ECE 5723 Homework 6

By Bruce Huynh

11/23/2021

Introduction:

In this assignment, we work with transaction level modeling to show our understanding of handshaking. We are given an example model *nbFwdBwdMemoryRW* to analyze and understand the inner workings.

A.) Communication Explanation

In the provided .cpp code, there are two main modules. There is the *memWriterReader* module that acts as the initiator and then there is the *memoryUnit* module that acts as the target. *memWriterReader* has a SC_THREAD called *nbMemWR* that starts the transfer process over to the target with a *BEGIN_REQ* signal. This signal goes to the SC_EXPORT of the *memoryUnit*. This happens to be *nb_transport_fw()*. If the transfer is complete without any complications, the return path from the target to the initiator issues a 0. Whenever the return path is 0, this means that **the TLM is accepted.** This occurs for all the signal transfers of the modules. We then hand the process over to the target which sends the *send_ENDREQ* signal to the SC_EXPORT of the initiator. This signal goes to *nb_transport_bw()*. After the end request acts upon the begin request, another signal gets sent from the target to the initiator which is the *send_BEGINRSP* signal which also goes to *nb_transport_bw()*. Finally, the initiator sends an *END_RESP* signal to the target which ends the transaction. All these communications are done with delays to simulate real communication. It takes 20ns between *BEGIN_REQ* and send*_ENDREQ* signals, 80ns between the send*_ENDREQ* and *send_BEGINRSP* signals. Lastly it takes 10ns between *send_BEGINRSP* and *END_RESP*, for a total of 110ns for the full transaction.

B.) Payload Attribute Switching

```
nBlockWriteRead->set_data_length( 3 );
```

```
int data[3];
```

```
if (cmd == tlm::TLM_WRITE_COMMAND) {
    data[0] = (i+1); data[1] = (i+2); data[2] = (i+3);
    //data[3] = (i+4); data[4] = (i+5);
}
```

```
The initiator sends BEGIN_REQ for W data: 1 2 3  at 0 s delay=0 s
```

C.) Removing PEQ

The following is the new *callENDRESP* function:

```
SC_METHOD(callENDRESP);
sensitive << start_ENDRESP;
```

```
void memWriterReader::callENDRESP() {
    if (start_ENDRESP == 1 and sc_time_stamp() != sc_time(0, SC_NS)) {
        cout << "callENDRESP ACTIVE" << endl;
        tlm::tlm_phase forwardPhase = tlm::END_RESP;
        tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
        sc_time delay = sc_time(10, SC_NS);
        cout << "==> The initiator sends END_RESP at " << sc_time_stamp() << " delay=" << delay << endl;
        tlm::tlm_sync_enum returnStatus =
            memWRSocket->nb_transport_fw(*trans, forwardPhase, delay);
        cout << "<-- Return path for END_RESP is " << returnStatus << " at " << sc_time_stamp() << endl;
        doSomethingGood(*nBlockWriteRead);
    }

}
```

It calls the nonblocking forward transport function and it only operates when the time is not 0 and the current phase is BEGIN_RESP. This provides the same results as previously:

```
==> The initiator sends BEGIN_REQ for W data: 1 2 3  at 0 s delay=0 s
<-- Return path for BEGIN_REQ is 0 at 0 s
<== The target sends END_REQ at 0 s delay=20 ns
--> Return path for END_REQ is 0 at 0 s
<== The target sends BEGIN_RESP at 100 ns delay=0 s
--> Return path for BEGIN_RESP is 0 at 100 ns
The data is written in memArray: 1 2 3 at 100 ns
callENDRESP ACTIVE
==> The initiator sends END_RESP at 100 ns delay=10 ns
<-- Return path for END_RESP is 0 at 100 ns
The current transaction was completed at 110 ns
********************************************************************************
==> The initiator sends BEGIN_REQ for W data: 4 5 6  at 130 ns delay=0 s
<-- Return path for BEGIN_REQ is 0 at 130 ns
<== The target sends END_REQ at 130 ns delay=20 ns
--> Return path for END_REQ is 0 at 130 ns
<== The target sends BEGIN_RESP at 230 ns delay=0 s
--> Return path for BEGIN_RESP is 0 at 230 ns
The data is written in memArray: 4 5 6 at 230 ns
callENDRESP ACTIVE
==> The initiator sends END_RESP at 230 ns delay=10 ns
<-- Return path for END_RESP is 0 at 230 ns
The current transaction was completed at 240 ns
```

D. & E.) Change and run communication timing

```
delay = sc_time(50, SC_NS);
cout << "<== The target sends END_REQ at " << sc_time_stamp() << " delay=" << delay << endl;
```

```
tlm::tlm_phase int_phase = tlm::BEGIN_RESP;
delay = delay + sc_time(250, SC_NS);
mem_peq.notify(receivedTrans, int_phase, delay);
```

```
void memWriterReader::callENDRESP() {
    if (start_ENDRESP == 1 and sc_time_stamp() != sc_time(0, SC_NS)) {
        cout << "callENDRESP ACTIVE" << endl;
        tlm::tlm_phase forwardPhase = tlm::END_RESP;
        tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
        sc_time delay = sc_time(30, SC_NS);
```

