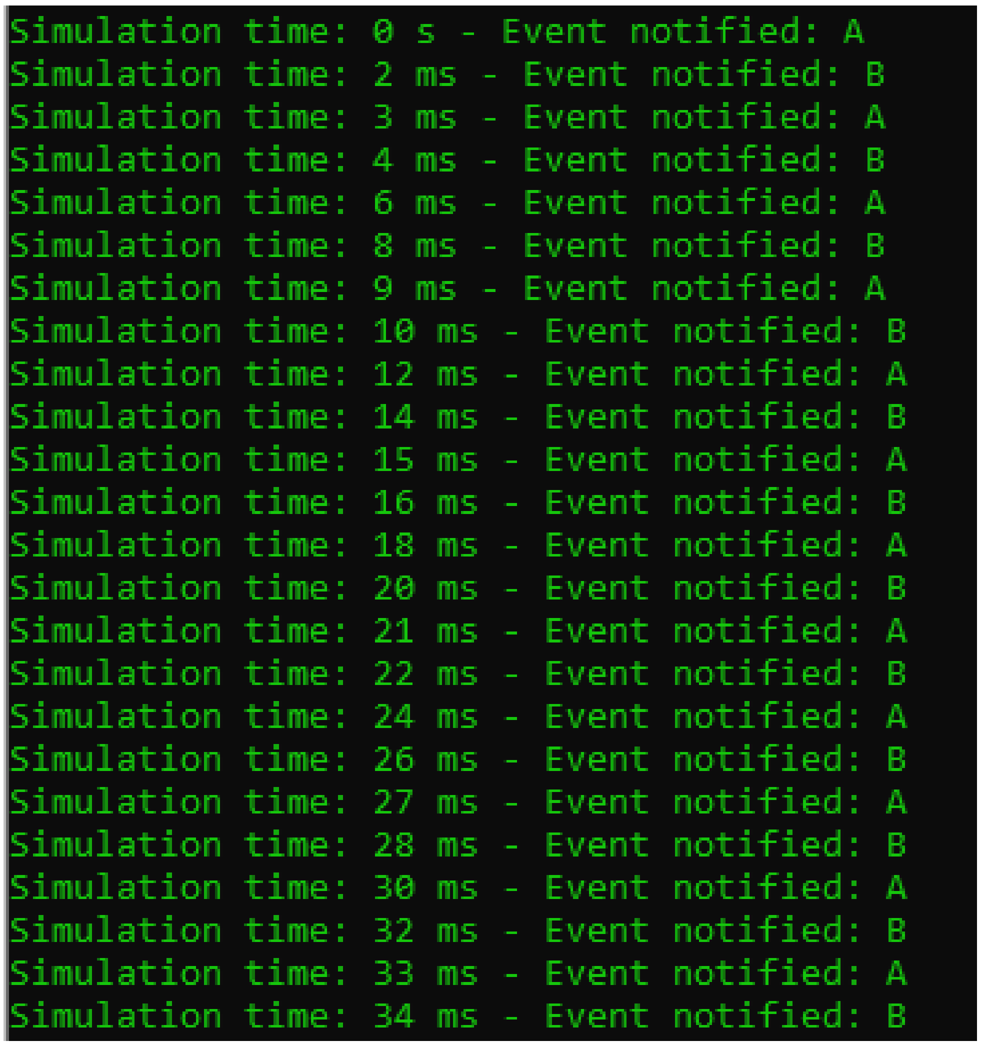
The implementation of Thread_A and Thread_B is showed in the code snippet below. Thread_A notifies event A every 3ms, or when event Aack is notified. Thread_B notifies event B every 2ms, or when even Back is notified.

Method_A must print out the simulation time and the event notified. To check which event that triggered Method_A an if-statement is implemented. The if -statement checks a Boolean called next_trigger_from_A. If that Boolean is true, the Event Aack is notified, and the call next_trigger(B) is used, to change the dynamic sensitivity of the process, so next time its triggered event B is notified.

```
#include "ModuleDouble.h"
#include <string>
using namespace std;

void ModuleDouble::Thread_A(void)
{
// Notifies event A every 3ms (simulation time).
// Thread is "woken up" if Aack-event is notified.
while (1)
{
A.notify();
wait(3, SC_MS, Aack);
}
}

void ModuleDouble::Thread_B(void)
{
// Notifies event B every 2ms (simulation time).
// Thread is "woken up" if Back-event is notified.
while (1)
{
B.notify();
wait(2, SC_MS, Back);
}
}

void ModuleDouble::Method_A(void)
{
```

```
string str;

// Check whether the event that triggered the method was event_A og
    event_B.
// Write the name of the event to the string, notify the
// correct acknowledge-event and set up dynamic sensitivity for the
    other event.
if (next_trigger_from_A)
{
next_trigger_from_A = false;
str = "A";
Aack.notify();
next_trigger(B);

}
else
{
next_trigger_from_A = true;
str = "B";
Back.notify();
next_trigger(A);
}

// Print simulation time and event notified.
cout << "Simulation time: " << sc_time_stamp() << " - Event
    notified: " << str << endl;
}
```

