Modeling An ALU Array

Steven A. Guccione
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 arbitrary digital logic used to efficiently implement some function. In this report, a simple ALU is defined and put into an existing communication network using the *Cmpware CMP-DK*.

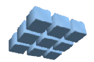
The Modeling Environment

The power and flexibility of the *Cmpware* system comes from its ability to abstract large, complex hardware components and simulate them efficiently. Often these components are traditional instruction set processors. In modern System On Chip (SoC) design, there is usually some combination of processors and fixed hardware components. By treating everything as a '*processor*', the level of abstraction for the simulation model is raised higher than than that other circuit and signal based simulation packages. This permits simpler modeling and faster simulation.

The *Cmpware* multiprocessor simulation models have three components.

- **Processors**: A 2D array of processors is defined by the simulation environment. These may be all of the same type (homogeneous) or of a variety of types (heterogeneous). Homogeneous arrays may be specified with a single parameter in the *Eclipse* Perference Page. A heterogeneous array is defined in a Java class that provides a 2D array of the names of the classes implementing the multiprocessor.

- **Links**: The links are the physical interconnection elements. These may be shared registers, FIFOs, broadcast busses or any other type of communication element that provides point to point or broadcast communication capabilities. These may be

Copyright © 2005 Cmpware, Inc. All rights reserved.


synchronized or unsynchronized. Most link models are fairly simple to implement, but they may be as sophisticated as desired.

- **Network**: The network definition uses links to connect the processors. This network is easily parameterizable and may be based on the array size and the link type used. For instance, changing the **Link** field in the *Eclipse* Preference Page can change a network based on bus interconnect to one based on FIFOs with a single command. As with the other models, the ***Network*** model is as simple or as complex as the underlying design being modeled.

The User Interface

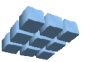
The user interface is designed to tie together the ***Processor***, ***Network*** and ***Link*** models and display a wide variety of 'live' information about the overall system. The display is based on the selection of a single processing element, which provides the data for the displays. Other systems tend to be based on multi-window debuggers originally designed for single processor use. This often results in a large number of confusingly identical windows along high overheads and with low simulation performance. Additionally, such extended uniprocessor tools tend to neglect the communication infrastructure. This can be a very significant part of the system and information about its behavior is often critical to system modeling, as well as software design and debug.

The *Cmpware* approach provides a fast, simple interface that supplies a large amount of information in a standard format. The data presented covers all aspects of the system, from hardware register values to software-defined variables. Individual processors, networks and system-wide information can be quickly and easily accessed. This helps simplify the potentially complex task of analyzing such multiprocessor systems. For more information on the details of the *Cmpware* GUI, see the related documents on this subject.

The ALU Node

The ALU node itself looks more like a '***hardwired***' node than a traditional instruction set processor. The model used, however, is actually the '***Processor***' model. This was done to permit flexibility and for possible future expansion. In particular some memory is allocated and the first word is used as the 'instruction' for configuring the node. This can be seen in the ***Disassembly*** window in the ALU node display. Other approaches could have been taken, but this permits easy loading / reloading of the configuration data and also allows for potential future expanision.

Copyright © 2005 Cmpware, Inc. All rights reserved.



The code to model the ALU node was generated by the *Cmpware* ***ProcGen*** tool. ***ProcGen*** is a standalone application used to take a simple input specification and generate a 'skeleton' for the processor node. **Appendix A** gives the description for the ALU array. This description contains 14 'instructions' or operations, each indicated by a single line of text. This file is then used to generate the ***ALU.java*** file in **Appendix B**. This file contains over 400 lines of text, most of which is comments, whitespace or tables of data. Some editing of this file is required, however, to further define the simulation model. In **Appendix B,** the generated code is indicated by the grey background; the added code is not highlighted.

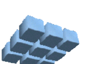
Much of the added code is used to support the 'assembly language' for this node. Rather than generating a full tool to program this simple example, a text file is used to program the ALU array. In this case, the first byte is the opcode, chosen to be an ASCII character which represents the operation ('+' for addition, '*' for multiplication, etc). The next three bytes describe the network interface for the node. This currently assumes a 2D grid with connections to neighbors indicated by 'n', 's', 'e' and 'w', which indicate "north". "south, "east" and "west", respectively. So the instruction "**+sew**" takes inputs from the east and west, adds them, and sends the result to the south. This ability to use a text file to configure a node is relatively easy to implement and saves some effort in producing an external assembler tool.

