Building a Cmpware Link 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* defines an extensible model for the network itself, which defines the topology of the interconnection. Beneath this is the physical implementation of this topology, or the *Links*. These links are the individual communication components used to build the network.

The *Cmpware CMP-DK* supplies a set of basic *Link* models. These currently include a *Shared Register* and a *FIFO*. These links are point to point and are used by the Network model to construct the processor to processor interconnection network.

This document describes the *Link* model and its implementation. Like the other models in the *Cmpware* system, the *Link* model provides the default capability. It may be extended and enhanced to suit the requirements of the particular design.

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 system 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 logic, 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. The network 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 *Link* model. Reference will be made to both the *Processor* and *Network* models, but these are discussed in more detail elsewhere.

The Link 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.

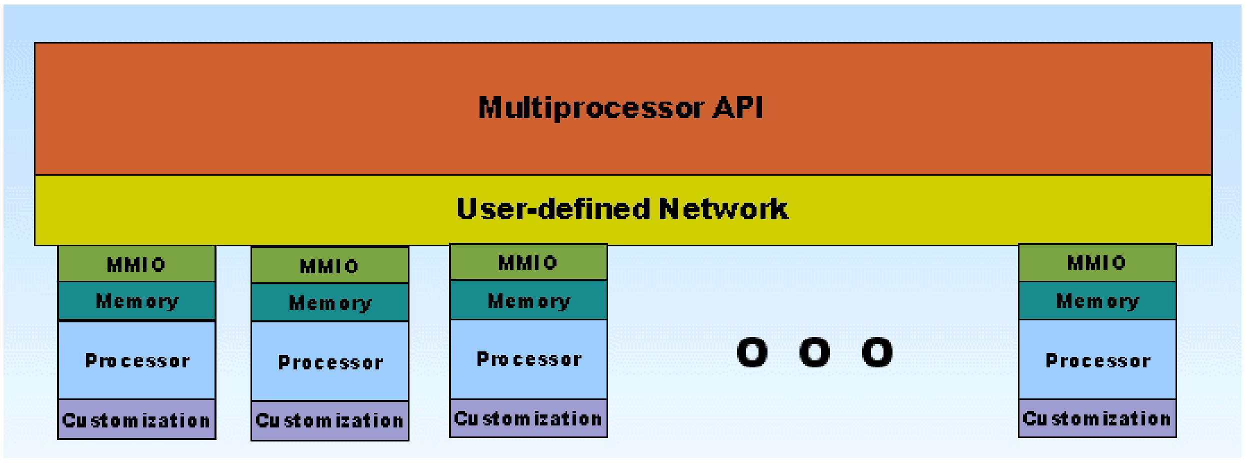


Figure 1: The Cmpware multiprocessor structure.

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 addresses which exists outside of the range of normal processor 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 because it permits a simple and well-defined linkage between the processor and the network and between the hardware and the software. The memory 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 Link Definition

The basis of all network interconnection models in the *Cmpware* system is a Java abstract class called `Link`. This class defines the structure of the Java classes used to build the communication channels used by the inter-processor communication networks. The *Link* class implements two interfaces, the `MemoryMappedIOReader` and the `MemoryMappedIOWriter` These two interfaces are used to seperate the input and output port interfaces of the *Link* class. This permits portions of the multiprocessor which are only concerned with the input port to have access to that port and portions of the multiprocessor which are only concerned with the output port to have access to that port. This simplifies the interface and also makes the implementation and use less error-prone. Figure 2 below shows the `MemoryMappedIOReader` interface. The individual methods are discussed in more detail in the following paragraphs.

```
public int read()  throws MemoryMappedIOException;

public void readCommit();

public void setReadAddress(int  addr);

public int getReadAddress();

public int getReadCount();

public int getReadStallCount();

public boolean isReadable();

public int getValue();
```

Figure 2: The MemoryMappedIOReader interface.

