ERTS - Assignment 1

|  |  |
| --- | --- |
| Name | Study number |
| Rasmus B. Langhoff | 201506966 |
| Mads M. Beck | 201508319 |
| Martin W. Kjær | 201509268 |

# Assignment 3.1

In this assignment we were supposed to make a single thread with a single method. The thread notifies the method each 2 ms by using an event and static sensitivity, and increments a counter of type sc\_unit<4>. The method is then supposed to print the value at the current simulation time, and is supposed to run for a total simulation time of 200 ms. The implementation of the method can be seen in the code snippet below in figure 1.

SC\_MODULE(ModuleSingle) {

sc\_event e1;

sc\_uint<4> counter;

SC\_CTOR(ModuleSingle) {

SC\_THREAD(ModuleSingle\_thread);

SC\_METHOD(countAndPrint);

sensitive << e1;

}

void ModuleSingle\_thread(void)

{

for (;;)

{

wait(2, SC\_MS);

e1.notify();

}

}

void countAndPrint(void)

{

std::cout << "Timestamp: " << sc\_time\_stamp() << " Counter: " << ModuleSingle::counter << std::endl;

ModuleSingle::counter++;

}

};

Figure 1 Method for exercise 3.1

As is seen above in figure 1, the thread and method are both defined, and their respective implementations are seen below, which are made accordingly to the specification described by the exercise. The thread runs in a forever loop and uses the *wait()* and *notify()* function. The printed output of the function *countAndPrint()* can be seen below in figure 2:

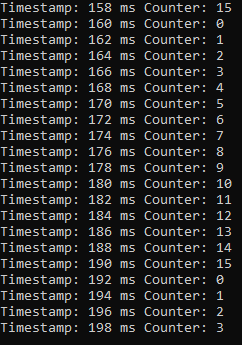


Figure 2 Output from exercise 3.1

# Assignment 3.2

In this assignment, we were supposed to create a module containing two threads and one method, denoted thread A and B, and method A. The module uses four events (A, B, Aack, and Back), where the threads A and B uses the respective events A and B to notify, and uses the method A to alternate between waiting on event A and B, which are notified at different time schedules. We were also supposed to use dynamic sensitivity in the method A by calling *next\_trigger()* to define the next event to trigger the method. Lastly, a method should be implemented to print the current simulation time and the notified events.

Below in figure 3, a code snippet of method A alternating between the two events is shown:

void AMethod(void)

{

if (LastEvent == 'b') //Triggered by event A

{

Aack.notify();

std::cout << "Timestamp: " << sc\_time\_stamp() << " Event A" << std::endl;

next\_trigger(eB);

LastEvent = 'a';

}

else if (LastEvent == 'a') //Triggered by event B

{

Back.notify();

std::cout << "Timestamp: " << sc\_time\_stamp() << " Event B" << std::endl;

next\_trigger(eA);

LastEvent = 'b';

}

else

{

next\_trigger(eA);

LastEvent = 'b';

}

Figure 3 Code snippet of method A from exercise 3.2

And below in figure 4 shown a code snippet of thread A, which is almost the same as thread B except from the time schedule used to notify:

void AThread(void)

{

for (;;)

{

wait(3, SC\_MS);

eA.notify();

wait(3,SC\_MS,Aack);

}

}

Figure 4 Code snippet of thread A from exercise 3.2

As is shown in figure 4, the *Athread()* first waits 3 ms, then notifies event eA, and afterwards waits for event Aack. In figure 3, we can see that the method uses a char named *lastEvent* in order to check which event is triggered, print out the time and which trigger it currently is, notifies the next event trigger, and lastly alternates the *lastEvent* to the other event. An output of the simulation is shown below in figure 5:

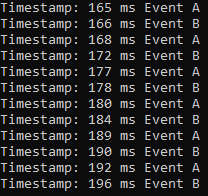


Figure 5 Output from assignment 3.2

As is shown in figure 5, the two events are alternating at the expected time schedules.

# Assignment 3.3

In this third exercise, we were supposed to create two modules which realize a producer and a consumer thread. These modules communicate via a sc­\_fifo channel, and uses the structure of a TCP package to simulate the data transmitted through the fifo channel, which is transmitted at a random interval between 2-10 ms. Again, the output of the simulation is supposed to be printed using the current simulation time and the sequence number of each received TCP package. Below in figure 6 and 7, there is shown code snippets of the TCP producer and consumer respectively:

SC\_MODULE(TCP\_Producer) {

sc\_port<sc\_fifo\_out\_if<TCPHeader \*>, 0> fifo\_out;

int counter = 0;

SC\_CTOR(TCP\_Producer) {

SC\_THREAD(Producer);

}

void Producer(void)

{

for (;;)

{

int number = rand() % 8 + 2;

wait(number, SC\_MS);

for (int i = 0; i < fifo\_out.size(); i++)

