Building a HardNode Model

Cmpware, Inc.

Introduction

The *Cmpware Configurable Multiprocessor Development Kit (CMP-DK)* is based around fast simulation models for processors. These models may be standard architectures from traditional microprocessor vendors, or they may be custom processors developed for a specific application. Or, in many cases these processors may be hard-wired logic used to efficiently implement some function. Like other extensible models in the *Cmpware* system, implementing *HardNodes* and interfacing them to other processors is a relatively simple task. This treatment of custom logic blocks as processors has a variety of advantages, including rapid simulation. This document is provided to describe the process of building *HardNode* models.

HardNodes

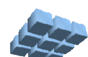
The power and flexibility of the *Cmpware* system comes from its ability to abstract large, complex hardware components and simulate them efficiently. Typically these components are traditional instruction set processors. In modern System On Chip (SoC) design, there is typically some combination of processors and fixed hardware components. There are many reasons for using fixed hardware instead of programmable instruction set processors. Among the most common reasons are:

- **Performance**: Sometimes it is not possible to perform all of the processing necessary for an application, even with a relatively large number of processors. In many cases, this processing occurs on high data rate inputs or outputs. Here raw input / output speed demmands that custom logic be used.

- **Power**: While it may be possible to perform all of the necessary processing using programmable processors, there may be cases where a small amount of custom logic can dramatically reduce the power consumption as compared to a multiprocessor approach. This power savings is often a result of allowing a lower system clock speed.

- **Available Intellectual Property**: In other cases, there may be some existing,

Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.



verified custom logic core already in existence from a previous effort. Rather than porting this functionality to one or more programmable processors, it may be simpler to use this existing circuitry.

There may be other reasons for using custom logic in an SoC design, but these are some of the more compelling ones. In order to support the addition of such custom logic in the *Cmpware* environment, an extensible modeling class called `HardNode` is defined.

The HardNode Class

The basis of all processor models in the *Cmpware* system is a Java class called `Processor`. This class contains much of the machinery necessary for modeling a standard instruction set processor. This involves such generic functionality as managing instruction fetch, branching with delay slots, and returning various pieces of information used by the command line and *Eclipse* interfaces.

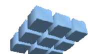
However, much of this does not directly relate to supporting simultaion models for hard wired logic. Fortunately, the `Processor` interface is generic enough to support any sort of processing engine, even a hard wired custom logic block. All that is required is to ignore much of the functionality of the `Processor` class and perhaps be aware of some interfacing issues which are usually hidden by the `Processor` class.

This is exactly what the `HardNode` class does. The `HardNode` class extends the `Processor` class, filling in some default values for the abstract methods which are not useful when defining a hard wired logic block. In fact, all that remains to be implemented from the original `Processor` class is the `execute()` method as shown in Figure 1.

```
public abstract void execute(int instr)
   throws MemoryAccessException,
          IllegalRegisterException;
```

Figure 1: The `HardNode` class abstract method.

To define the behavior of the custom logic block, all that is required is that the `execute()` method be supplied to provide the appropriate behavior. While this method takes a single input parameter, it is typically ignored. Similarly, there is a memory and register exception which may be thrown by the method. While these are

Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.


also typically ignored, there may be cases where memory or registers are used by the custom logic and it may be useful to throw such exceptions.

A HardNode Implementation

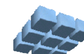
Figure 2 gives a simple *HardNode* implementation. While the calculation being performed by the HardNode is usually fairly straight forward to implement, some case should be taken when performing the processor communication. At the core of the issue is that the *HardNode* will be performing its operations in parallel. This must be taken into account in the simulation model, primarily in the way in which it communicates with other processor nodes.

```
public void execute(int instr)
   throws MemoryAccessException,
          IllegalRegisterException {

   /* Try to read a value from the east MMIO port */
   try {
      i = read32(east);
   } catch (MemoryMappedIOException sre) {
      ;  // Do nothing
   }

   i = i + 1;
   r[0] = i;

   /* Try to write a value out to the west MMIO port */
   try {
       write32(west, i);
   } catch (MemoryMappedIOException sre) {
      ;  // Do nothing
   }

   }  /* end execute() */
```

Figure 2: Loading the hand-compiled Simple code at reset.

The code in Figure 2 performs a very simple function. It reads a value from the “east” Memory Mapped IO port, increments it, and sends it to the “west” Memory Mapped IO port. The data is also written to general purpose register `r[0]`. The only reason for

Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.



this is so that it is visible in the *Eclipse* display. One of the defaults in the `HardNode` class is that a single register, `r[0]`, is available. Note that it is possible to define more registers, but we will make use of this default. Perhaps the only parts of the simulation model which may require some explanation are the catching of the two Memory Mapped IO exceptions.