The Network Model

A network model called ***Torus.java*** is one of the standard network models supplied with the *Cmpware CMP-DK*. It is defined to connect the nodes together and to produce a circuit for simulation. **Appendix C** gives the actual implementation of this model. This implementation provides a relatively flexible network which works for any size array. The *links* used in this node are also configurable. The default given in the *Cmpware* Preferences is a ***SharedRegister***, which is a semaphored single-word communication channel similar to a one word FIFO. This link allows execution to proceed only when data is available. This produces a synchronous dataflow style circuit, which has many advantages in its ability to initialize data and interface to a variety of other nodes. Other links, including asynchronous 'wires' may also be easily subsitiuted, producing a more synchronous, pipelined array. Note that the model for this network is relatively simple, consisting of approximately a dozen fairly repetitive and highly readable lines of code.

The actual implementation of the underlying links such as the ***SharedRegister*** can be viewed as memory mapped IO; any reads or writes to the port addresses send the data over the Network / Link. At another level, this abstraction could be viewed as named or numbered ports accessed by the node models. This structure is fairly typical of

Copyright © 2005 Cmpware, Inc. All rights reserved.



modeled networks. Other more complex networking structures, such as programmable networks will require more effort, but the effort should be proportional to the complexity and regularity of the network. In any case, each of these model is fast, parameterizable and reusable and takes on the order of tens of lines each.

Running the ALU Array

Figure 1 shows the Eclipse Preference Page for the *Cmpware* environment. This is set up to use the models discussed in the previous sections of the document, producing a 10 x 10 ***ALU*** array with a ***Torus*** network built from ***SharedRegister*** links..

Preferences

Workbench
Ant
Build Order
Cmpware
Help
Install/Update
Java
Plug-in Development
Run/Debug
Team

Cmpware

Cmpware Processor array definition

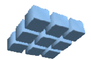
Processors: com.cmpware.cmp.models.ALU:10:10
Network: com.cmpware.cmp.networks.Torus
Link: com.cmpware.cmp.links.SharedRegister
Step size: 1
License: Beta
Proxy server:
Model path: Browse...

Restore Defaults Apply

Import... Export... OK Cancel

Figure 1: Configuring the ALU Array in the Cmpware Preference Pages.

Copyright © 2005 Cmpware, Inc. All rights reserved.



Once the models are specified by the preference page, a multiprocessor simulation model is allocated and the network interconnections put into place. Unlike many other simulation environments, this happens more or less instantaneously. For an array of this size, no noticable lag is present in allocating the array. The array simulation model is then represented as the array of elements in the main window as shown in **Figure 2**. Clicking on these nodes specifies the 'focus' node, and all displays show data relating to this selected processing node in the various windows.