The main.cpp just creates an instance of the ModuleDouble and calls sc_start(200, SC_MS); to make it run for 200ms. Below is a screenshot of the Console.



```
Simulation time: 0 s - Event notified: A
Simulation time: 2 ms - Event notified: B
Simulation time: 3 ms - Event notified: A
Simulation time: 4 ms - Event notified: B
Simulation time: 6 ms - Event notified: A
Simulation time: 8 ms - Event notified: B
Simulation time: 9 ms - Event notified: A
Simulation time: 10 ms - Event notified: B
Simulation time: 12 ms - Event notified: A
Simulation time: 14 ms - Event notified: B
Simulation time: 15 ms - Event notified: A
Simulation time: 16 ms - Event notified: B
Simulation time: 18 ms - Event notified: A
Simulation time: 20 ms - Event notified: B
Simulation time: 21 ms - Event notified: A
Simulation time: 22 ms - Event notified: B
Simulation time: 24 ms - Event notified: A
Simulation time: 26 ms - Event notified: B
Simulation time: 27 ms - Event notified: A
Simulation time: 28 ms - Event notified: B
Simulation time: 30 ms - Event notified: A
Simulation time: 32 ms - Event notified: B
Simulation time: 33 ms - Event notified: A
Simulation time: 34 ms - Event notified: B
```

Figur 2.1: A screenshot of the console

Kapitel 3

ASSIGNMENT 3.3

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.

For this problem two modules are implemented; a Consumer and a Producer module. A code snippet of the Costumer module is snippet below. The consumer module implements sc_fifo_in of the type <TCPHeader> because its supposed to read from the FIFO.

3.0.1 Consumer.h

```
#pragma once
#include <systemc.h>
#include "TCPHeader.h"

SC_MODULE(Consumer)
{
sc_fifo_in <TCPHeader*> in;
TCPHeader *header;
SC_CTOR(Consumer)
{
SC_THREAD(ConsumerThread);
}

void ConsumerThread(void);
};
```

3.0.2 Producer.h

The same goes for the Producer module just vise versa it implements sc_fifo_out of the type <TCPHeader> because it's supposed to write to the FIFO. The code snippet of the Producer module header file is pictured below.

```
#pragma once
#include <systemc.h>
#include "TCPHeader.h"

SC_MODULE(Producer)
{
sc_fifo_out <TCPHeader*> out;
SC_CTOR(Producer)
{
SC_THREAD(ProducerThread);
}

void ProducerThread(void);
};
```

3.0.3 TCPHeader.h

In the header file for TCPheader the code snippet from the exercise was used as a template for the structure of the TCPheader. The implementation can be seen below. Comments in the code explain the implementation.

```
#pragma once
#include <systemc.h>
#define PACKET_SIZE 512
#define DATA_SIZE (PACKET_SIZE-20)

class TCPHeader
{
sc_uint<16> SourcePort;
sc_uint<16> DestinationPort;
sc_uint<32> SequenceNumber;
sc_uint<32> Acknowledge;
sc_uint<16> StatusBits;
sc_uint<16> WindowSize;
sc_uint<16> Checksum;
sc_uint<16> UrgentPointer;
char Data[DATA_SIZE];

public:

// Empty constrtuctor
TCPHeader()
{

}

// Parametrized constructor
TCPHeader(sc_uint<16> SourcePort, sc_uint<16> DestinationPort,
    sc_uint<32> SequenceNumber, sc_uint<32> Acknowledge, sc_uint<16>
    StatusBits, sc_uint<16> WindowSize, sc_uint<16> Checksum,
    sc_uint<16> UrgentPointer, char Data[DATA_SIZE])
{
this->SourcePort = SourcePort;
this->DestinationPort = DestinationPort;
this->SequenceNumber = SequenceNumber;
this->Acknowledge = Acknowledge;
this->StatusBits = StatusBits;
this->WindowSize = WindowSize;
this->Checksum = Checksum;
this->UrgentPointer = UrgentPointer;
strcpy(this->Data, Data);
}

sc_uint<32> getSequenceNumber()
{
return SequenceNumber;
}

void setSequenceNumber(sc_uint<32> sn)
{
SequenceNumber = sn;
}

// Overload of '='-operator.
// Creates a new instance with matching attributes.
TCPHeader& operator=(
const TCPHeader& rhs
)
{
SourcePort = rhs.SourcePort;
DestinationPort = rhs.DestinationPort;
SequenceNumber = rhs.SequenceNumber;
Acknowledge = rhs.Acknowledge;
StatusBits = rhs.StatusBits;
WindowSize = rhs.WindowSize;
Checksum = rhs.Checksum;
UrgentPointer = rhs.UrgentPointer;
strcpy(Data, rhs.Data);
return *this;
}
// Overload of '=='-operator.
// Checks if all values are the same, in which case true is
    returned. Otherwise false is returned.
bool operator==(const TCPHeader& rhs)
const {
return SourcePort == rhs.SourcePort
&& DestinationPort == rhs.DestinationPort
&& SequenceNumber == rhs.SequenceNumber
&& Acknowledge == rhs.Acknowledge
```

```
&&StatusBits == rhs.StatusBits
&&WindowSize == rhs.WindowSize
&&Checksum == rhs.Checksum
&&UrgentPointer == rhs.UrgentPointer
&&strcmp(Data, rhs.Data);
}


//Definition of '<<'-operator and sc_trace.
friend ostream& operator<<(ostream& file, const TCPHeader& trans);
friend void sc_trace(sc_trace_file*& tf, const TCPHeader& trans,
    std::string nm);
};
```

