Building a Cmpware Network Model

Cmpware, Inc.

Introduction

The *Cmpware Configurable Multiprocessor Development Kit (CMP-DK)* is based around fast simulation models for multiple processors. While the processing nodes themselves may be standard or custom processors, or even hard-wired logic, perhaps the defining element of such a system is its interconnection network. The ability to define and program a multiprocessor is highly dependent on the ability to define the network for the architecture.

The *Cmpware CMP-DK* supplies a set of basic models for both use and for illustration on how to construct new custom models. Once implemented, these network models may be quickly deployed in the simulation environment and even completely changed with a few mouse clicks. This document is provided to give an overview of the the inter-processor network models and process of building network models in the *Cmpware* environment.

The Cmpware Multiprocessor Structure

A multiprocessor system at its most basic is a collection of processing elements connected by some communication network. This definition, while functional, provides for a very wide variety of possible multiprocessor architectures. Because this definition is so flexible, it has been difficult to find design and development tools that adequately support more than a small subset of the multiprocessor architectures made possible by this definition. The *Cmpware CMP-DK* seeks to refine this definition to provide a more limited, but still highly functional definition of multiprocessing.

By limiting the scope of the definition of multiprocessing, *Cmpware* seeks to provide a development environment which supports a variety of useful multiprocessing platforms, while lowering the effort to produce both the software development environment and the actual hardware and software implementation of such architectures. The *Cmpware CMP-DK* seeks to take a relatively small number of components and support the fast and flexible arrangement of these components to produce a complete multiprocessor model. This model will then be used by the development environment, which provides a sophisticated interface for software design and evaluation. The development environment supports three basic types of customizable models for constructing multiprocessors. These are:

- **Processor**: the processor is the element which does much of the work in the multiprocessor. This element is typically a standard instruction set architecture with a standard High Level Language (HLL) compiler used for programming. Nodes may take other forms, including hard wired login, but for the purposes of this discussion they are considered simply the the processing elements that are connected together to produce the multiprocessor.

- **Network**: The network is the hardware used by the processors to communicate with each other. The network is primarily defined by its topology. This may take a variety of regular forms, such as a mesh, which connects each processor to its nearest neighbors. It may implement any regular or irregular connection of processing nodes.

- **Links**: The links are the components used to implement the Network. If the Network is a collection of communication channels, the Links are the implementation of these channels. These may be components such as a shared register or a FIFO and may implement a variety of interfaces used by the processors.

This document is primarily concerned with the implementation of the *Network* model. Reference will be made to both the *Processor* and *Link* models, but these are discussed in more detail elsewhere.

The Interconnection Network Model

The top level view of the multiprocessor architecture used by the *Cmpware* development environment is illustrated in Figure 1. The system consists of two primary layers. At the bottom layer is a collection of processing nodes. These node may be standard processor simulation objects, custom processor simulation objects or even hardwired nodes. The structure and design of the processor simulation objects are discussed in other documents from *Cmpware*. At this time, all that is significant is that each implements the standard `Processor` interface as defined in the Cmpware `com.cmpware.cmp.Processor` class.

The processor simulation models exist in a two dimensional array. While it is possible to interpret this collection of processors in any number of ways, the basic interface assumes this 2D array structure. Above this processor array is the `Multiprocessor` simulation object which defines the interface to the array of processors, as well as supplying the definition and control of the inter-processor interconnection network.