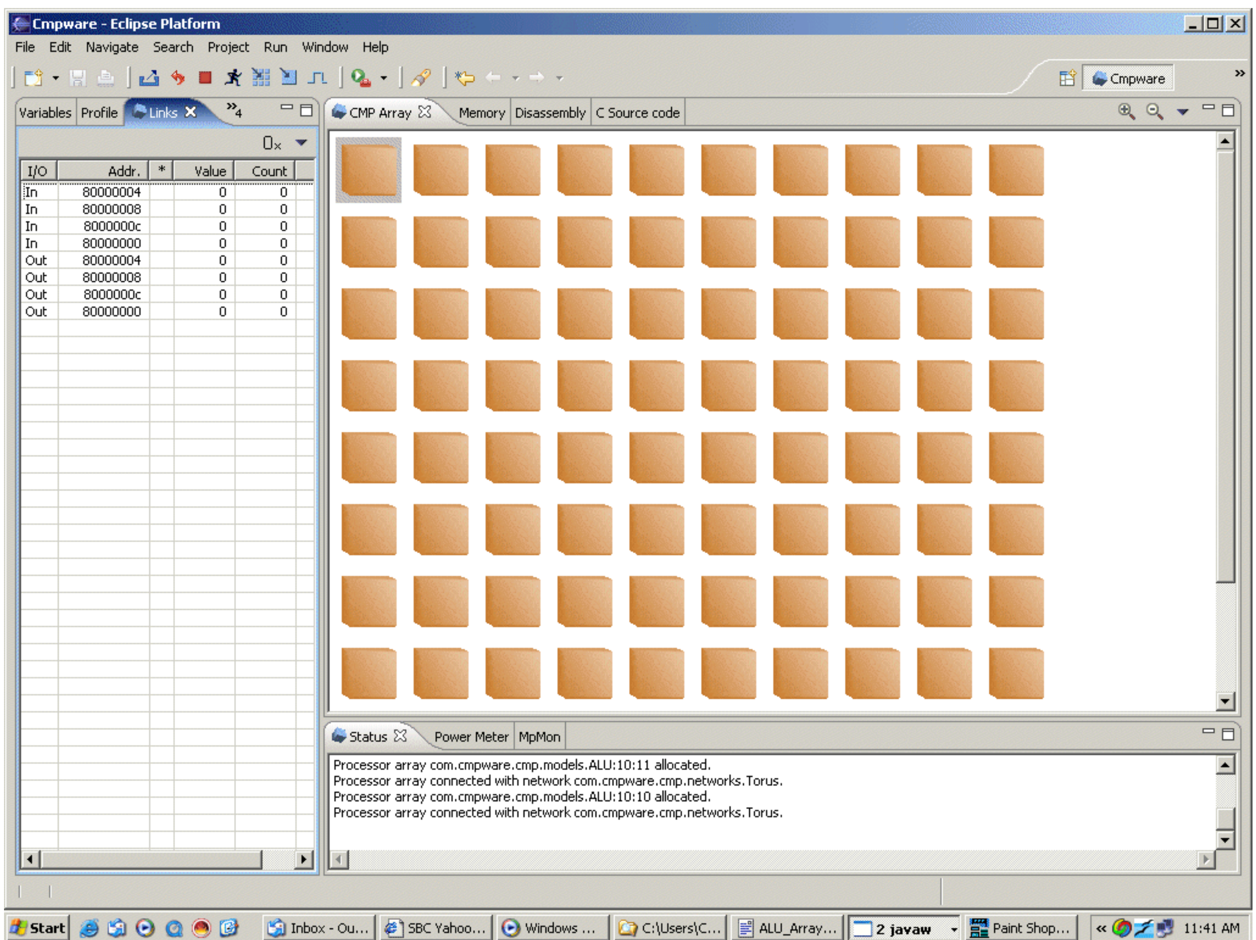
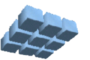


Figure 2: The ALU array.

As per the definition in the ALU model, these nodes come up in an unconfigured state. As a simple demonstation, the nodes will be configured to read a value from the 'north' input, increment it, and send it to the 'south' output. This can be done with a text file containing the string "**isn**". This file can be loaded with the **Load All** button ( ), which

Copyright © 2005 Cmpware, Inc. All rights reserved.



sends its contents to each of the nodes in the array. The memory dump in **Figure 3** shows the first three bytes are configured to "**isn**", which is the code for "*increment north input, write result to south*". A check of the **Memory** display shows the first three bytes set to "**isn**" (hexadecimal 69 73 6e) as expected.