`read()`: The `read()` method is used to return the currently available data from the read port of the *Link*. This method may also throw a `MemoryMappedIOException`. This exception can be thrown for a variety of reasons, but it is primarily used to control the synchronization between communicating processors. By default, the *Link* object functions as a fully synchronized communication resource. This means that data is first written to the link by a processing node, then later read by another processing node. This strict sequence of operations is supported by the *Link* interface. This permits synchronized communication between processors without data being lost. Data may be lost in one of two cases. The first is when data is read before anything was written (overread). This results in a previously read piece of data being re-read, or perhaps an uninitialized value piece of data being read. The second case is data being written before it was read (overwrite). This results in data being overwritten and destroyed before it has had a chance to be read. Both of these cases result in incorrect values being communicated. The `read()` method supports a `MemoryMappedIOException` to be thrown in either of these cases. Inside of the processor models, these exceptions are used to stall the processor until the link is available for communication.

`readCommit():` The next method is the `readCommit()`. This method is perhaps the most complicated of the class. Because *Cmpware* performs a simulation is of a multiprocessor running on a uniprocesson platform, some allowance must be made for proper sequencing of events. This is perhaps a subtle point, but an important one.

In order for the simulation to run accurately, each processor must execute each cycle in unison, then respond to communication events from *Links*. If each processor performs its communication over its links during the processor simulation, data may be communicated across processors within the same simulation cycle.

For instance: suppose Processor A is sending data to Processor B across a Link. The simultator will simulate Processor A for cycle N, which sends its data to Processor B. Now Processor B is simulated for Cycle N, and it sees the new data from Processor A. But Processor B was not supposed to see this data from Processor A until Cycle N+1! The serial nature of the simulation has hit a problem in dealing with the parallel nature of the architecture.

This is a well-known and well-understood problem in simulation, and there are many techniques for handling this. In the *Cmpware* system, we have opted to expose this to the model builders in order to provide a higher-performance solution. The basic solution is to have a commit scheme. This means that the actual update of the state of the *Link* is not changed in the read (or write) method, but is instead done later in the `commit()` method. This causes all communication to happen after all processor nodes have been simulated. This is handled transparently by the *Cmpware* simulator and done at the proper time to guarantee correct ordering and simulation of the multiprocessor.

For example, if the *Link* implementation is a FIFO, the `read()` and `write()` methods will return their correct values or throw their exceptions as expected. But the `commit()` method will be used update any necessary pointers and flags that provide the control. *Appendix A* gives the full source code listing for the `FIFO` link. It is a good guide for illustrating the functionality of the *Link* interface and the behavior of the `commit()` method..

`setReadAddress():` The `setReadAddress()` method is not crucial to the correct functioning of the link. It simply sets the address that the reading processor uses to access this port. Having this address as part of the *Link* object just makes it easier other portions of the software to display link status.

`getReadAddress():` The `getReadAddress()` method returns the address value set in the setReadAddress() method.

`getReadCount():` The `getReadCount()` method returns an internal counter which the link maintains to keep track of the number of reads performed on this link. This is not crucial to the functionality of the link, but it is simple to implement and is highly recommended for debug and performance analysis purposes.

`getStallCount():` The `GetStallCount()` method returns an internal counter which the link maintains to keep track of the number of stalls performed on this link. This is essentially the number of times an exception was thrown on a read indicating that the link did not have data available for read. Like the Read Count, this is not crucial to the functionality of the link, but it is simple to implement and is highly recommended for debug and performance analysis purposes.

`isReadable():` `isReadable()` is the method which determines if the link has data available for read. This method may be used by the `read()` method to determine if an exception will be thrown on a read. This method may also be used for polling schemes for reading ports.

`getValue():` The `getValue()` method returns the current output value of the port. This is used primarily for status display and should not be used as a substitute for the `read()` method.

This is the implementation of the read port of a Link. While there are eight methods, many just return a simple local value and others should take only a few simple lines of code.

The other side of the link, the write port, has a very similar interface. Figure 3 below shows the `MemoryMappedIOWriter` interface. The individual methods are discussed in more detail in the following paragraphs. The implementation of the `MemoryMappedIOWriter` is symmetrical to the implementation of the `MemoryMappedIOReader` and provides similar access to the write port of the Link.

```
public int write() throws MemoryMappedIOException;

public void writeCommit();

public void setWriteAddress(int  addr);

public int getWriteddress();

public int getWriteCount();

public int getWriteStallCount();

public boolean isWriteable();

public int getValue();
```

Figure 3: The MemoryMappedIOWriter interface.