{

TCPHeader \*tempMSG = new TCPHeader;

tempMSG->SequenceNumber = counter++;

fifo\_out[i]->write(tempMSG);

}

}

}

};

Figure 6 Code snippet of TCP producer from exercise 3.3

SC\_MODULE(TCP\_Consumer) {

sc\_fifo\_in<TCPHeader\*> fifo\_input;

SC\_CTOR(TCP\_Consumer) {

SC\_THREAD(Consumer);

}

void Consumer(void)

{

for (;;)

{

TCPHeader \*tempMSG = fifo\_input.read();

std::cout << name() << " MSG RECIVED @ "<< sc\_time\_stamp() << " -- SEQ: " << tempMSG->SequenceNumber << std::endl;

delete tempMSG;

}

}

};

Figure 7 Code snippet of TCP consumer from exercise 3.3

The above is actually the extended producer and consumer code, which in are able to connect to multiple ports, which was the objective of assignment 3.3.2. As can be seen on both figure 6 and 7, the producer and consumer use a *sc\_fifo\_out* and *sc\_fifo\_in* respectively, which are both of the type *TCPHeader* which is the structure given by the assignment description.   
In figure 6 it can be seen, that the producer thread first generates a random number, which is used for the following *wait()* function. Afterwards, it iterates through the amount of fifo channels and creates a new TCP message, increment the sequence number, and then transmit it over the fifo channel using the *write()* function.   
In figure 7 it can be seen, that the consumer thread iteratively reads from the fifo channel into a TCP message. It then prints the message along with the timestamp and sequence number, and lastly it deletes the TCP message.

Below the printed simulation output is shown in figure 8, which works as intended with two consumer threads:

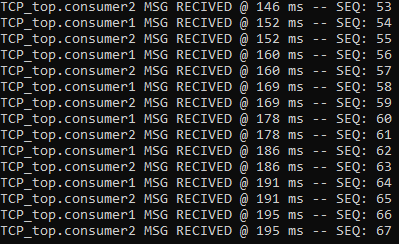


Figure 8 Output from assignment 3.3

# Assignment 3.4

In the fourth assignment, we were supposed to create communication model of a master and slave relationship based upon the Avalon Streaming Bus Interface (ST interface). The actual ST interface to be modelled is transfer with backpressure and readyLatency = 1. The master is supposed to read data from a text file, which it then sends to the slave which writes the received data to another text file. As the simulated response is to be presented using GTK viewer, a trace file is also to be made. In figure 9 and 10 below, code snippets of the ST master thread and ST slave thread respectively are shown:

void entry() //Master

{

valid.write(false);

FILE\* fp\_data;

int tmp\_val;

fopen\_s(&fp\_data,"in\_data\_arch.txt", "r");

for (;;)

{

do

{

wait();

} while (ready == false);

if (fscanf\_s(fp\_data, "%d", &tmp\_val) == EOF)

{

cout << "End of Input Stream: Simulation Stops" << endl;

sc\_stop();

break;

}

data.write(tmp\_val);

valid.write(true);

do

{

wait();

} while (ready == true);

valid.write(false);

}

}

Figure 9 Code snippet of ST master thread

void entry() //Slave

{

ready.write(true);

FILE\* fp\_data;

int tmp\_val;

fopen\_s(&fp\_data, "outputAdapter.txt", "w");

for (;;)

{

do

{

wait();

} while (valid == false);

tmp\_val = (int)data.read();

fprintf\_s(fp\_data, "%d\n", tmp\_val);

ready.write(false);

while (valid == true)

{

wait();

}

wait();

wait();

wait();

ready.write(true);

}

}

Figure 10 Code snippet of ST slave thread

In both figure 9 and 10, the first things we do is setting up the thread to either read/write to/from a given txt-file, depending on whether it is the slave or master.   
In figure 9 it is shown, that after the initial txt-file setup, we wait while ready is false. When this is no longer the case, we write the contents of the txt-file and also write that valid is true to the slave. Then we wait for ready to be false again, so we can write that valid is false to the slave.  
In figure 10 it is shown, that after the initial txt-file setup, we wait until valid is true, as is indicated by the master. We then read the data and print it, before setting ready to false and sending it to the master. Then we wait for the master to tell us that valid is false, before waiting for a total of three clock cycles, before writing ready equals true to the master.

Below in figure 11, the top-level design for the ST interface is shown. Here the various bindings are shown along with the creation of the tracefile along with the trace definitions, which are used to create the wave simulation file.

