Embedded Real Time Systems – Assignment 1
System level modeling using SystemC

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?

You have to hand-in your solution to all exercises described below. Write a short journal (pdf format) including code snippets that explains your solution and simulation results.

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

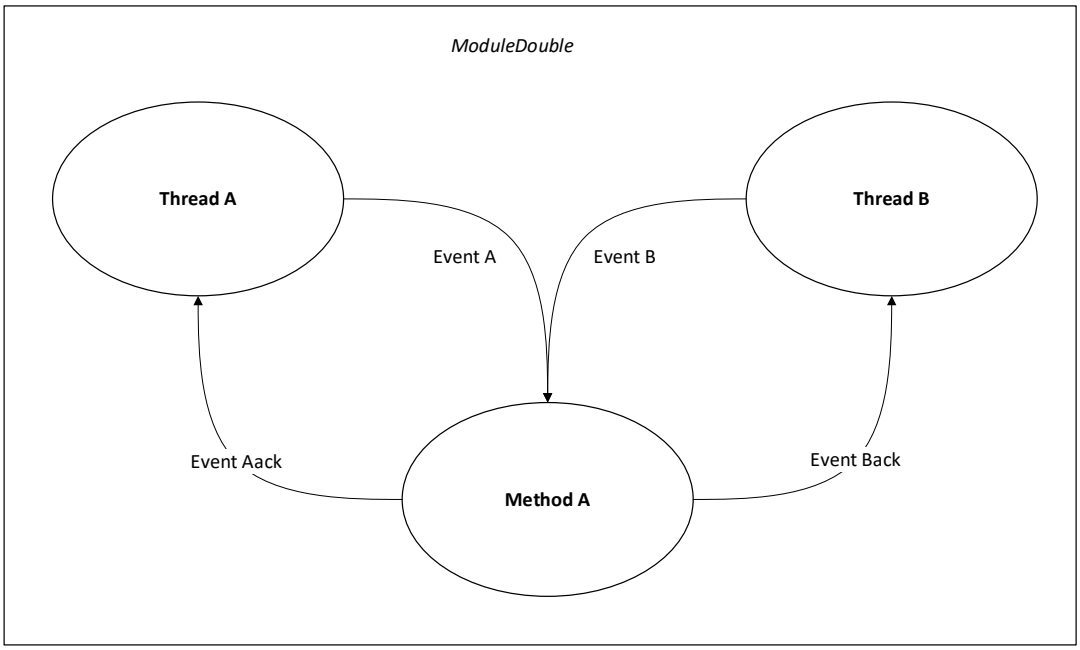
The **examples** folder on the Linux image contains a number of SystemC projects that can be used as inspiration.

The below exercises cover theory from chapters[1] 3 – 7:
(Modules, threads, methods and events)

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?

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.

[1] David C. Black, Jack Donovan, "SystemC: From The Ground Up", Springer 2004



Figur 1 ModuleDouble with method, threads and events

The below exercises cover theory from chapters 8 – 11:
(Channels, signals, hierarchy, communication)

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.

TCP Header

Offsets Octet		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if Data Offset > 5, padded at the end with "0" bytes if necessary) ...																															

Below code snippet shows the structure of the TCP header using SystemC types.

```
#define PACKET_SIZE       512
#define DATA_SIZE         (PACKET_SIZE-20)

typedef struct
{
        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];
} TCPHeader;
```

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:

```
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);
}
```

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.

```
#define CHANNEL_BITS        4    // Number of channel wires
#define ERROR_BITS          2    // Number of error wires
#define DATA_BITS           16   // Number of data wires
#define MAX_CHANNEL         2    // Maximum number of channels actual used
#define CLK_PERIODE         20   // ns
```

Inspiration can be found in the example project **AlgoBlock**. For more details on the ST interface see http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