`write()`: The write`()` method is used to send data to the write port of the *Link*. This method may also throw a `MemoryMappedIOException`. This exception can be thrown for a variety of reasons, but it is primarily used to control the synchronization between communicating processors. Like in the `read()` method, this exception is thrown when a write is attempted and the *Link* is unable to accept the data. Typically this occurs in situations such as a full FIFO. This exception is used by the processor models to stall the processor to re-try the write() on the next cycle.

`writeCommit()`: The next method is the `writeCommit()`. Like `readCommit()`, this method is used to provide the proper simulation sequencing when processors communicate. Esentially, the `write()` method is used to send a value to the input port and perhaps throw an exception if necessary. The `commit()` method is used later in the simulation cycle to update the state of the Link. This typically means updating any control flags or pointers. For more information on this subject, see the paragraph in this document describing the `read()` method, and the source code listing of the *FIFO* link in *Appendix A*.

`setWriteAddress()`: The `setWriteAddress()` method is not crucial to the correct functioning of the link. It simply sets the address that the writing processor uses to access this port. Having this address as part of the *Link* object just makes it easier for other parts of the software to display link status.

`getWriteAddress()`: The `getWriteAddress()` method returns the address value set in the `setWriteAddress()` method.

`getWriteCount():` The `getWriteCount()` method returns an internal counter which the link maintains to keep track of the number of writes performed on this link. This is not crucial to the functionality of the link, but it is simple to implement and is highly recommended for debug and performance analysis purposes.

`getWriteStallCount():` The `GetWriteStallCount()` method returns an internal counter which the link maintains to keep track of the number of write stalls performed on this link. This is essentially the number of times an exception was thrown on a write indicating that the link could not accept data. Like the Write Count, this is not crucial to the functionality of the link, but it is simple to implement and is highly recommended for debug and performance analysis purposes.

`isWriteable():` `isWriteable()` is the method which determines if the link can accept data for write. This method may be used by the `write()` method to determine if an exception will be thrown on a write. This method may be used for polling schemes for writing ports.

`getValue():` The `getValue()` method returns the current output value of the port. This is used primarily for status display and should not be used as a substitute for the `read()` method.

This is the implementation of the write port of a Link. While there are eight methods, many just return a simple local value and others should take only a few simple lines of code.

Compiling the Link Model

Once the Link 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 Link model for the *FIFO* Link. 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.