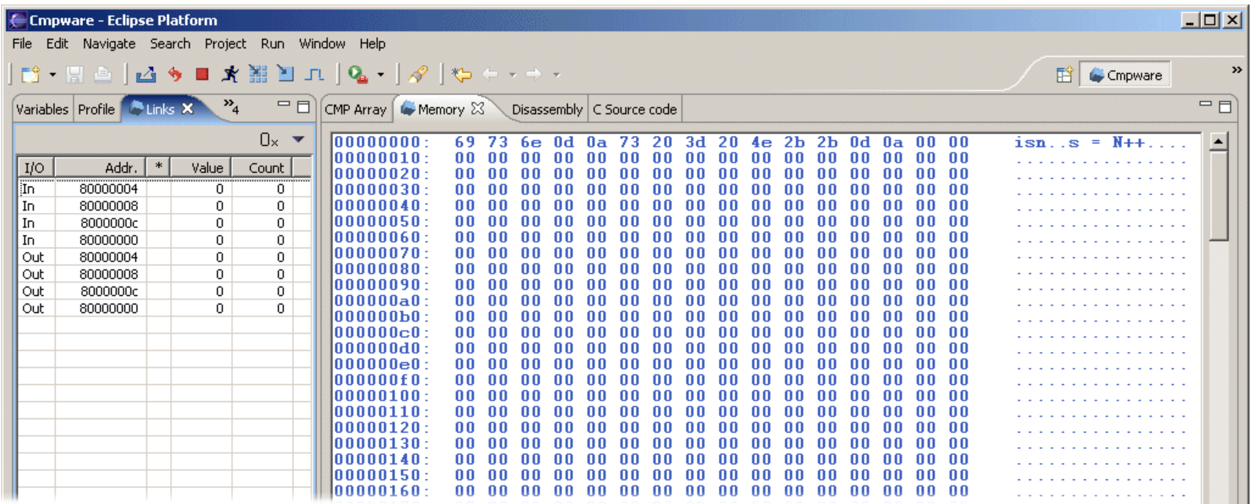


Figure 3: The ALU memory with "isn" configuration.

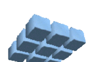
One small wrinkle in this circuit is that because it uses the 'data ready' handshaking in the ***SharedRegister*** links, it will not start up spontaneously. Instead, each of the node will wait for data to be written before it will attempt to perform the increment operation. So some selected nodes will be loaded with a different configuration, temporarily, to 'kick-start' the computation.

The 'kick start' code simply sends a constant value to the 'south' link to be read by the node below on the next cycle. This instruction is "**cs01**", which sends the **c**onstant value 0x**01** to the '**s**outh' link. This insruction is loaded to selected ALUs, then clocked for a single cycle with the Step button (). In this example, *ALU(1,1)*, *ALU(2,5)* and *ALU(5,7)* are loaded with the "**cs01**" configuration from a file named ***ALU_test0.txt***

After stepping for one cycle, the display will appear se in **Figure 4**. Here, only the nodes with the "**cs01**" operation configured are performing any work, sending the constant 0x01 to the south link. This can also be verified with the **Links** display on the left. All of the other nodes are idle and waiting for input and colored grey in the display. Subsequent clock steps will also result in the entire array being idle, since the data in the constant nodes has been written, but not yet read.

After pre-loading some of the south links with the constant value 0x01, all of the nodes

Copyright © 2005 Cmpware, Inc. All rights reserved.



are re-loaded with the original "**isn**" configuration from the ***ALU_test.txt*** file. Again, this file is loaded with the **Load All** button ( ), which sends its contents to each of the nodes in the array.

Subsequent clock stepping results in active (orange) nodes "walking" up the columns of nodes, as data is made available. And as data in the selected node is incremented, the values in the **Links** display can be seen to advance. Since there are 10 nodes in each column, each incrementing the value being passed on, the input value will ten larger than the previous value, which is then incremented and again passed to the next element in the array.