TopST::TopST(sc\_module\_name name) : sc\_module(name),

master("master"),

slave("slave"),

clock("clock", sc\_time(CLK\_PERIODE, SC\_NS)),

valid("valid"),

data("data"),

error("error"),

channel("channel"),

reset("reset"),

ready("ready")

{

master.valid(valid);

master.data(data);

master.error(error);

master.channel(channel);

master.clk(clock);

master.ready(ready);

master.reset(reset);

slave.valid(valid);

slave.data(data);

slave.error(error);

slave.channel(channel);

slave.clk(clock);

slave.ready(ready);

slave.reset(reset);

tracefile = sc\_create\_vcd\_trace\_file("STWave");

if (!tracefile) cout << "no tracefile for you" << std::endl;

tracefile->set\_time\_unit(1, SC\_NS);

sc\_trace(tracefile, clock, "clock");

sc\_trace(tracefile, ready, "ready");

sc\_trace(tracefile, valid, "valid");

sc\_trace(tracefile, data, "data");

}

TopST::~TopST()

{

sc\_close\_vcd\_trace\_file(tracefile);

}

Figure 11 Code snippet of top-level design from exercise 3.4

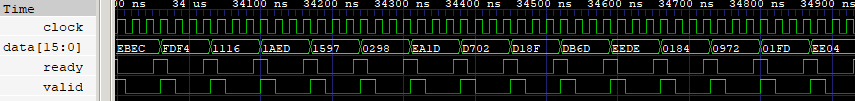
The final output is shown below in figure 12:

Figure 12 Wave simulation of exercise 3.4

As is easily seen, when compared with figure 5.8 in the assignment 1 document, the ST interface works as intended.

# Assignment 3.5

In this last assignment, we were supposed to implement a model which demonstrates the transferring of data the TLM level refined to the BCAM level. This means that the ST interface from assignment 3.4 was to be extended, where the master sends data to the slave using a sc\_fifo channel and an adapter, which is used to convert to the bus cycle accurate interface on the receiving slave. In figure 13 below, the master module is shown:

class FifoMaster : sc\_module {

public:

//Output ports

sc\_in\_clk clk;

sc\_fifo\_out<sc\_int<DATA\_BITS>> fifo;

SC\_CTOR(FifoMaster)

{

SC\_THREAD(entry);

sensitive << clk;

dont\_initialize();

}

void entry()

{

FILE\* fp\_data;

int tmp\_val;

fopen\_s(&fp\_data, "in\_data\_arch.txt", "r");

for (;;)

{

if (fscanf\_s(fp\_data, "%d", &tmp\_val) == EOF)

{

cout << "End of Input Stream: Simulation Stops" << endl;

sc\_stop();

break;

}

fifo.write(tmp\_val);

}

}

};

Figure 13 Code snippet of Master module from exercise 3.5

The master is similar to the one of assignment 3.4, but instead uses a fifo channel as is shown above. The actual adapter is shown below in figure 14. Here it is shown, that the input to the adapter is the masters fifo channel and master clock. The adapter then binds using the ST interface which is then connected to the slave ST interface. Again, we also produce a tracefile and bind the traces as before in exercise 3.4.

TopAdapter::TopAdapter(sc\_module\_name name) : sc\_module(name),

master("master"),

slave("slave"),

clock("clock", sc\_time(CLK\_PERIODE, SC\_NS)),

valid("valid"),

data("data"),

error("error"),

channel("channel"),

reset("reset"),

ready("ready"),

adapter("adapter")

{

master.clk(clock);

master.fifo(adapter);

adapter.clock(clock);

adapter.reset(reset);

adapter.ready(ready);

adapter.valid(valid);

adapter.channel(channel);

adapter.error(error);

adapter.data(data);

slave.valid(valid);

slave.data(data);

slave.error(error);

slave.channel(channel);

slave.clk(clock);

slave.ready(ready);

slave.reset(reset);

tracefile = sc\_create\_vcd\_trace\_file("STWaveAdapter");

if (!tracefile) cout << "no tracefile for you" << std::endl;

tracefile->set\_time\_unit(1, SC\_NS);

sc\_trace(tracefile, clock, "clock");

sc\_trace(tracefile, ready, "ready");

sc\_trace(tracefile, valid, "valid");

sc\_trace(tracefile, data, "data");

}

TopAdapter::~TopAdapter()

{

sc\_close\_vcd\_trace\_file(tracefile);

}

Figure 14 Code snippet from top-level design of exercise 3.5

The wave simulation result is shown below in figure 15:

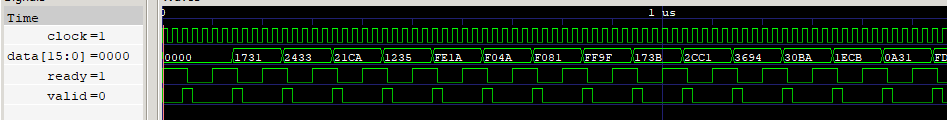


Figure 15 Wave simulation of exercise 3.5

As can be seen above, there are some differences when using a bus cycle accurate model instead. This can be seen when comparing the ready waveform compared to the one of figure 12. The data used in this exercise is the same as used in exercise 3.4, and it has been verified that the data received is the same in both exercises.