PacketNoc - How To

Fabio Arlati - arlati.fabio@gmail.com

July 2010

Contents

What is this document?

This document is an hands-on guide to the use of the packet NOC component defined in ReSP. The instantiation, connection and manipulation of this component will be described in the next few chapters with the aid of some pictures.

1 Definition

This NOC model is a standard TLM component (which could be attached to other elements in ReSP), but is internally built as a common SystemC component that represents a set of packet-exchanging elements. In particular, the internal architecture of the NOC model is based upon three different components:

1. Master Processing Elements (MPE), which have the purpose of packeting the TLM transactions coming from the attached TLM initiator;

2. Switches, which have to receive, memorize into a buffer, and forward the packets according to a given route;

3. Slave Processing Elements (SPE), which have to unpacket TLM transactions, forward them to the attached TLM target, and send back a packeted answer to the requesting MPE.

For every TLM transaction, the appropriate MPE starts a new session (composed by one or more packets). This session is concluded only when the entire answer (composed, in turn, by one or more packets) is received from the SPE. If the answer is not received before a fixed timeout, the session is restarted, and a new series of requesting packets is created by the MPE. Until a session has not been succesfully completed, the TLM transaction will not be concluded.

2 Instantiation

A simple example of how a packet NOC can be instantiated is contained in the following ReSP architecture file:

`resp-sim/architectures/test/test_packetNoc.py`

In particular, at lines 132-133 the NOC is instantiated starting from a small set of parameters:

- A name for the component (*noc* in the example).
- The path to an `xml` file describing the NOC (contained in the NOC_DESCRIPTION variable of the example, further description in Section 2.1).
- The period of the internal clock of the NOC (calculated starting from the NOC_FREQUENCY variable of the example).

At lines 160/162 and 175/177 the NOC is connected to the initiator components (the masters, usually the processors and their caches). At lines 134-135 the NOC is connected to the target components (the slaves, usually the memories). Keep in mind that the NOC externally provides only two multi passthrough TLM ports (one for targets and one for initiators); every TLM connection assigned to the ports through the `connectPorts` command

```
130
131 if NOC_ACTIVE:
132     clockNoc = scwrapper.sc_time(float(1000)/float(NOC_FREQUENCY), scwrapper.SC_NS)
133     noc = packetNoc32.packetNoc('noc',NOC_DESCRIPTION,clockNoc)
134     connectPorts(noc, noc.initiatorSocket, mem1, mem1.targetSocket)
135     connectPorts(noc, noc.initiatorSocket, mem2, mem2.targetSocket)
136     # Add memory mapping
137     noc.addBinding(5,0x0,memorySize1-1)
138     noc.addBinding(6,memorySize1,memorySize1+memorySize2-1)
```

Figure 1: The instantiation of our NOC

is identified by an index, which is used to access the correct port as in a common array. This index starts from 0 and is automatically incremented for every connection. Internally, the multi passthrough ports are connected to a set of different processing elements: target ports are connected to Master Processing Elements, while initiator ports are connected to Slave Processing Elements. The order in which the internal ports are connected to the elements depends on the order in which the processing elements are described in the `xml` file (see Section 2.1). So, for example, if the Slave PE 5 is declared before the Slave PE 6 (as happens in our example), the first initiator port of the NOC will be associated to SPE 5, while the second initiator port is associated to SPE 6. According to lines 134-135, the component *mem1* is connected to the first initiator port and to SPE 5, while *mem2* is connected to the second initiator port and to SPE 6. Given this situation, we can finally bind different set of adddresses to the different memories, as in lines 137-138.

2.1 XML Description of the NOC

A simple example of how the NOC is described can be found in the following `xml` file:

`resp-sim/architectures/test/test_packetNoc.xml`

The architecture introduced in Section 2 is composed by 4 master components (2 processors, which, according to the Harvard architecture, have both a data and an instruction port) and 2 slave components (2 simple memories). Thus, inside our NOC we need to declare 4 Master Processing Elements and 2 Slave Processing Elements: every element requires a unique ID as an attribute. Between these elements we are free to declare an undefined number of switches with any kind of topology: even the switches require a unique ID attribute. The connections can be declared through the *connect* tag

embedded inside an element, specifing the target ID (connections are mono-directional, if you want a direct return path another *connect* tag should be declared for the inverse direction). The number of connections is limited in the following ways:

- Processing elements are allowed to have only one connection.
- Switches can have up to *PORTS* connections, where *PORTS* is a constant defined in `/resp-sim/components/interconnection/packetNoc/nocCommon.hpp`. The standard value of *PORTS* is 5, if you want to modify it, you should recompile the tool (use `./waf` in the main folder).