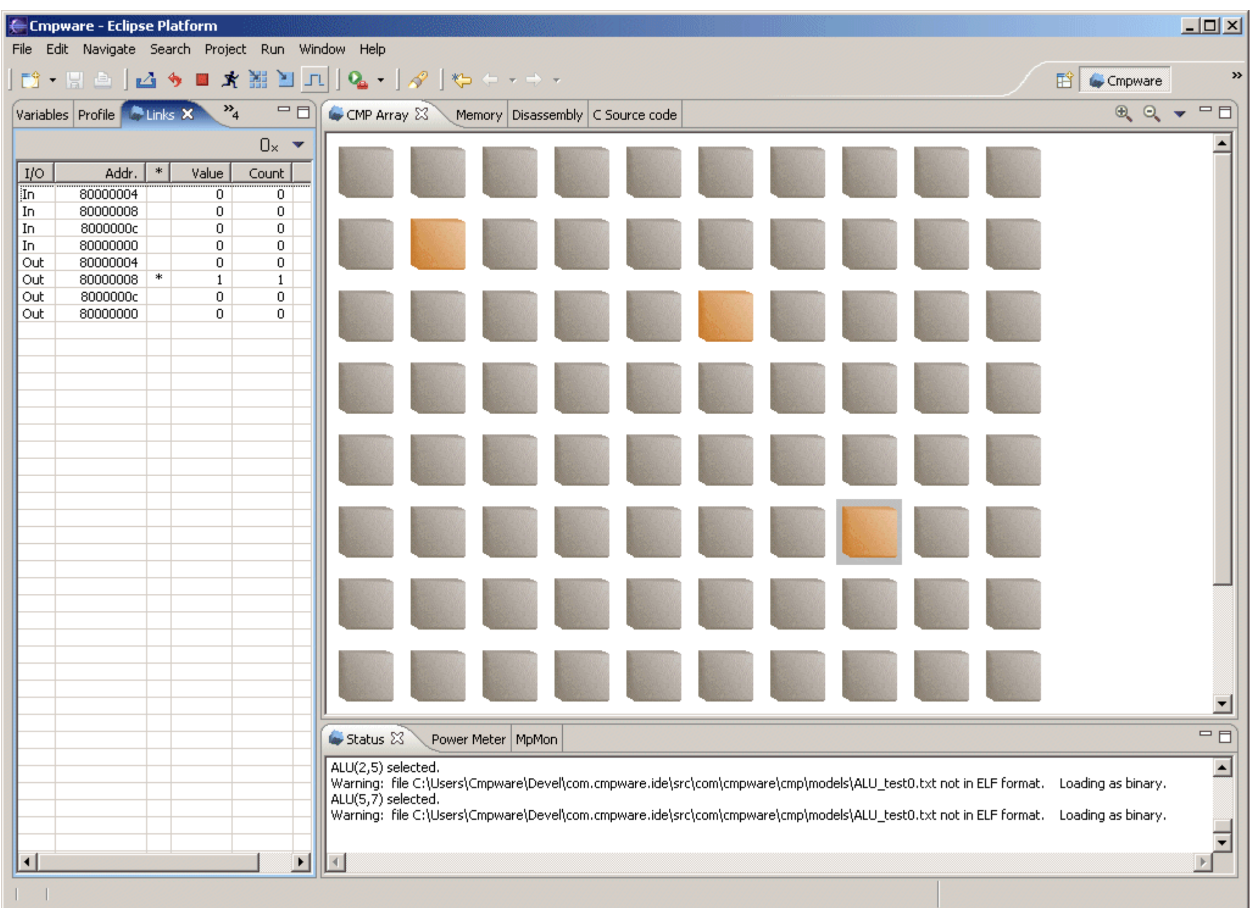
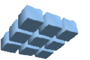


Figure 4: "Kick-starting' the ALU array.

Rather than stepping each cycle, it is possible to use the **Run** button ( ) to continuously step the simulation and update the displays. The **Stop** button ( ) can be used to pause the simulation. Note that this updates the simulation and displays on

Copyright © 2005 Cmpware, Inc. All rights reserved.


each clock cycle. It is possible to change the **Step Size** in the Preference page as in **Figure 1**. This will permit faster run-time simulation by updating the displays after some number of cycles. By setting this step size to a large value, it can be verified that the simulation for this model runs at approximately one million cycles per second. This is exceptional, considering that this model does relatively little work and quite a bit of communication, relative to other larger-grained processor systems.

Conclusions

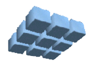
The document has describes the modeling of a simple ALU array. The effort to produce the model was very low, taking on the order of one hour by an engineer experienced in the use of the *Cmpware* tool. Perhaps notably, very little debug time was involved. Because of the structure of the models and the large number of displays, problems in the models tend to show up quickly and typically in some obvious fashion. In general, model debug is a very small part of development.

While modeling is often a significant component of the SoC design effort, tool and application development will likley be an even larger effort. In general, the higher the degree of programmability of the system, the more this will hold true. While *Cmpware* provides a powerful and friendly modeling environment, it is also a very powerful software development environment. Speeding the development of tools and applications can be very significant in decreasing overall system development time and decreasing the time to develop a final product.

Ultimately, the goal of the *Cmpware CMP-DK* is to provide a powerful, robust modeling tool, which in turn produces a software development environment early in the design process. Today, modeling tends to run in tandem with hardware development and is of a similar level of complexity. This is typically followed by tools development, then application development. It is often the tools and application development that lag in the product development of a modern SoC / programmable device. The focus of the *Cmpware* toolkit is to accelerate this process and pull in software development times, while at the same time increasing the quality of tools and applications.