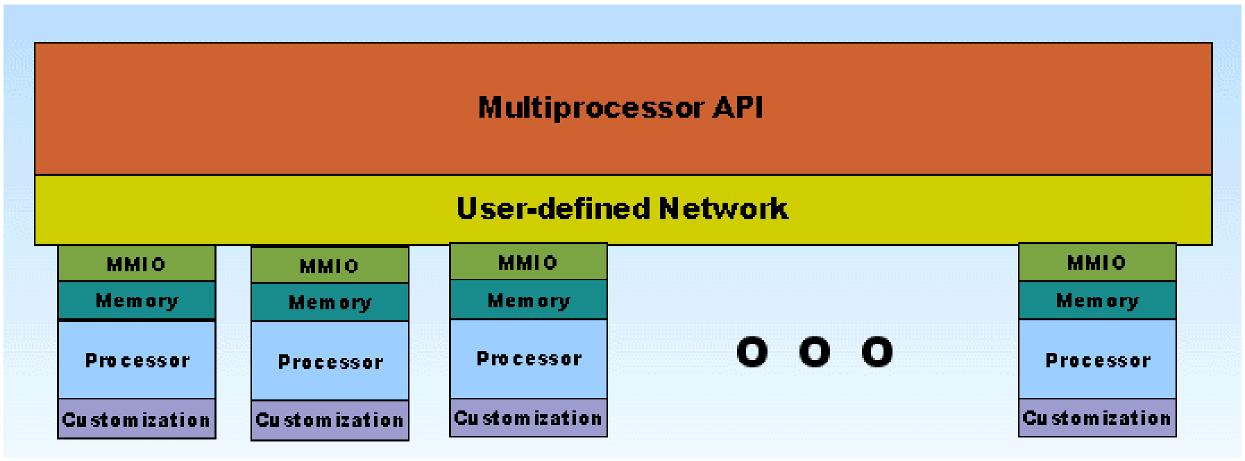


Figure 1: The Cmpware multiprocessor structure.

Of note is the interface between the Processor and the Network. A key concept in the default *Cmpware* behavior is that inter-processor communication exists as channels accessed via Memory Mapped IO. What this means is that some memory address which exists outside of the range of normal memory are used to access external resources. In this case the resource is a *Link* object used for communication.

This Memory Mapped IO approach is significant ibecause it permits a simple and well-defined linkage between the processor and the network and between the hardware and the software. A simple address of the communication port is the only information that all components in the system must share. The other advantage of this approach are that it does not break the existing uniprocessor system. The software development tools such as the compiler still function normally. As well, the processor architecture does not require any special modification to support memory mapped IO.

All of this said, this is the default behavior of the system. Like much of the *Cmpware* development environment, more sophisticated interfaces may be put into place. These will, however, tend to be more difficult to implement and will tend to have an impact of other external portions of the system, such as the processor architecure and the associated processor development tools such as compilers, assemblers and linkers.

The Network Definition

The basis of all network interconnection models in the *Cmpware* system is a Java abstract class called `Network`. This class defines the structure of the Java classes used to build the inter-processor communication networks. This interface is relatively simple, containing the single abstract method, `connectNetwork()`, which must be supplied by the network implementer.

```
public abstract void connectNetwork(Multiprocessor  mp,
String  linkName) throws LinkException;
```

Figure 2: The Link.connectNetwork() method..

Figure 2 shows this method. Note that this method takes in two parameters. The first is called `mp,` which defines a multiprocessor system which has already been allocated. The second parameter is a String indicating the name of link used to construct the network. This link must be a vaild `Link` object implementing the interface in the `com.cmpware.cmp.links` class.

The implementation of the `ConnectNetwork()` method is as simple or as complex as the network it is defining. Figure 3 gives a fragment of code used to connect two processors with a single link. The code is relatively self-explanatory. A new Link object is created, using the `Link.get()` helper method. This link is then added as an input to one processor and as an output to another. This provides a single communication channel between the two processors.

All that is required in the rest of the `connectNetwork()` method is to repeat this process, once for each link in the network. Every effort should be made to make this network conform fully to all of the parameters used to define the multiprocessor. In particular the number of rows and columns in the multiprocessor should be used as well as the name of the link passed in as a parameter. Using hard-wired values may be acceptable, depending on the application, but for a little extra effort, a completely generalized Network definition can be produced which can be used in other projects, or for experimentation with other types of links and other sized arrays.