3.0.4 Producer.cpp

If we look at the source file for the Producer which is pictured below. It generates a new TCPHeader called header, and write information about the package to the console, before writing the package to the FIFO. Then the sequence number is incremented before it waits for a random amount of time (between 2ms and 10ms).

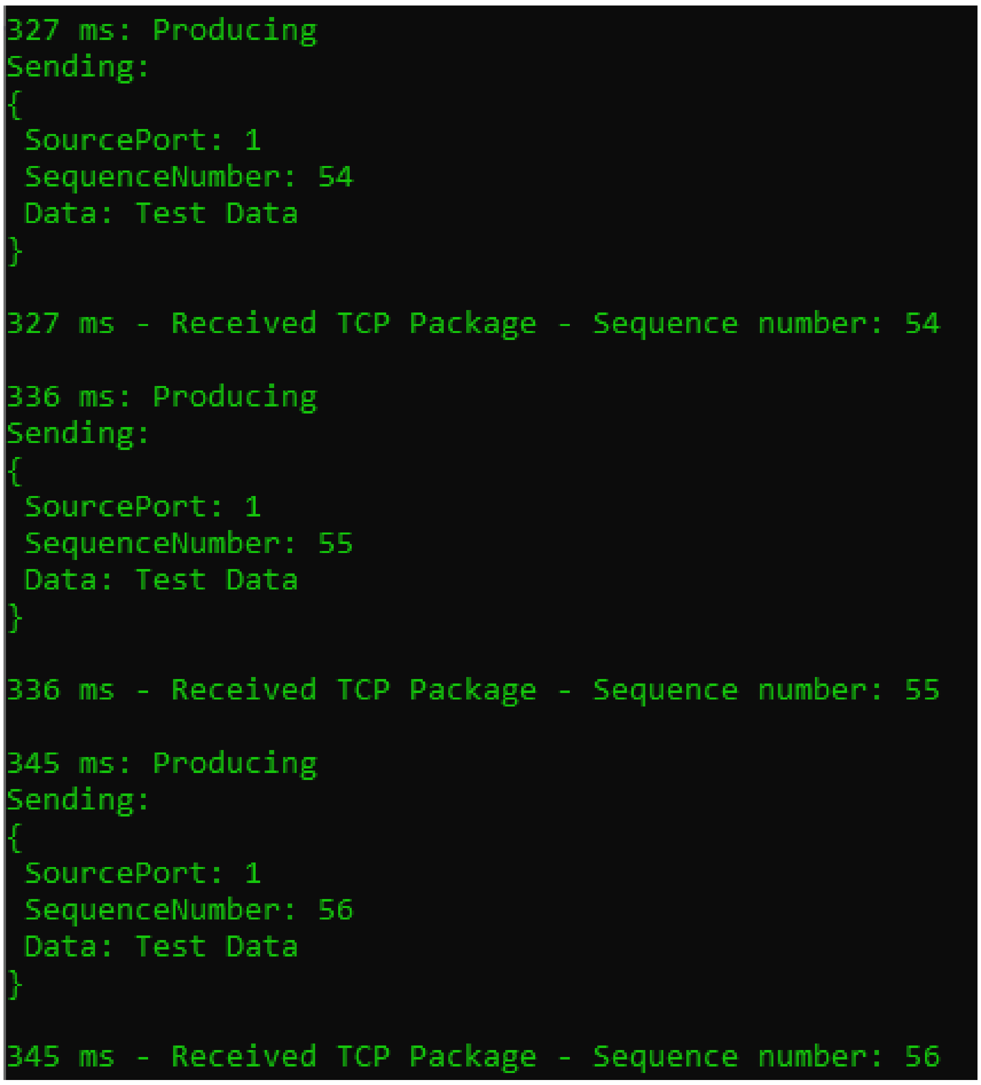
```
#include "Producer.h"

sc_uint<16> SourcePort = 1;
sc_uint<16> DestinationPort = 2;
sc_uint<32> SequenceNumber = 1;
sc_uint<32> Acknowledge = 4;
sc_uint<16> StatusBits = 5;
sc_uint<16> WindowSize = 6;
sc_uint<16> Checksum = 7;
sc_uint<16> UrgentPointer = 8;
char Data[DATA_SIZE] = "Test Data";

void Producer::ProducerThread(void)
{
TCPHeader *header;
while (1)
{
// Generate new TCP header.
header = new TCPHeader(SourcePort, DestinationPort, SequenceNumber,
    Acknowledge, StatusBits, WindowSize, Checksum, UrgentPointer,
    Data);
cout << sc_time_stamp() << ": Producing" << endl;
cout << "Sending: " << endl << *header << endl << endl;

//Write to FIFO
out.write(header);

// Increment sequence number.
SequenceNumber++;

// Wait for 2-10 ms.
int waitTime = rand() % 10 + 2;
wait(waitTime, SC_MS);
}
}
```

