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

15. september 2019

Kapitel 1

ASSIGNMENT 3.1

Kapitel 2

ASSIGNMENT 3.2

Kapitel 3

ASSIGNMENT 3.3

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.

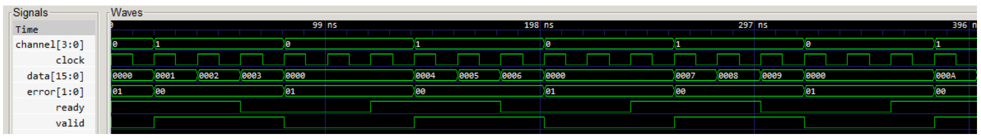
```
void Master::MasterThread(void)
{
  //Arbitrary data to send.
  dataToSend = 1;
  while (1)
  {
    // Update ready state - Will not be updated until next clock
        cycle.
    readyState = ready.read();

    // 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();
  }
}
```

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.

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

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++;
  }
}
```

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());
  }
};
```

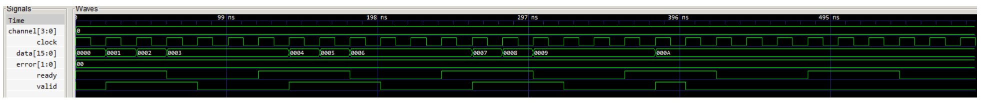
The Top.cpp files show how the adaptor is in between the master and slave. On the figure below the code show how this is done.

```
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");

}
```

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