Part of the default `Link` model used by *Cmpware* is a handshaking protocol. This is a semaphore bit used to tell the processor if the particular port is ready to send or receive data. In the case of this simple node, as long as it is communicating with other `HardNodeDemo` processors, the communication links will always be available to send and receive data.

In other cases, the `HardNodeDemo` may be connected to other processing nodes which may at some points not be able to receive data on every cycle. In these cases, the `HardWiredNode` will attempt to write data to a port which is currently unavailable, and will generate a `MemoryMappedIOException`. In the standard `Processor` model, this is used to internally generate a processor stall, which retries the communication until it succeeds.

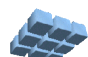
In this *HardNode* model, the hardware just keeps on writing, whether the port is ready to accept the data or not. This is, in fact, the way hardware often works. It produces a result each cycle, and the receiver had better be able to accept it. But this model could just as easily be modified to handle cases where the ports may be busy. This would just involve additional code in the exception handlers. Of course, the ultimate decision on how the communication is to be modeled depends on the hardware itself. If it uses this type of flow control, then it should be included in the model. Otherwise, it is safe to ignore these exceptions.

Testing the New HardNode Model

At this point we have a compiled Java class file that will serve as the *HardNode* model. 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 *HardNode* 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.

Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.



$ 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 3 x 3 array of *HardNodeDemo* processors. Then attempt to step the processors. Figure 4 below shows a 3 x 3 array of processors stepped for five cycles. A look at the registers shows the expected values, five. Additionally, the ports show the commiunication network operating as expected. One input and one output port are used for a total of five cycles each. This particular test verifies that the new model is visible and working. Moving to the Eclipse IDE will verify that the model is fully functional in this environment.

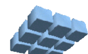
```
$ java -classpath ide.jar com.cmpware.cmp.MpMon
(null)> a 3 3 HardNodeDemo
[0,0]HardNodeDemo> s 5
[0,0]HardNodeDemo> r
none:00000005

none:00000000

[0,0]HardNodeDemo> ports
Input ports:
80000004  *00000005  (2/3)
80000008   00000000  (0/0)
8000000c   00000000  (0/0)
80000000   00000000  (0/0)
Output ports:
80000004  *00000000  (0/0)
80000008  *00000000  (0/0)
8000000c   00000005  (3/2)
80000000  *00000000  (0/0)
[0,0]HardNodeDemo> q
[0,0]HardNodeDemo> q
$
```

Figure 3: Simulating the HardWiredDemo.

Using the Model 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 HardNodeDemo processor must be selected from the **Windows --> Preferences --> Cmpware** preference page. This is

Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.



done by changing the Processor field to **HardNodeDemo**. 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 of *HardNodeDemo* processors is allocated and initialized.

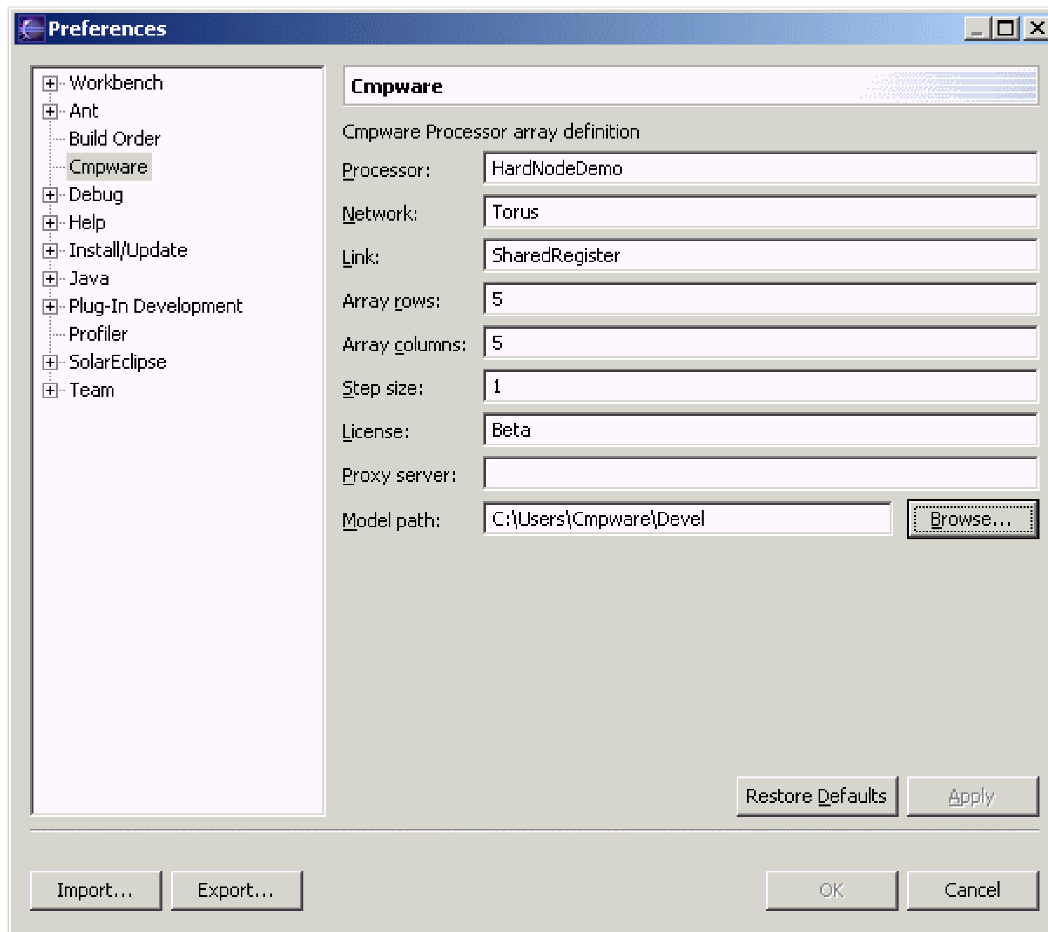
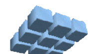