3.0.5 Consumer.cpp

The consumer just reads from the FIFO, and writes out the sequence number on the received package to the console.

```
#include "Consumer.h"
#include "TCPHeader.h"

void Consumer :: ConsumerThread(void)
{
while (1)
{
// Blocking read from FIFO
header = in.read();
// Output timestamp and info about TCP package.
cout << sc_time_stamp() << " - Received TCP Package - Sequence
    number: " << (*header).getSequenceNumber() << endl << endl;
}
}$  $
```

Below is a picture of the simulated result from the console. It matches the expected result. The sequence numbers match and the packages are being send in a random interval between 2ms and 10ms.



```
327 ms: Producing
Sending:
{
 SourcePort: 1
 SequenceNumber: 54
 Data: Test Data
}

327 ms - Received TCP Package - Sequence number: 54

336 ms: Producing
Sending:
{
 SourcePort: 1
 SequenceNumber: 55
 Data: Test Data
}

336 ms - Received TCP Package - Sequence number: 55

345 ms: Producing
Sending:
{
 SourcePort: 1
 SequenceNumber: 56
 Data: Test Data
}

345 ms - Received TCP Package - Sequence number: 56
```

Figur 3.1: A screenshot of the console

3.1 Assignment 3.3.2

The producer must be rewritten to connect to more ports. As illustrated below:

```
sc_port<sc_fifo_out_if<TCPHeader *>,0> out;
```

Further, the producer code must be rewritten to send the TCP packet to all connected fifo channels as shown below:

```
for (int i = 0; i < out.size(); i++)
{
out[i]->write(package);
}
```

In this exercise its only the header file for the Producer and the two source files for Producer and Consumer that has changes the rest is identical to the previous exercise.

3.1.1 Producer.h

If we look at the Producer.h. The only thing that has changed is line 7, it is now possible for the producer to connect to five ports.

```
1  #pragma once
2  #include <systemc.h>
3  #include "TCPHeader.h"
4
5  SC_MODULE(Producer)
6  {
7  sc_port<sc_fifo_out_if<TCPHeader*>, 5> out;
8  SC_CTOR(Producer)
9  {
10 SC_THREAD(ProducerThread);
11 }
12
13 void ProducerThread(void);
14 };
```

3.1.2 Producer.cpp

The Producer.cpp hasn't changed mush either. Its now running a for-loop to make sure it writes to all its connected ports.

```
#include "Producer.h"

sc_uint<16> SourcePort = 1;
sc_uint<16> DestinationPort = 2;
sc_uint<32> SequenceNumber = 1;
sc_uint<32> Acknowledge = 4;
sc_uint<16> StatusBits = 5;
sc_uint<16> WindowSize = 6;
sc_uint<16> Checksum = 7;
sc_uint<16> UrgentPointer = 8;
char Data[DATA_SIZE] = "Test Data";

void Producer::ProducerThread(void)
```

```
{
TCPHeader *header;
while (1)
{
header = new TCPHeader(SourcePort, DestinationPort, SequenceNumber,
    Acknowledge, StatusBits, WindowSize, Checksum, UrgentPointer,
    Data);


//Write to FIFO
for (int i = 0; i < out.size(); i++)
{
// Generate new TCP header.
cout << sc_time_stamp() << ": Producing" << endl;
header->setDestinationPort(i + 1);
cout << "Sending: " << endl << *header << endl << endl;

out[i]->write(header);
}

SequenceNumber++;

// Wait for 2-10 ms.
int waitTime = rand() % 10 + 2;
wait(waitTime, SC_MS);
}
}
```