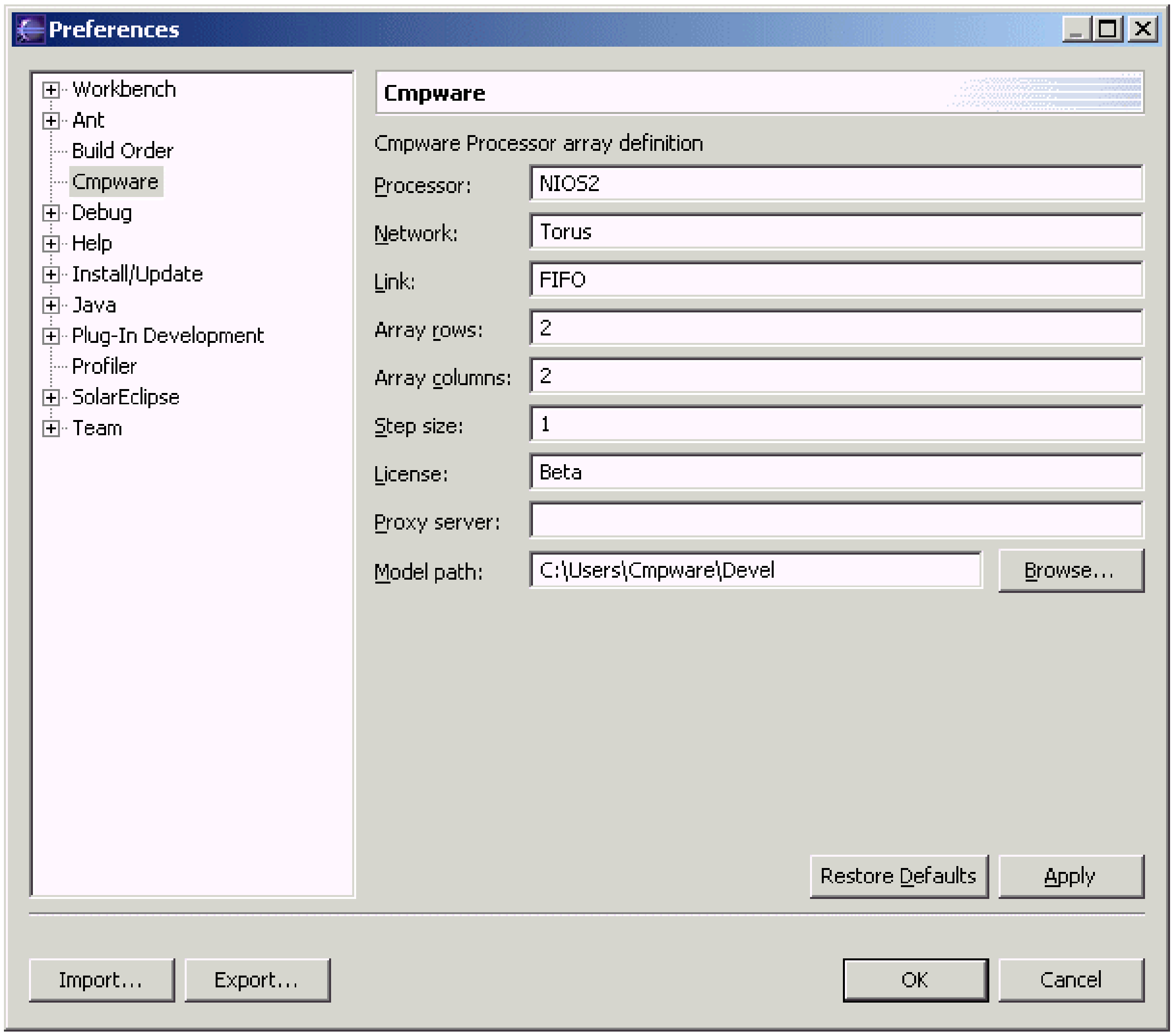
```
$ java com.cmpware.cmp.MpMon
(null)> a 2 2 NIOS2 Torus FIFO
[0,0]NIOS2> t
0000:   3a880100    nop

[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.

Perhaps the best way to test the new link 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 *FIFO* 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 Link definition.

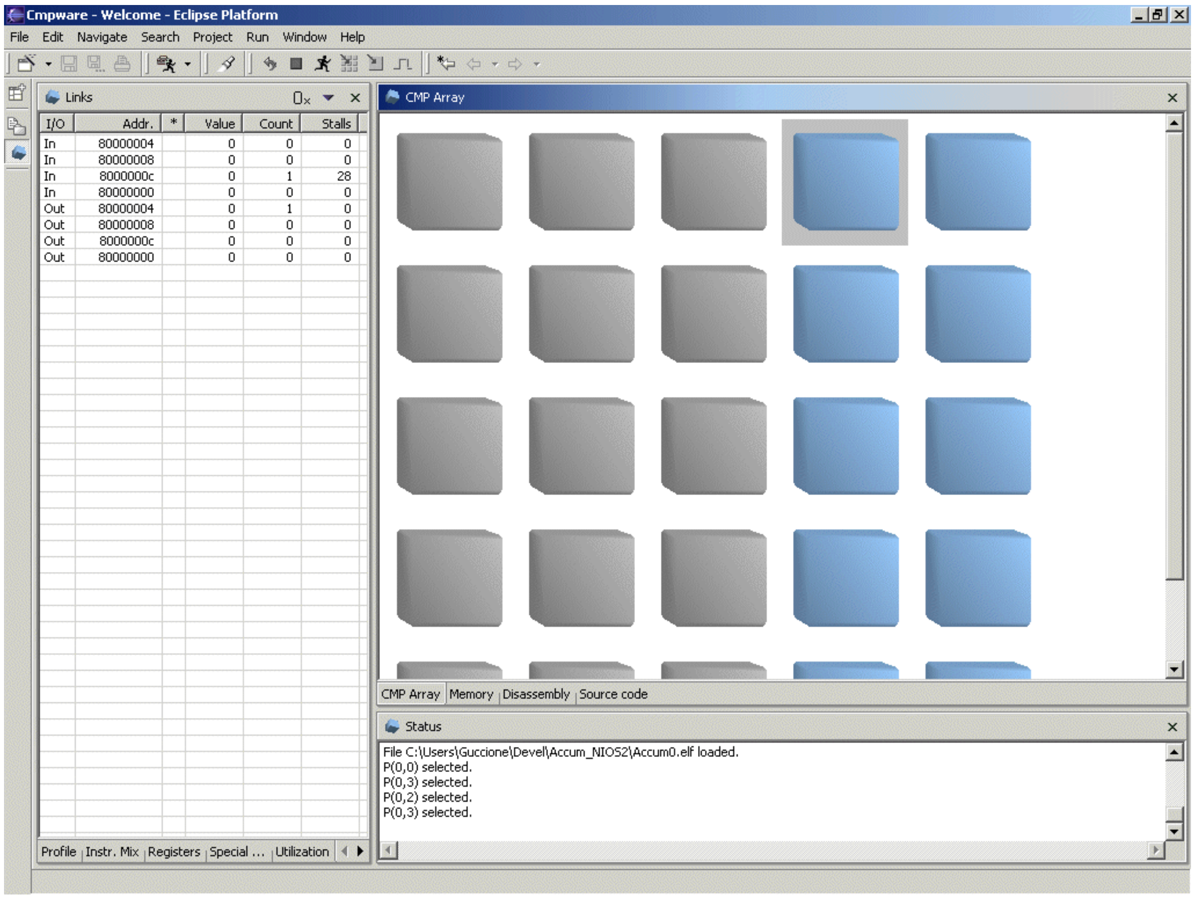


Cmpware Processor array definition

Processor:	NIOS2
Network:	Torus
Link:	FIFO
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.

Using the Model in the Eclipse IDE



I/O	Addr.	*	Value	Count	Stalls
In	80000004		0	0	0
In	80000008		0	0	0
In	8000000c		0	1	28
In	80000000		0	0	0
Out	80000004		0	1	0
Out	80000008		0	0	0
Out	8000000c		0	0	0
Out	80000000		0	0	0