Having a fast, powerful simulation model early in the system design process can not only find potential problems sooner, it can dramatically decrease the time to market of a product while also providing a higher quality end result.

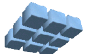
Copyright © 2005 Cmpware, Inc. All rights reserved.



Appendix A: ALU.txt

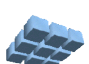
```
--
-- This file describes a simple ALU processing element.
-- It operates on network inputs and sends outputs to
-- a network output.  There is no state or sequencing.
--
-- The 'program' is stored at address zero in 4 bytes.  The
-- first byte is the opcode.  ASCII codes for C/C++/Java
-- operations are used, as well as ASCII digits for the
-- specific outpout ports.  This allows a simple text
-- editor to be used to produce the 'executable'.  The codes
-- used are below:
--
-- Operation   Opcode  ASCII
-- ---------   ------  -----
--  Add          +       43
--  Subtract     -       45
--  Multiply     *       42
--  Divide       /       47
--  Modulus      %       37
--  AND          &       38
--  OR           |      124
--  XOR          ^       94
--  NOT          ~      126
--  Shift right  >       62
--  Shift left   <       60
--  Constant     c       99
--  Increment    i      105
--  Decrement    d      100
--
--  The two input and one output ports are defined by
--  ASCII characters representing the ordinal directions
--  North (N), South (S), East (E) and West (W).  So
--  an addition that takes inputs from east and west and
--  sends them south is described by the text "+SEW".
--  Other data in the file will be ignored.
--  Constant uses a different format.  The second byte is
--  the output destination, but the next two bytes are
--  a hex string representation of an 8-bit constant.
--
--  Copyright (c) 2005 Cmpware, Inc.  All Rights Reserved.
--

ADD    43  d = a + b;
SUB    45  d = a - b;
MULT   42  d = a * b;
DIV    47  d = a / b;
MOD    37  d = a % b;
AND    38  d = a & b;
OR    124  d = a | b;
XOR    94  d = a ^ b;
NOT   126  d = ~a;
SHR    62  d = a >>> 1;
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



```
SHL     60  d = a << 1;
CONST   99  d = imm8;
INCR   105  d = a++;
DEC    100  d = a--;
```

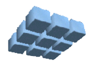

Copyright © 2005 Cmpware, Inc. All rights reserved.



Appendix B: ALU.java

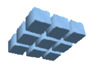
```
package com.cmpware.cmp.models;

import com.cmpware.cmp.Processor;
import com.cmpware.cmp.MemoryAccessException;
import com.cmpware.cmp.IllegalRegisterException;
import com.cmpware.cmp.IllegalOpcodeException;

import  com.cmpware.cmp.networks.Torus;


/**
**  This is a simple ALU processor.
**
**  <p>
**  Copyright (c) 2004, 2005 Cmpware, Inc.  All Rights Reserved.
**  <p>
**
**  @author SAG
*/

/*
**  Things to do to implement your processor:
**
**  - Set the definitions in the constructor to
**    appropriate values.
**  - Implement the processor decode logic in
**    decode().
**  - Modify execute() to pre-compute any values
**    that may be required by the op_*() methods.
**    These values should also be defined as
**    private data at the end of this class.
**  - Modify the getPC() and setPC() methods.  The
**    Program Counter (PC) is often a Special Register
**    or some bitfield contained in a Special Register.
**    If it is not, it may be useful to add a Special
**    register to hold the PC value.
**  - Implement the disassembler in dasm().  This
**    usually involves grouping similar operations
**    and building a string representation of these
**    operations.  You may want to look at other
**    implementations as examples.
**  - Implement the op_*() methods.  There should be
**    exactly one op_*() method per decoded instruction.
**    These should directly modify registers and perhaps
**    local data.
**  - Define the processor NOOP instruction.  Note that
**    this may be a dedicated instruction or just an
**    operation such as AND 0,0,0 that does not modify
**    the state of the processor.
**  - Define the processor Breakpoint instruction.  Note
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