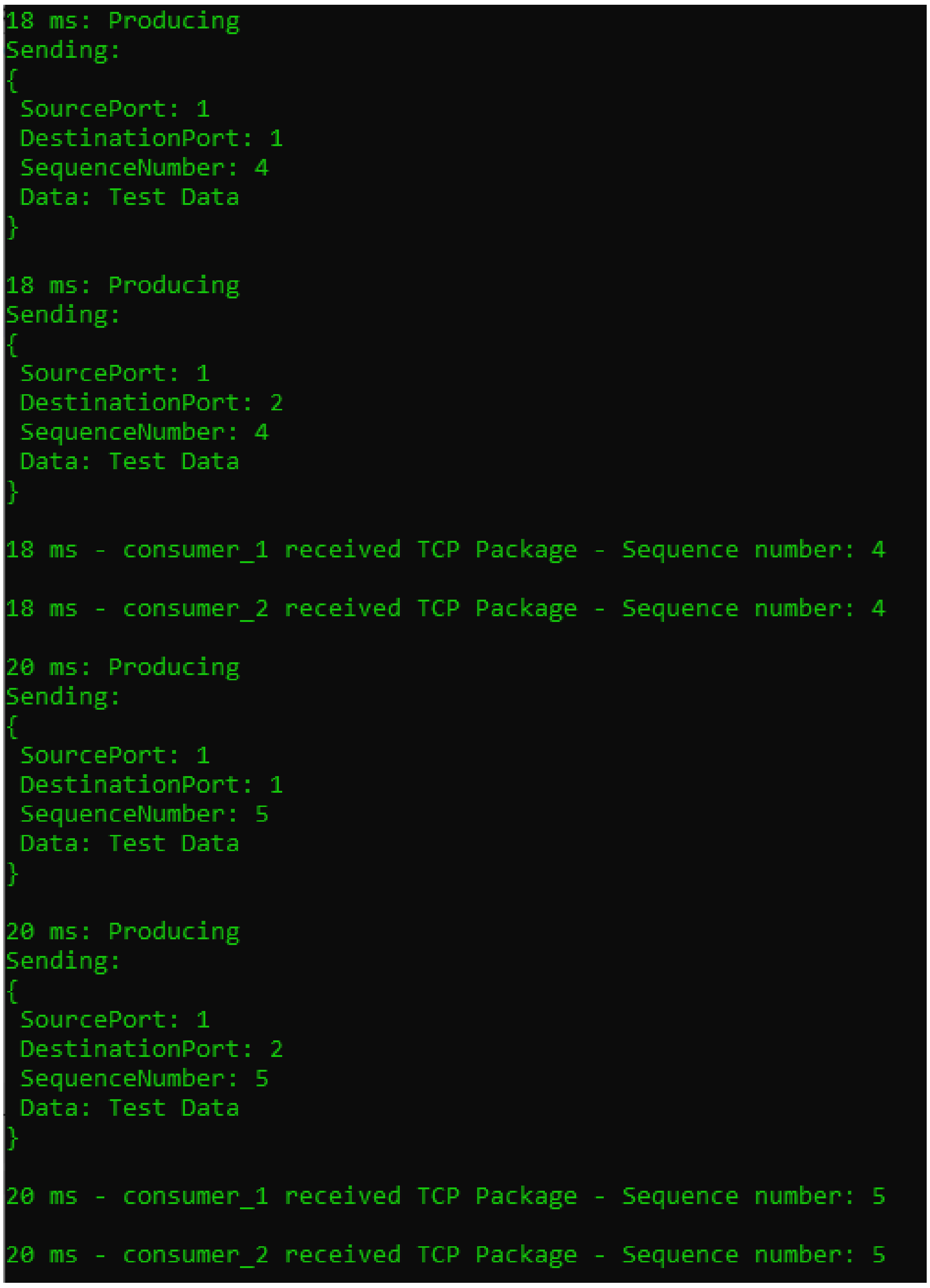
3.1.3 Consumer.cpp

Again the Consumer.cpp hasn't changed mush either. The main difference is the consumer now has to keep track of the process which is done in line 7.

```
 1  #include "Consumer.h"
 2  #include "TCPHeader.h"
 3
 4  void Consumer :: ConsumerThread(void)
 5  {
 6  //To get the current process name
 7  const char* processName =
        sc_core::sc_get_current_process_b()->get_parent()->basename();
 8  while (1)
 9  {
10  header = in.read();
11  cout << sc_time_stamp() << " - " << processName << " received TCP
        Package - Sequence number: " << (*header).getSequenceNumber() <<
        endl << endl;
12  if (strcmp(processName,"consumer_2")==0)
13  {
```

```
delete header;
}

}
}
```