```
File C:\Users\Guccione\Devel\Accum_NIOS2\Accum0.elf loaded.
P(0,0) selected.
P(0,3) selected.
P(0,2) selected.
P(0,3) selected.
```

Figure 6: The FIFO Link in the Eclipse IDE.

In general, if the command line interface operates properly, the IDE should also. Once the *Cmpware* Eclipse IDE is brought up, the FIFO link must be selected from the **Windows --> Preferences --> Cmpware** preference page. This is done by changing the Link field to **FIFO**. 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 Link network model. The applicationbeing 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.

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 may 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 the links used by the inter-processor communication networks in the *Cmpware* environment. The process is relatively simple and can quickly produce high-quality link and network simulation models with little effort. It is estimated that a link and network simulation model for a regular network can be engineered in just a few minutes. Once implemented, such 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 FIFO.java Source Code

```
package com.cmpware.cmp.links;

import com.cmpware.cmp.MemoryMappedIOException;

import com.cmpware.cmp.Link;


/**
**  This class implements a synchronous FIFO.  This is
**  used as a link component to build inter-processor
**  communication networks.  The input port of the FIFO
**  is typically mapped to a memory mapped IO address
**  on one processor and the output port to a memory
**  mapped IO address on another processor.  This creates
**  a communication link between the processors.
**
**  Note that the internal implementation of this link
**  object requires some care to support the multiprocessor
**  simulation environment.  All state update should be
**  done at commit time.
**
**  <p>
**  Copyright (c) 2004 Cmpware, Inc.  All Rights Reserved.
**  <p>
**
**  @author SAG
*/


public class FIFO extends Link {


/** Copyright string */
public final static String copyright =
   "Copyright (c) 2004 Cmpware, Inc.  All Rights Reserved.";


/**
**  This constructor allocates a FIFO using the default
**  FIFO size.
```

```
*/

public FIFO() {
   fifo = new int[DEFAULT_SIZE+1];
   }  /* end FIFO() */


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.Link#setBufferSize(int)
*/

public void setBufferSize(int size) {
   fifo = new int[size+1];
   head = 0;
   tail = 0;
   }  /* end setBufferSize() */