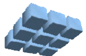
```
**    that this may be a dedicated instruction or just a
**    selected instruction with an Illegal Opcode.
**  - Define the General Purpose Register names.
**  - Define the Special Purpose Register names.
**
**  For many processors, this is all that needs to be done.
**  More complex processors may require overloading of
**  some of the predefined methods in the Processor()
**  class.  How and when this is done is highly dependent
**  on the particular processor implementation.
**
*/

public class  ALU extends Processor {


/** Copyright string */
public final static String copyright =
   "Copyright (c) 2004, 2005 Cmpware, Inc.  All Rights Reserved.";


/**
**  The constructor
**
*/

public ALU() {

   /* Define the processor */
   defineName("ALU");
   defineInstructionSize(4);
   defineRegisters(1);
   defineSpecialRegisters(1);
   defineBranchDelay(0);
   defineRegisterNames(regName);
   defineSpecialRegisterNames(sregName);
   defineOpcodeNames(opcodeName);
   defineNoop(NOOP);
   defineBreakpoint(BREAKPOINT);

   setEndian(BIG);

   /* Resize the memory */
   resize(1024);

   /* Reset the processor */
   reset();

   }  /* end ALU() */


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


Copyright © 2005 Cmpware, Inc. All rights reserved.



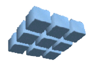
```
*/

public int  decode(int  instr) throws IllegalOpcodeException {
   int opcode = 0;

   /* *** implement decode logic here *** */
   opcode = (instr >> 24) & 0xff;

   /* Check for illegal opcodes (incl. breakpoint) */
   switch (opcode) {
      case  ADD:
      case  SUB:
      case  MULT:
      case  DIV:
      case  MOD:
      case  AND:
      case  OR:
      case  XOR:
      case  NOT:
      case  SHR:
      case  SHL:
      case  CONST:
      case  INCR:
      case  DEC:
            break;
      default:
         throw(new IllegalOpcodeException(instr));
      }  /* end switch{} */

   return (opcode);

   }  /* end decode() */


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

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

   /*  *** pre-compute any necessary values here *** */
   port_d = (byte) ((instr >> 16) & 0xff);
   port_a = (byte) ((instr >> 8) & 0xff);
   port_b = (byte) (instr & 0xff);
   /* Convert last two bytes from ASCII hex to binary */
   imm8 = ((((instr >> 8) & 0x000000ff) - (int) '0') * 16) +
          ((instr & 0x000000ff) - (int) '0');

   if (currentInstrCode != CONST) {
       a = read32(getPort(port_a));
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



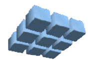
```
        if ((currentInstrCode != INCR) &&
            (currentInstrCode != DEC))
           b = read32(getPort(port_b));
        }

   /* Execute instruction */
   switch (currentInstrCode) {
      case  ADD:   op_add(); break;
      case  SUB:   op_sub(); break;
      case  MULT:   op_mult(); break;
      case  DIV:   op_div(); break;
      case  MOD:   op_mod(); break;
      case  AND:   op_and(); break;
      case  OR:   op_or(); break;
      case  XOR:   op_xor(); break;
      case  NOT:   op_not(); break;
      case  SHR:   op_shr(); break;
      case  SHL:   op_shl(); break;
      case  CONST:   op_const(); break;
      case  INCR:   op_incr(); break;
      case  DEC:   op_dec(); break;

      /* Opcode not found */
      default:
         /* Illegal opcodes should be caught in */
         /* the decode -- but just in case */
         System.out.println("Unexpected illegal opcode encountered.  "+
                            "(Instruction:  0x"+Integer.toHexString(instr)+")"
);
      }  /* end switch{} */

   /* Write the 'd' result to port */
   write32(getPort(port_d), d);

   }  /* end execute() */


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

public  int  getPC() {return (sr[0]);}


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

public  void  setPC(int pc) {sr[0] = pc;}
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



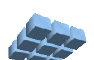
```
/**
**  Overrides the standard PC advancement.
**
*/

public void advancePC() {;}

/*
**  (non-Javadoc)
**  @see com.cmpware.cmp.Processor#dasm(byte[])
*/

public String  dasm(byte instr[]) {
   int   instrCode;
   String dasmStr;
   int  instrWord;

   /* Convert bytes to int */
   instrWord = toInt(instr);

   /* Decode the instruction */
   try {
      instrCode = decode(instrWord);
   } catch (IllegalOpcodeException ioe){
      return ("<error>");
   }  /* end try{} */

   /* Start with the opcode string */
   dasmStr = opcodeName[instrCode];