Appendix A at the end of this document gives the complete source code for the `Network` class used to define the default Torus topology. A Torus is essentially a two-dimensional mesh, with all of the nearest neighbor processors connected to each other. On the edges of the array, the links are wrapped around to the node on the other side. This has the advantage that all of the processors have the same number of useful links and there are no dangling links. Note that the difference between the definition of a simple Mesh and a Torus is relatively trivial. Removal of the modulus (“%”) operation and testing for the ends of the array will convert this Torus definition to that of a Mesh. Other topologies, such as a fully connected array can be done in even less code.

```
/* Get a new Link */
link = Link.get(linkName);

/* Attach the output port */
mp.get(i, j).addOutput(OUT, link);

/* Attach the input port */
mp.get((i+1), j).addInput(IN, link);

/* The address of the input port */
final int IN = 0x80000000;

/* The address of the output port */
final int OUT = 0x80000004;
```

Figure 3: Adding a Link between two processors.

Compiling the Network Model

Once the Network class is implemented in Java, it must be compiled. This can be done with any standard Java compiler, but there are a few things that should be mentioned. First, because this class relies on other classes already in the *Cmpware* system, access to these classes must be provided. The code for these classes can be found in a Java JAR file in the Eclipse plugin.

If you are using Eclipse to develop this model, you should go to the **Project --> Properties** menu item. This will bring up a dialog box. Select the **Java Build Path** item. This will bring up a tabbed window, Select the **Libraries** tab. From there you will click on the **Add External JARs ...** button. This will bring up a dialog box asking for the location of the JAR file. This JAR file can be found under your Eclipse plugins directory, under the most recent plugin from *Cmpware*. The name of the JAR file is **ide.jar**. This should immediately resolve any dependency errors. Other Java development systems should have a similar method for including external JAR files. Consult the documentation for your particular system.

Testing the Model

At this point we have a compiled Java class file that will serve as the network model for the Torus network. But this class file currently exists as a single Java file in its own project. This must now be made visible to the rest of the *Cmpware* code. Fortunately, this is a very simple process. All that is required is that the directory containing the class is specified in the Java ***CLASSPATH*** environment variable. Setting this variable is system dependent, but should be well documented in your system and in the Java documentation.

Perhaps the best way to test the new network model is using the *Cmpware* command line debug monitor. This tool is embedded in the `ide.jar` file in the Eclipse *Cmpware* plugin directory. Use this JAR file as below to bring up the command line debug monitor.

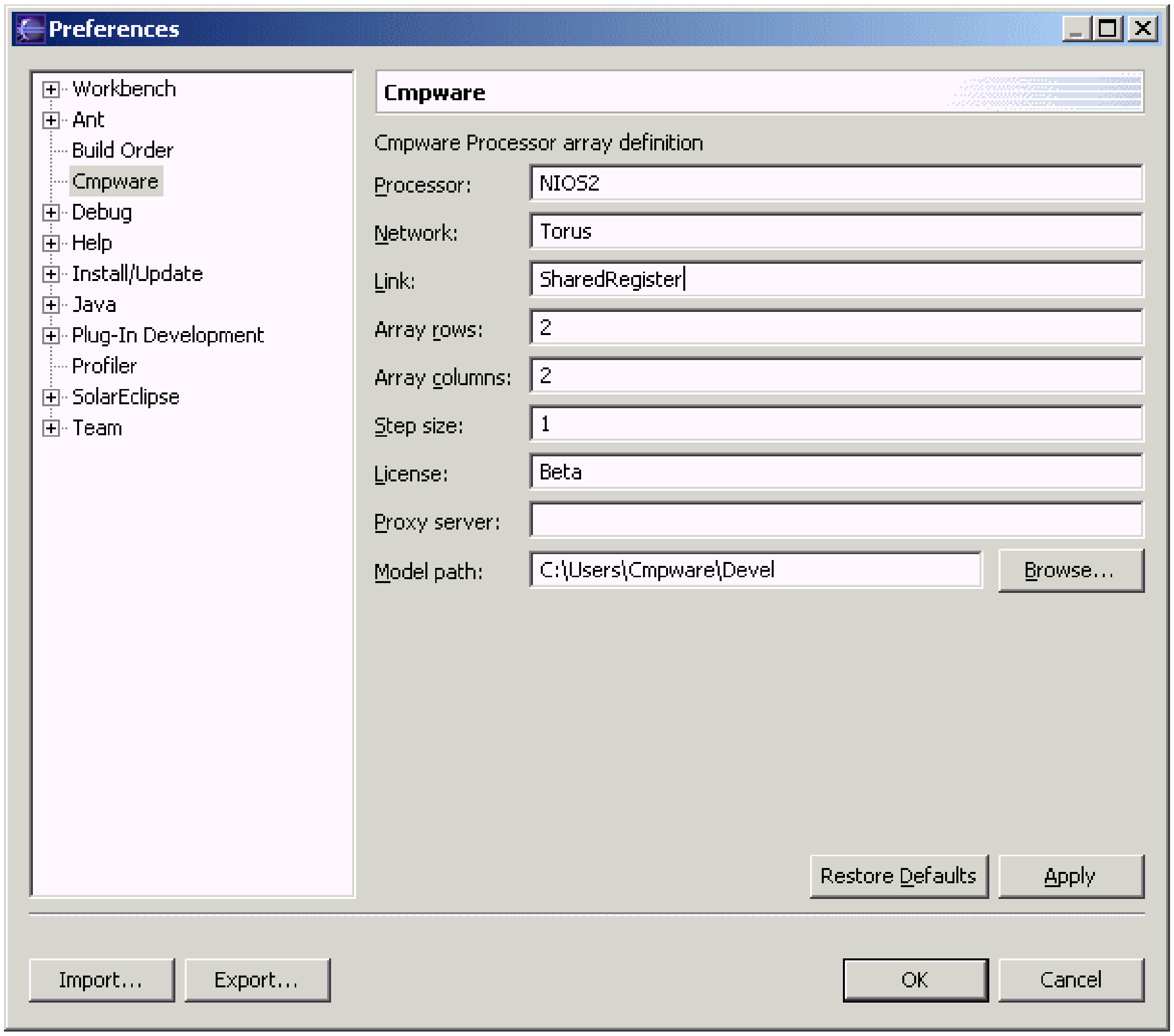
```
$ java -classpath ide.jar com.cmpware.cmp.MpMon
```