In main.cpp two FIFO channels, one producer and two consumers where created and the simulation time was 120ms. Below is a picture of the simulation result from the console window to show the producer sends data to both consumers.



```
18 ms: Producing
Sending:
{
 SourcePort: 1
 DestinationPort: 1
 SequenceNumber: 4
 Data: Test Data
}

18 ms: Producing
Sending:
{
 SourcePort: 1
 DestinationPort: 2
 SequenceNumber: 4
 Data: Test Data
}

18 ms - consumer_1 received TCP Package - Sequence number: 4

18 ms - consumer_2 received TCP Package - Sequence number: 4

20 ms: Producing
Sending:
{
 SourcePort: 1
 DestinationPort: 1
 SequenceNumber: 5
 Data: Test Data
}

20 ms: Producing
Sending:
{
 SourcePort: 1
 DestinationPort: 2
 SequenceNumber: 5
 Data: Test Data
}

20 ms - consumer_1 received TCP Package - Sequence number: 5

20 ms - consumer_2 received TCP Package - Sequence number: 5
```

Figur 3.2: A screenshot of the console

Kapitel 4

EXERCISE 3.4

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.

A total of of three .h files and 4 .cpp files have been made in this exercise. A Master.h, Slave.h and Top.h with corresponding .cpp files. A main.cpp has also been made.

The code below shows the Master.cpp code. The comments explain how the code works, but the essence of it is just to send an integer(1) and afterwards increment that integer for the next send cycle. That means the master will send the numbers 1...10. The master and slave are setup to use the Avalon Streaming Bus interface. The data transfer uses backpressure with a ready latency set to 1.

4.1 Master.cpp

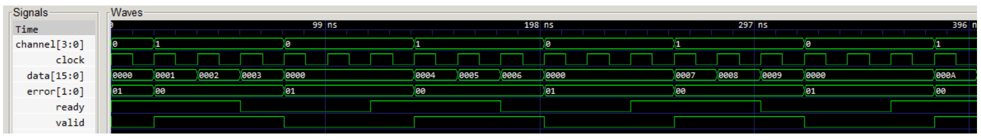
```
void Master::MasterThread(void)
{
  //Arbitrary data to send.
  dataToSend = 1;
  while (1)
  {
    // Update ready state - Will not be updated until next clock
        cycle.
    readyState = ready.read();

    // Check if slave is ready to receive data
    if (readyState)
    {
      // Indicate that data is being written
      valid.write(true);

      // Write data
      data.write(dataToSend);

      //Change data
      dataToSend += 1;

      // Set channel to 1
      channel.write(1);

      // set error to 0
      error.write(0);
    }
    else
    {
      // Indicate that no data is being written
      valid.write(false);

      // Write 0 to data (only to prettify in GTK-viewer).
      data.write(0);

      // Set channel to 0
      channel.write(0);

      // set error to 1
      error.write(1);
    }

    // Wait until next clock cycle.
    wait();
  }
}
```

4.2 Slave.cpp

The slave.cpp can be seen below. Again the comments explains how the code works. Whenever the slave has received 10 integers from the master, it begins to print those numbers to a text file. The slave also changes it's ready state every three clock cycle.

```
void Slave::SlaveThread(void)
{
  // Initial state of "ready".
  ready.write(false);

  // Simulate ready going low for 3 cycles, then high for 3 cycles,
      etc..
  while (1)
  {
    // read new data if any valid data
    if (valid.read())
    {
      test_data = data.read();
      test_data_array[array_index] = test_data;
      array_index++;
      cout << "Slave received data: " << test_data << endl;

      // Print the first 10 numbers recieved by the slave to a text
          file
      if (test_data_array[9] != 0)
      {
        ofstream myfile("Slave_data.txt");
        if (myfile.is_open())
        {
          for (int i = 0; i <= 9; i++)
          {
            myfile << test_data_array[i] << "\n";
          }

          myfile.close();
        }
        else cout << "Unable to open file";
      }
    }

    // Make sure that slave is ready for 3 cycles, then not ready for
        3 cycles.
    if (state_counter < 3)
    {
      ready.write(true);
    }
    else
    {
      ready.write(false);
    }
    state_counter++;
    state_counter = state_counter % 6;

    wait(clk.posedge_event());
  }
}
```

In the Top.cpp file the different modules are being connected to each other using signal variables. It is also here the tracefile is made. A screenshot of the tracefile can be seen below.