/*
**  (non-Javadoc)
**  @see com.cmpware.cmp.MemoryMappedIOWriter#write(int)
*/

public void write(int val) throws MemoryMappedIOException {

   /* Stall the writing processor on overwrite */
   if (isFull() == true) {
      writeStallCount++;
      throw mmioe;
      }

   /* Write the data*/
   /*(but don't update the pointers until commit) */
   fifo[tail] = val;
   writtenFlag = true;

   }  /* end write() */


/*
**  (non-Javadoc)
**  @see com.cmpware.cmp.MemoryMappedIOReader#read()
*/
```

```
public int read() throws MemoryMappedIOException {

   /* Check for underread */
   if (isEmpty() == true) {
      readStallCount++;
      throw mmioe;
      }

   /* Read the data */
   /* (but don't update the pointers until commit) */
   readFlag = true;
   return (fifo[head]);

   }  /* end read() */


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOWriter#writeCommit()
*/

public void writeCommit() {

   if (writtenFlag == true) {
      tail = ((tail+1)%fifo.length);
      writtenFlag = false;
      writeCount++;
      }

   }  /* end writeCommit() */


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOReader#readCommit()
*/

public void readCommit() {

   if (readFlag == true) {
      head = ((head+1)%fifo.length);
      readCount++;
      readFlag = false;
      }

   }  /* end readCommit() */


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOReader#setReadAddress(int)
*/

public void setReadAddress(int  addr) {
   readAddr = addr;
   }  /* end setReadAddress() */


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOReader#getReadAddress()
*/

public int getReadAddress() {return (readAddr);}


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOReader#getReadCount()
*/

public int getReadCount() {return (readCount);}


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOReader#getReadStallCount()
*/

public int getReadStallCount() {return (readStallCount);}


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOWriter#setWriteAddress(int)
*/
```

```
public void setWriteAddress(int  addr) {
   writeAddr = addr;
   }  /* end setWriteAddress() */


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOWriter#getWriteAddress()
*/

public int getWriteAddress() {return (writeAddr);}


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOWriter#getWriteCount()
*/

public int getWriteCount() {return (writeCount);}


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOWriter#getWriteStallCount()
*/

public int getWriteStallCount() {return (writeStallCount);}


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOWriter#getValue()
*/

public int getValue() {return(fifo[head]);}


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOReader#isReadable()
*/

public boolean isReadable() {
   return (!isEmpty());
   }  /* end isReadable() */


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.MemoryMappedIOWriter#isWriteable()
*/

public boolean isWriteable() {
   return (!isFull());
   }  /* end isReadable() */

/*
** Privates
*/

/**
**  This method returns a true if the FIFO is
**  empty and a false otherwise.
*/

private boolean isEmpty() {
   if (head == tail)
      return (true);
   else
      return (false);
   }  /* end isEmpty() */


/**
**  This method returns a true if the FIFO is
**  full and a false otherwise.
*/

private boolean isFull() {
   if (((tail+1)%fifo.length) == head)
      return (true);
   else
      return (false);
   }  /* end isFull() */



/** The default FIFO size */
```

```
public static final int DEFAULT_SIZE = 8;

/** The FIFO data */
private int  fifo[];

/** The FIFO 'head' pointer */
private int head = 0;

/** The FIFO 'tail' pointer */
private int tail = 0;

/** The read address (for information only) */
private int  readAddr = 0;

/** The write address (for information only) */
private int  writeAddr = 0;

/** The number of reads */
private int  readCount = 0;

/** The number of writes */
private int  writeCount = 0;

/** The number of read stalls */
private int  readStallCount = 0;

/** The number of write stalls */
private int  writeStallCount = 0;

/** This flag indicates that a write
**  was performed on this cycle */
private boolean writtenFlag = false;

/** This flag indicates that a read
**  was performed on this cycle */
private boolean readFlag = false;

/** A Memory Mapped IO Execption
** (so we don't have to keep re-allocating them) */
private MemoryMappedIOException mmioe = new
MemoryMappedIOException();

}  /* end class FIFO */
```