This will bring up a prompt that will permit you to interact with the new model. The first step is to allocate a 2 x2 array of *NIOS2* processors with a *Torus* network built using *SharedRegister* links. Then attempt to execute some code in one or more of the processors. Figure 4 below shows an array running on two different processors in the array. This particular test just verifies that all of the new code is visible and working. It is probably best to move to the Eclipse IDE to do detailed testing of the new Network definition.

```
$ java com.cmpware.cmp.MpMon
(null)> a 2 2 NIOS2 Torus SharedRegister
[0,0]NIOS2> t
0000:   3a880100    nop

[0,0]NIOS2> t
0004:   06fe3f00    br  -8

[0,0]NIOS2> t
0000:   3a880100    nop

[0,0]NIOS2> p 1 1
[1,1]NIOS2> t
0004:   06fe3f00    br  -8

[1,1]NIOS2> q
$
```

Figure 4: Testing the Network in MpMon.

Using the Model in the Eclipse IDE

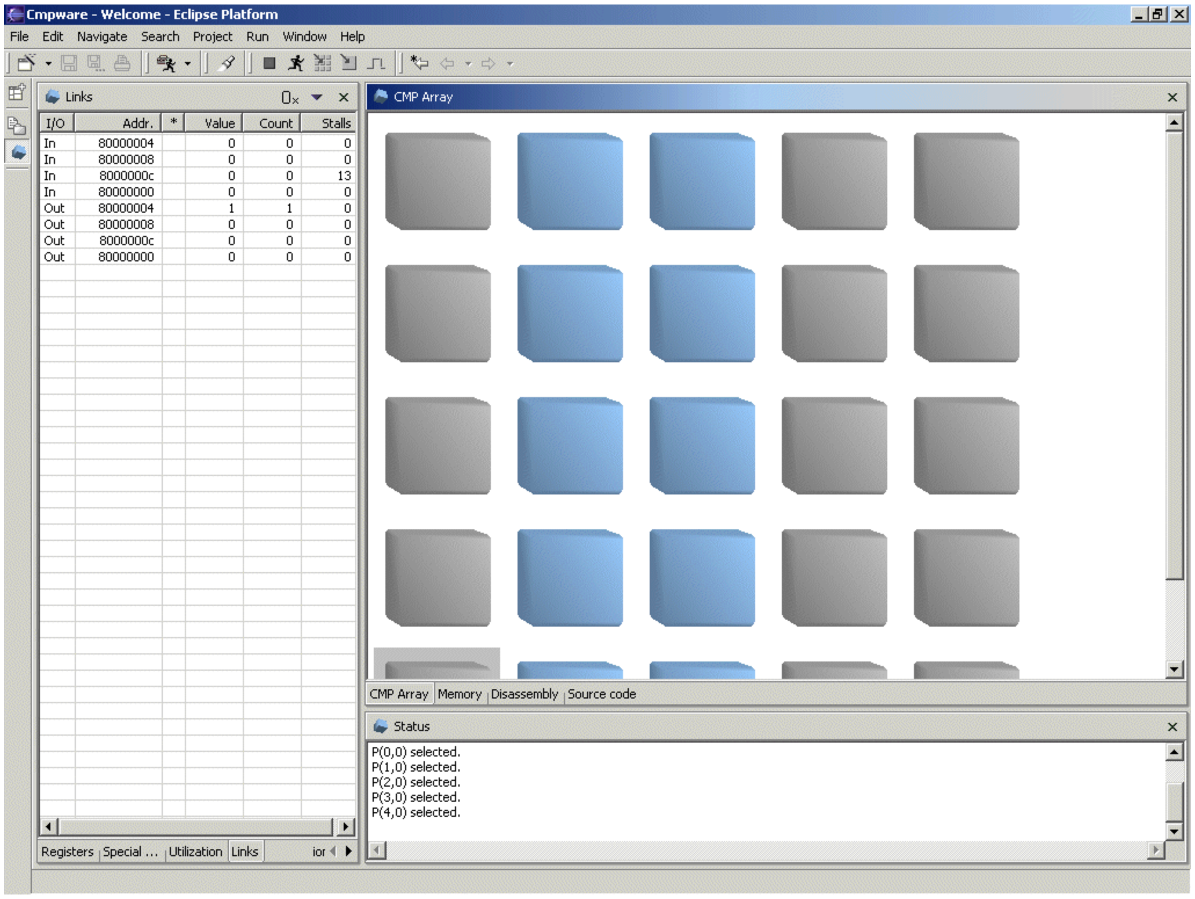


Cmpware Processor array definition

Processor:	NIOS2
Network:	Torus
Link:	SharedRegister
Array rows:	2
Array columns:	2
Step size:	1
License:	Beta
Proxy server:	
Model path:	C:\Users\Cmpware\Devel

Figure 5: The Cmpware Eclipse Preferences.

In general, if the command line interface operates properly, the IDE should also. Once the *Cmpware* Eclipse IDE is brought up, the Torus network must be selected from the **Windows --> Preferences --> Cmpware** preference page. This is done by changing the Network field to **Torus**. In addition, the **Model Path** field must be set to indicate the directory containing the model. This directory should be the one containing the *com/cmpware/cmp/models/* directory tree, which contains the *HardNodeDemo.class* file. Note that the standard *CLASSPATH* search for class files may or may not work depending on your system. It is best to use the **Model Path** preference to indicate the location of these models. Once the Ok button is pressed, a new processor array with a Torus network is allocated and initialized. Figure 6 shows the *Cmpware* IDE using this new Torus network model. The applicatication being in this example sends a single value to the node to the right. This value is passed in a circle through the array. The pattern of each node communicating on a different cycle can be seen in the main array display. In addition, the table of Link activity verifies this expected behavior.


Cmpware - Welcome - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Links
CMP Array
CMP Array Memory Disassembly Source code
Status
Registers Special ... Utilization Links

I/O	Addr.	*	Value	Count	Stalls
In	80000004		0	0	0
In	80000008		0	0	0
In	8000000c		0	0	13
In	80000000		0	0	0
Out	80000004		1	1	0
Out	80000008		0	0	0
Out	8000000c		0	0	0
Out	80000000		0	0	0

```
P(0,0) selected.
P(1,0) selected.
P(2,0) selected.
P(3,0) selected.
P(4,0) selected.
```

Figure 6: The Torus network in the Eclipse IDE.