Figure 5–2. Typical Avalon-ST Interface Signals

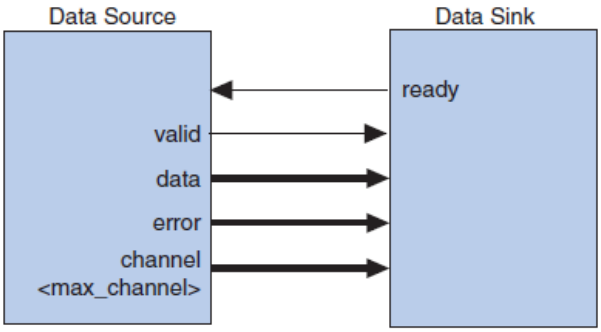
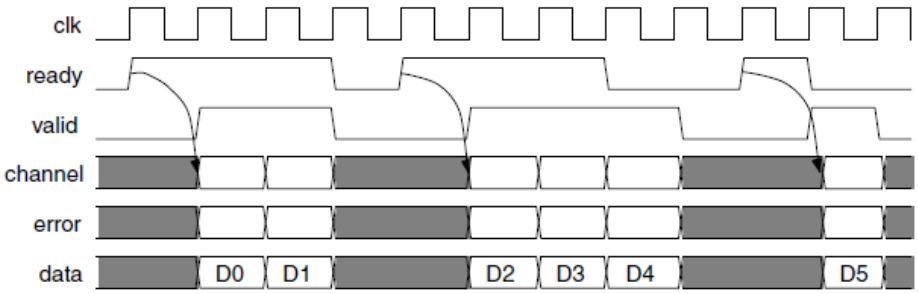
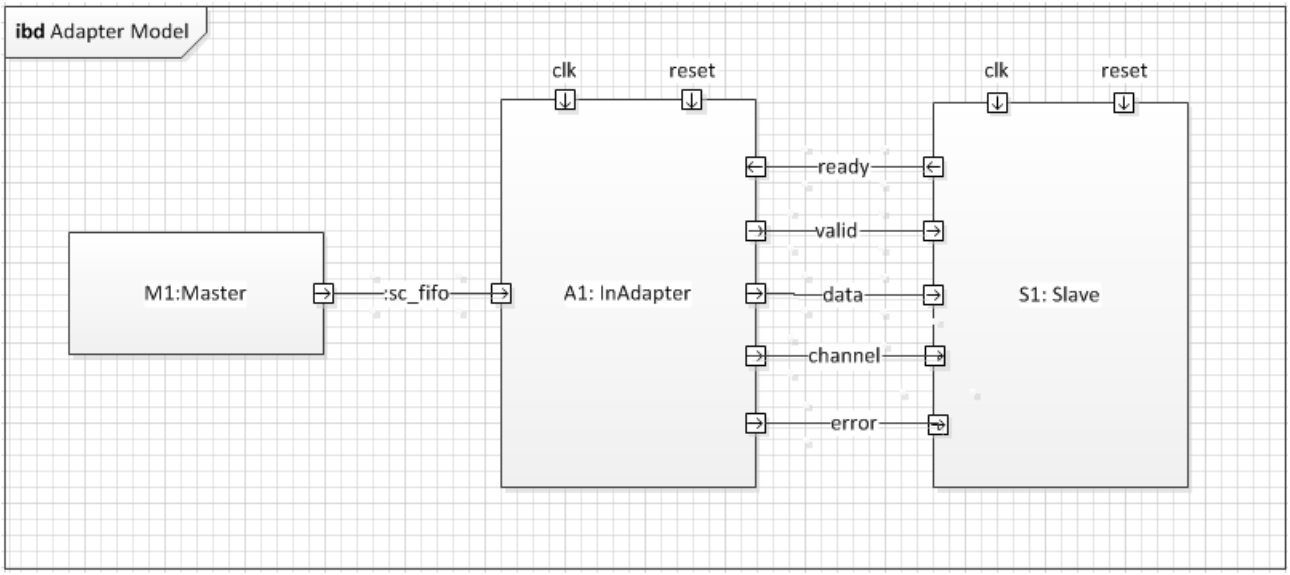


Figure 5–8. Transfer with Backpressure, readyLatency=1



The below exercise relates to TLM and BCAM abstraction of system level modelling:

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.



Use the adapter code below as inspiration for how to convert from a sc_fifo buffer to the Avalon-ST interface.

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

                    // Handshake ports for ST bus
                    sc_in<sc_logic> ready; // Ready signal
                    sc_out<sc_logic> 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)
   {
                    if (reset == SC_LOGIC_0)
                    {
```

```
                                // Output sample data on negative edge of clock
                                while (ready == SC_LOGIC_0)
                                        wait(clock.posedge_event());
                                wait(clock.posedge_event());
                                data.write(value);
                                channel.write(0); // Channel number
                                error.write(0); // Error
                                valid.write(SC_LOGIC_1); // Signal valid new data
                                wait(clock.posedge_event());
                                valid.write(SC_LOGIC_0);
                        }
                        else wait(clock.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;
        }

};
```