Figur 4.1: A screenshot of GTK viewer

According to the figure above we can confirm that the protocol is acting as it should and the model is cycle accurate.

Kapitel 5

EXERCISE 3.5

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

5.1 Master.cpp

In this exercise the master sends 10 integers to the slave. The code for the master can be seen below.

```
void Master::MasterThread(void)
{
  // send the values in the array
  int i = 0;
  while(i<10)
  {
    cout << "Master writing: " << dataToSend[i] << endl;
    data_out.write(dataToSend[i]);
    i++;
  }
}
```

5.2 Slave.cpp

The slave behaves almost as in exercise 3.4 but without the ability to write to a text file. The code for the slave can be seen below.

```
void Slave::SlaveThread(void)
{
  // Initial state of "ready".
  ready.write(false);

  // Simulate ready going low for 3 cycles, then high for 3 cycles,
    etc..
  while (1)
  {
    // read new data if any valid data
    if (valid.read())
    {
      data_read = data.read();
      data_read_array[array_index] = data_read;
      array_index++;
      cout << "Slave received data: " << data_read << endl;
    }

    // Make sure that slave is ready for 3 cycles, then not ready for
        3 cycles.
    if (state_counter < 3)
    {
      ready.write(true);
    }
    else
    {
      ready.write(false);
    }
    state_counter++;
    state_counter = state_counter % 6;

    wait(clk.posedge_event());
  }
};
```

5.3 Top.cpp

The Top.cpp files show how the adaptor is in between the master and slave. On the figure below the code show how this is done.

```
TOP::TOP(sc_module_name nm) :
clock("clock", sc_time(20, SC_NS))
{
  // Set reset to false
  reset.write(false);

  // Create instance of Master, Slave and InAdapter
  slave = new Slave("slave");
  master = new Master("master");
  inAdapter = new InAdapter<sc_int<DATA_BITS>>("inAdapter");

  // Connect inputs and outputs of Slave to signals.
  slave->ready(ready);
  slave->valid(valid);
  slave->clk(clock);
  slave->data(data);
  slave->error(error);
  slave->channel(channel);

  // Connect master to fifo.
  master->data_out(*inAdapter);

  inAdapter->data(data);
  inAdapter->ready(ready);
  inAdapter->valid(valid);
  inAdapter->clk(clock);
  inAdapter->error(error);
  inAdapter->channel(channel);
  inAdapter->reset(reset);

  //Tracefile configuration
  sc_trace_file *tracefile;
  tracefile = sc_create_vcd_trace_file("Avalon_Streaming_Bus");
  if (!tracefile) cout << "Could not create trace file." << endl;
  tracefile->set_time_unit(1, SC_NS); // Resolution of trace file =
      1ns
  sc_trace(tracefile, clock, "clock");
  sc_trace(tracefile, ready, "ready");
  sc_trace(tracefile, valid, "valid");
  sc_trace(tracefile, data, "data");
  sc_trace(tracefile, error, "error");
  sc_trace(tracefile, channel, "channel");

}
```

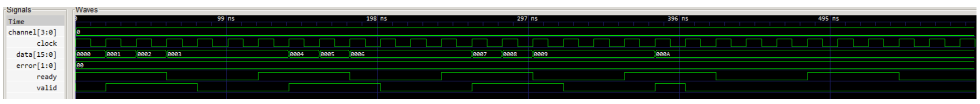
5.4 InAdaptor.cpp

The adaptor makes sure that you get a cycle accurate model when using TLM style communication, like the sc_fifo. The code for the adaptor can be seen below.

```
template <class T>
class InAdapter : public sc_fifo_out_if <T>, public sc_module
{
  public:
  // Clock and reset
  sc_in_clk clk; // Clock
  sc_in<bool> reset; // Reset

  // Handshake ports for ST bus
  sc_in<bool> ready; // Ready signal
  sc_out<bool> valid; // Valid signal