System Issues and Interconnection Networks

Using a simple, regular network in a design is often the best overall solution. It simplifies the hardware design and provides the programmer with a uniform view of the architecture, which should also simplify the software development. In addition, the very high bandwidth of the interconnections on a single-chip device makes many of the issues concerning interconnection network design obsolete. It is unlikely that a single processing element can saturate the bandwidth available to a single dedicated communication channel, much less to several channels.

However, the full architectural design space should be explored. One option is to provide a very rich interconnection fabric. While the utilization of the links will be low, they tend to be relatively inexpensive hardware resources. Even providing a fully-connected network, which directly connects every processor to every other processor in the array, may be an acceptable solution for smaller arrays. And because of the memory mapped approach used by *Cmpware*, such large or rich networks will not have an appreciable impact on simulation performance.

Also, more irregular Networks may be useful. This is likely to be of interest in more special-purpose architecutures targetting a single application. But such a design may be the simplest path to a functional system. Such an ad-hoc or irregular interconnection network is not difficult to describe in the *Cmpware* system, but the complexity will be proportional to the complexity of the network. And while this approach may complicate the software portion of the design in some cases, it may in fact simplify the software in other cases. It may be worth the effort to explore the possibilities of various interconnection schemes, especially since it is so easy to do in the *Cmpware* environment.

Conclusions

The document has described the modeling of inter-processor communication networks in the *Cmpware* environment. The process is relatively simple and can quickly produce high-quality network simulation models with little effort. It is estimated that a network simulation model for a regular network can be engineered in just a few minutes. Once implemented, such network models can be used with a variety of processors and links to create and explore new architectures and applications with the *CmpwareConfigurable Multiprocessor Development Kit*.

Appendix A: The Torus.java Source Code

```
package com.cmpware.cmp.networks;

import  com.cmpware.cmp.Multiprocessor;
import  com.cmpware.cmp.Network;
import  com.cmpware.cmp.LinkException;
import  com.cmpware.cmp.Link;


/**
**  This implements a 2D Torus (a 2D grid with the ends
**  connected around -- a doughnut) network.  It is
**  implemented using Shared Registers.
**
**  <p>
**  Copyright (c) 2004 Cmpware, Inc.  All Rights Reserved.
**  <p>
**
**  @author SAG
*/

public class Torus extends Network {


/*
**  (non-Javadoc)
**  @see com.cmpware.cmp.INetwork#connectNetwork()
*/

public void connectNetwork(Multiprocessor mp, String linkName)
   throws LinkException {
   int  i;
   int  j;
   int  rows;
   int  cols;
   Link  link;

   cols = mp.getCols();
   rows = mp.getRows();
   for (i=0; i<rows; i++)
```

```
      for (j=0; j<cols; j++) {

         /* North output */
         link = Link.get(linkName);
         mp.get(i, j).addOutput(NORTH, link);
         mp.get(((i+1)%rows), j).addInput(SOUTH, link);

         /* East output */
         link = Link.get(linkName);
         mp.get(i, j).addOutput(EAST, link);
         mp.get(i, ((j+1)%cols)).addInput(WEST, link);

         /* South output */
         link = Link.get(linkName);
         mp.get(i, j).addOutput(SOUTH, link);
         mp.get(((i+rows-1)%rows), j).addInput(NORTH, link);

         /* West output */
         link = Link.get(linkName);
         mp.get(i, j).addOutput(WEST, link);
         mp.get(i, ((j+cols-1)%cols)).addInput(EAST, link);

         }  /* end for(j) */

   }  /* end connectNetwork() */


/** The start address of the IO registers */
public static int IO_ANCHOR = 0x80000000;

/** The address of the 'north' IO register */
public static int NORTH = (IO_ANCHOR + 0);

/** The address of the 'north' IO register */
public static int EAST = (IO_ANCHOR + 4);

/** The address of the 'north' IO register */
public static int SOUTH = (IO_ANCHOR + 8);

/** The address of the 'north' IO register */
public static int WEST = (IO_ANCHOR + 12);

}  /* end class Torus */
```