Starting from this information the entire NOC is initialized, the SystemC connections between the components are automatically performed, and the routing tables of the switches are automatically computated by applying the Dijkstra algorithm to an ad-hoc Boost Graph. Figure 2 shows the graph obtained by reading the file in our example.

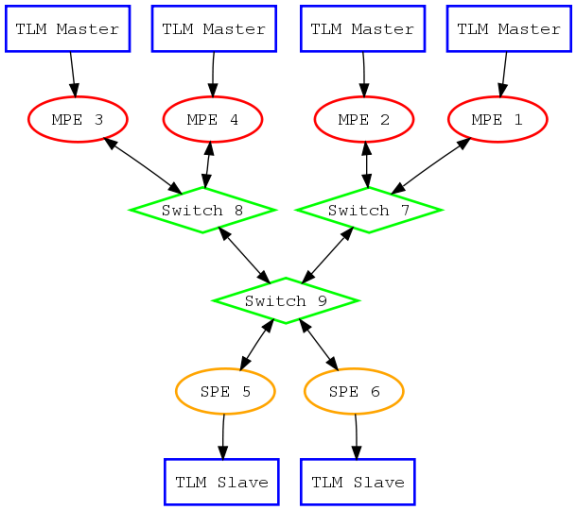

Figure 2: The connection graph of our NOC

3 Interaction

The NOC provides a complete set of functions to modify its behavior at run-time:

printBindings() prints on the standard output the address bindings of the NOC.

printGRT() prints on the standard output all the Routing Tables of the NOC switches.

printStats() prints on the standard output all the current statistics of the NOC elements (flits received, sent and dropped).

getFlitsIn(unsigned int elId) returns the number of flits received by a particular element.

getFlitsOut(unsigned int elId) returns the number of flits sent by a particular element.

getTimeouts(unsigned int masterId) returns the number of sessions timed out in a particular master element.

getAllTimeouts() returns the number of sessions timed out in all master elements.

getDropped(unsigned int switchId) returns the number of flits dropped by a particular switch.

getAllDropped() returns the number of flits dropped by all the switches.

changeTimeout(unsigned int masterId, sc_time tO) changes the timeout value of a particular master element. The standard timeout is 100 ns, and is hardcoded in `/resp-sim/components/interconnection/packetNoc/nocCommon.hpp`. If you wish to change the standard timeout, you should reissue `./waf` in the main folder.

changeAllTimeouts(sc_time tO) changes the timeout value of every master element.

changeBufferSize(unsigned int switchId, unsigned int size) changes the buffer size of a particular switch. The standard buffer size if 1, and is hardcoded in `/resp-sim/components/interconnection/packetNoc/nocCommon.hpp`. If you wish to change the standard buffer size, you should reissue `./waf` in the main folder.

changeAllBufferSizes(unsigned int size) changes the buffer size of every switch.

changePath(unsigned int switchId, unsigned int destinationId, unsigned int nextHop)
changes the forwarding path for a given destination in a particular switch. The `nextHop` parameter requires the ID of a switch (or an element) directly attached to the switch under analysis. The forwarding paths are initialized by the Dijkstra shortest path algorithm applied to the network graph when the NOC is constructed.

An example of how these functions are used can be found in `resp-sim/architectures/test/test_packetNoc.py` at line 223, where all the timeouts of the master elements are set to 1 µs. This operation is necessary in our example, because whenever one of the caches requires to load an entire block of memory, a huge amount of packets is generated during the session; in order to allow the completion of these sessions, a greater timeout is required in the master elements, otherwise every session will reach the standard timeout of 100 ns. It is strongly recommended to use this approach to change the standard parameters of the NOC, instead of modifying the constants in `nocCommon.hpp`.

4 Automatic Interaction

It is possible to instantiate a particular component (named *governor*) which automatically executes a specified routine every specified period of time. This governor is directly attached to the packet NOC, thus it is possible to perform an automated interaction using the functions specified in Section 3.

The governor can be instantiated as shown in `resp-sim/architectures/test/test_packetNoc.py` at line 225. It requires three parameters:

- A name for the component (*governor* in the example).
- The NOC object previosuly instantiated (*noc* in the example).
- The period of the internal clock of the governor (1 ms in the example).

The behavior of the governor is simple: at every clock cycle (i.e. every ms), a routine is executed: this routine is named `activity` and is specified in `resp/components/interconnect/packetNoc/governor/packetNocGovernor.cpp`.

Currently, the routine simply prints the statistics of the NOC at every cycle, but it can be simply expanded to apply optimization strategies to the NOC component. Remember that, after the routine has been changed, a recompilation should be issued by launching `./waf` in the main folder.