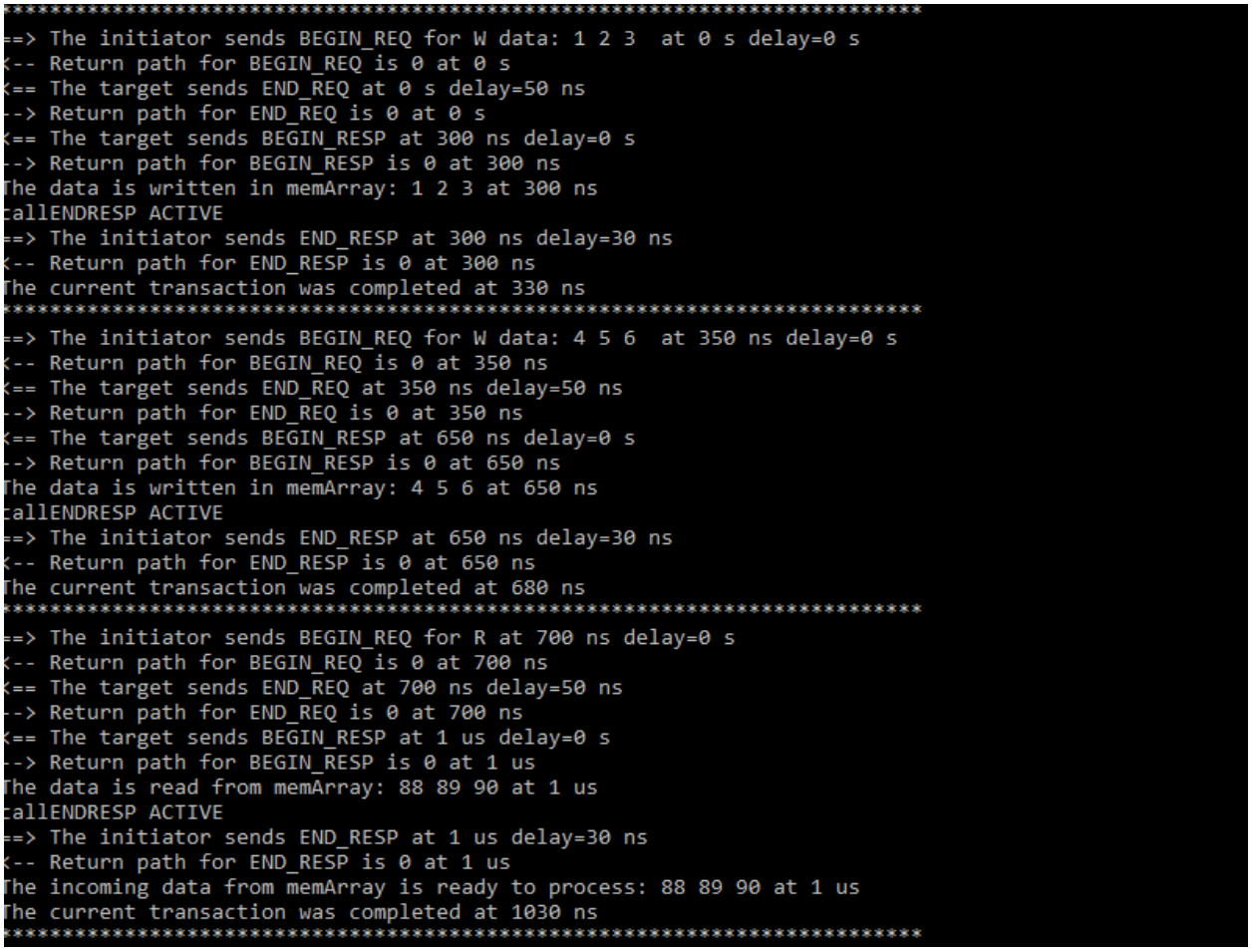
For the first two writes, this is the result.

```
********************************************************************************
==> The initiator sends BEGIN_REQ for W data: 1 2 3  at 0 s delay=0 s
<-- Return path for BEGIN_REQ is 0 at 0 s
<== The target sends END_REQ at 0 s delay=50 ns
--> Return path for END_REQ is 0 at 0 s
********************************************************************************
==> The initiator sends BEGIN_REQ for W data: 4 5 6  at 130 ns delay=0 s
<-- Return path for BEGIN_REQ is 0 at 130 ns
<== The target sends END_REQ at 130 ns delay=50 ns
--> Return path for END_REQ is 0 at 130 ns
********************************************************************************
```

It does not go past END_REQ because we only wait 130 ns for the nbMemWR function.

```
cout << "<-- Return path for BEGIN_REQ is " << returnStatus << " at " << sc_time_stamp() << endl;
wait(130 , SC_NS);
```

If we were to change this to 350 ns (300 ns + 30 ns and another 20 ns between each initiation), we get results that are very similar to the previous testbenches except that the timing is different.



All the steps are the same and all the data is the same, only the time has changed.

Conclusion:

In this assignment, we took a look at transaction level modeling. Through modifying the code, we learn how the code communicates with itself and it causes us to understand the code on a deeper level.