Figure 4: Setting up the HardNodeDemo.

Figure 5 shows the *Cmpware* IDE using this new `HardNodeDemo` model. This array of hard wired processors just takes values in form the communication link to the left, increments the value read, then writes the value out the port to the right. The pattern of each node communicating on a different cycle can be seen in the main array display. In

Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.



addition, the table of Link activity verifies this expected behavior.

Comments on Using HardWired Nodes

As mentioned earlier, there are many reasons for using hard wired logic in a design. Performance, power and practical considerations all are factors. But one interesting feature of the ease at which the *Cmpware* tools define and integrate such models is that it permits designs to evolve in new directions.

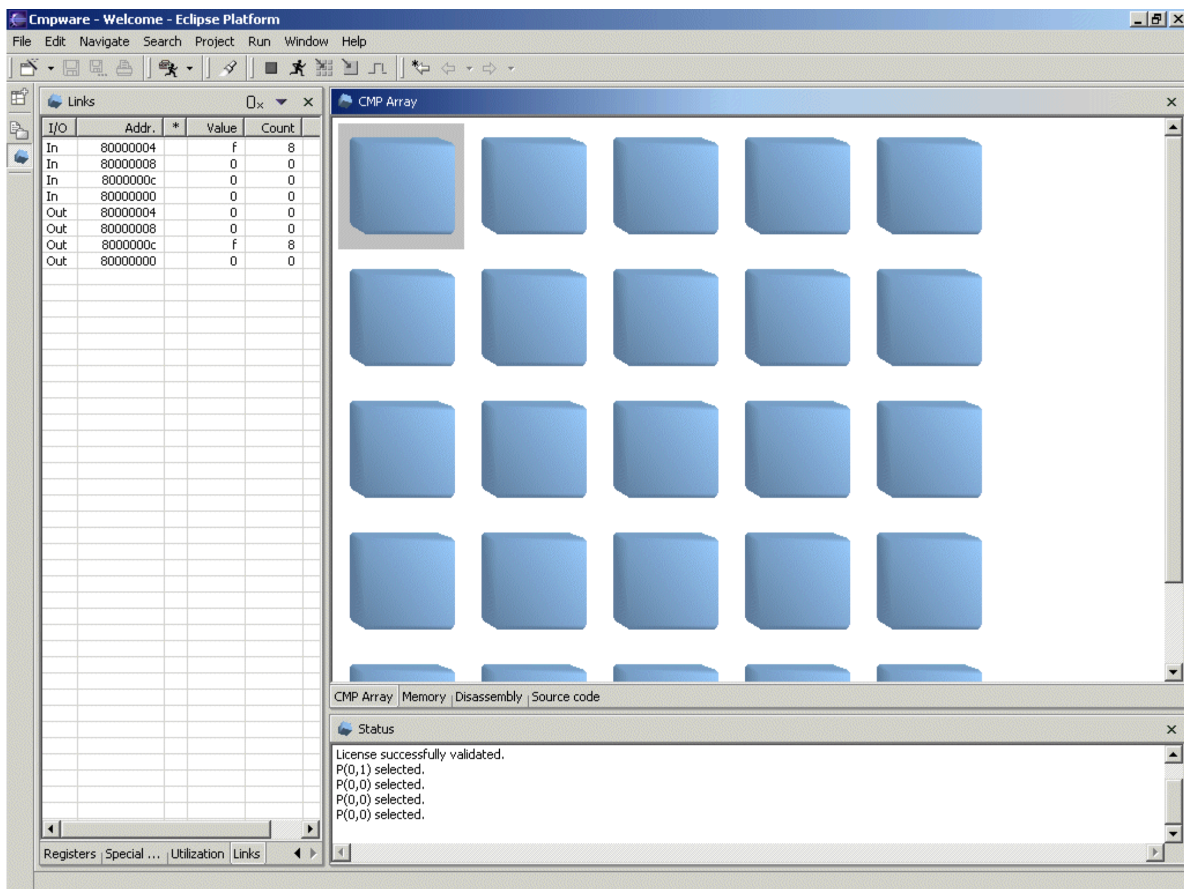
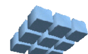