```

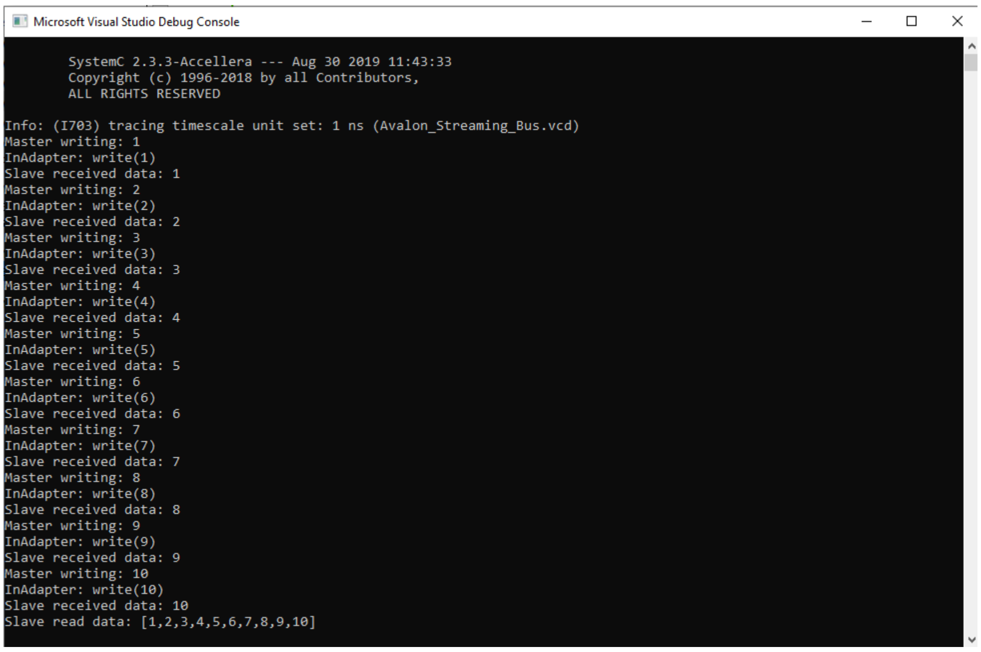
```
// Channel, error and data ports ST bus
sc_out<sc_int<CHANNEL_BITS> > channel;
sc_out<sc_int<ERROR_BITS> > error;
sc_out<sc_int<DATA_BITS> > data;

void write(const T & value)
{
  cout << "InAdapter: write(" << value << ")" << endl;

  // If 'reset' is high, just wait.
  if (reset == false)
  {
    // Output sample data on negative edge of clock

    // Wait until ready is high
    while (ready == false)
    wait(clk.posedge_event());

    // Wait 1 clock cycle to simulate 1 clock cycle delay.
    // wait(clk.posedge_event());

    // Write "high" to valid, indicating that data is being written.
    valid.write(true);

    // Write data
    data.write(value);

    // Write channel and error info
    channel.write(0); // Channel number
    error.write(0); // Error

    // Wait one clock cycle, then indicate that data is not being
        written anymore
    wait(clk.posedge_event());
    valid.write(false);
  }
  else wait(clk.posedge_event());
}


InAdapter(sc_module_name name_)
: sc_module(name_)
{ }
private:
bool nb_write(const T & data)
{
SC_REPORT_FATAL("/InAdapter", "Called nb_write()");
return false;
}
virtual int num_free() const
{
  SC_REPORT_FATAL("/InAdapter", "Called num_free()");
  return 0;
  }
  virtual const sc_event& data_read_event() const
  {
  SC_REPORT_FATAL("/InAdapter", "Called data_read_event()");
  return *new sc_event;
  }
};
```

A tracefile has also been generated for this exercise. It is almost identical to the tracefile from exercise 3.4, which means the adaptor works as intended and the model is cycle accurate. The tracefile can be seen below.



Figur 5.1: A screenshot of GTK viewer

In the console windows it can be seen how the master sends data to the adaptor which redirects the data to the slave.



```
        SystemC 2.3.3-Accellera --- Aug 30 2019 11:43:33
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED

Info: (I703) tracing timescale unit set: 1 ns (Avalon_Streaming_Bus.vcd)
Master writing: 1
InAdapter: write(1)
Slave received data: 1
Master writing: 2
InAdapter: write(2)
Slave received data: 2
Master writing: 3
InAdapter: write(3)
Slave received data: 3
Master writing: 4
InAdapter: write(4)
Slave received data: 4
Master writing: 5
InAdapter: write(5)
Slave received data: 5
Master writing: 6
InAdapter: write(6)
Slave received data: 6
Master writing: 7
InAdapter: write(7)
Slave received data: 7
Master writing: 8
InAdapter: write(8)
Slave received data: 8
Master writing: 9
InAdapter: write(9)
Slave received data: 9
Master writing: 10
InAdapter: write(10)
Slave received data: 10
Slave read data: [1,2,3,4,5,6,7,8,9,10]
```

Figur 5.2: A screenshot of the console