Figure 5: An array of HardWiredDemo processors.

The first approach is the gradual addition of hard wired nodes from an existing array of traditional processors. If done correctly, instruction set processors can be incrementally replaced with hard wired logic with minimal disturbance to the existing system. This can permit a system, for instance, to be rapidly designed and implemented with

Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.



programmable processors for a faster time to market. Then the design can be improved in power, performance and / or other system parameters. This permts designs to evolve incrementally without a complete overhaul of the hardware and software in the system.

A design may also evolve in the opposite direction. An array of various hard wired nodes may be incermentally replaced with instruction set processors as part of the design process. This replaces a fixed platofrm with a programmable one. This may be desirable for a number of reasons. In general, a programmable system may have several advantages over a hard wired design. Among these advantages are:

- Field Repair: Bugs and other degisn errors can be fixed in fielded systems if the error is in software. In hardware, such bugs may either permanently cripple functionality of the system or even make such systems require repalcement.

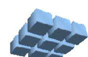
- Upgradeability: Fielded systems can have their functionality modified with a software change. This permits new features to be added. This may extend the lifetime of a product by adding new functionality with software without having to obsolete the hardware.

- Reuse: Fixed logic requires hardware resource to be dedicated for each system function, no matter how frequently or infrequently it is used. A programmable solution can reprogram its processing nodes to perform one function at one instant, and onother later. This can actually dramatically reduce the amount of hardware required in a system, potentially reducing cost and power consumption.

Conclusions

The document has described the modeling of a simple hard wired node in the *Cmpware* system. These nodes permit custom logic and other non-traidtional processing elements to be easily integrated into the *Cmpware* environment. By maintaining compatability with the existing `Processor` interface, the *Cmpware* toolkit is able to easily manipulate and simulate the model as if it were a standard processor. In fact, in the *Cmpware* environment, a `HardNode` is just another processor, and one that is often much simpler than traditional instruction set processing nodes.

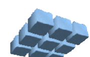
Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.



Appendix A: HardNodeDemo.java

```
package com.cmpware.cmp.models;

import com.cmpware.cmp.HardNode;
import com.cmpware.cmp.MemoryAccessException;
import com.cmpware.cmp.IllegalRegisterException;
import com.cmpware.cmp.MemoryMappedIOException;


/**
**  This gives an example of a HardNode.  It simply
**  takes data from the "east" input, increments it,
**  then sends it to the "west" output.
**
**  <p>
**  Copyright (c) 2004 Cmpware, Inc.  All Rights Reserved.
**  <p>
**
**  @author SAG
*/

public class HardNodeDemo extends HardNode {

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


/*
**  (non-Javadoc)
** @see com.cmpware.cmp.Processor#execute(int)
*/

public void execute(int instr)
   throws MemoryAccessException, IllegalRegisterException {

   /* Try to read a value from the east MMIO port */
   try {
      i = read32(east);
```


Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.



```
    } catch (MemoryMappedIOException sre) {
       ;  // Do nothing
    }

    /* Try to write a value out to the west MMIO port */
    try {
       write32(west, i);
       /* Save it to the r[0] register */
       /* (just so we can see it in the IDE) */
       r[0] = i++;
    } catch (MemoryMappedIOException sre) {
       ;  // Do nothing
    }

    }  /* end execute() */

/** The East MMIO port */
public final static int east = 0x80000004;

/** THe West MMIO port */
public final static int west = 0x8000000c;

/** The value being passed along */
private int  i = 0;

}  /* end class HardNodeDemo() */
```


Copyright © 2004, 2005 Cmpware, Inc. All rights reserved.