   /* Catch NOOP */
   if (toInt(getNoop()) == instrWord)
      return ("nop");

   /*  *** pre-compute any necessary values here *** */
   port_d = (byte) ((instrWord >> 16) & 0xff);
   port_a = (byte) ((instrWord >> 8) & 0xff);
   port_b = (byte) (instrWord & 0xff);
   /* Convert last two bytes from ASCII hex to binary */
   imm8 = ((((instrWord >> 8) & 0x000000ff) - (int) '0') * 16) +
          ((instrWord & 0x000000ff) - (int) '0');

   /* Decode instruction */
   switch (instrCode) {
      case  ADD:
      case  SUB:
      case  MULT:
      case  DIV:
      case  MOD:
      case  AND:
      case  OR:
      case  XOR:
      case  NOT:
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



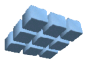
```
      case  SHR:
      case  SHL:
            dasmStr = dasmStr + " " + (char) port_d +
                      "," + (char) port_a + "," + (char) port_b;
            break;
      case  INCR:
      case  DEC:
            dasmStr = dasmStr + " " + (char) port_d +
                      "," + (char) port_a;
            break;
      case  CONST:
            dasmStr = dasmStr + " " + (char) port_d + "," + imm8;
            break;
      default:
         dasmStr = "";
         break;
      }  /* end switch{} */

   return(dasmStr);

   }  /* end dasm() */



/** The ADD operation */
private void op_add() {
   d = a + b;
   }  /* end op_add() */


/** The SUB operation */
private void op_sub() {
   d = a - b;
   }  /* end op_sub() */


/** The MULT operation */
private void op_mult() {
   d = a * b;
   }  /* end op_mult() */


/** The DIV operation */
private void op_div() {
   d = a / b;
   }  /* end op_div() */


/** The MOD operation */
private void op_mod() {
   d = a % b;
   }  /* end op_mod() */
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



```
/** The AND operation */
private void op_and() {
   d = a & b;
   }  /* end op_and() */


/** The OR operation */
private void op_or() {
   d = a | b;
   }  /* end op_or() */


/** The XOR operation */
private void op_xor() {
   d = a ^ b;
   }  /* end op_xor() */


/** The NOT operation */
private void op_not() {
   d = ~a;
   }  /* end op_not() */


/** The SHR operation */
private void op_shr() {
   d = a >>> 1;
   }  /* end op_shr() */


/** The SHL operation */
private void op_shl() {
   d = a << 1;
   }  /* end op_shl() */


/** The CONST operation */
private void op_const() {
   d = imm8;
   }  /* end op_const() */


/** The INCR operation */
private void op_incr() {
   d = a + 1;
   }  /* end op_incr() */


/** The DEC operation */
private void op_dec() {
   d = a - 1;
   }  /* end op_dec() */
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



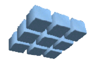
```
/**  The NOOP instruction */
private final static byte NOOP[] =
    {(byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00};

/**  The Breakpoint code */
private final static byte BREAKPOINT[] =
   {(byte) 0xff, (byte) 0xff, (byte) 0xff, (byte) 0xff};


/** The General purpose register names */
private static final String  regName[] = {
   "r0"
   };


/** The Special purpose register names */
private static final String  sregName[] = {
   "pc"
   };


/** The opcode names (mapped to instruction codes) */
private static final String  opcodeName[] = {
"<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",   "<illegal
opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "mod",   "and",   "<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "mult",   "add",
   "<illegal opcode>",   "sub",   "<illegal opcode>",   "div",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
```


Copyright © 2005 Cmpware, Inc. All rights reserved.

```
"<illegal opcode>",
   "shl",   "<illegal opcode>",   "shr",   "<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "xor",   "<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",   "const",
   "dec",   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
   "<illegal opcode>",   "incr",   "<illegal opcode>",   "<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "<illegal opcode>",   "<illegal opcode>",   "<illegal opcode>",
"<illegal opcode>",
   "or",   "<illegal opcode>",   "not"

   };


/**  The instruction codes */
private final static int  ADD = 43;
private final static int  SUB = 45;
private final static int  MULT = 42;
private final static int  DIV = 47;
private final static int  MOD = 37;
private final static int  AND = 38;
private final static int  OR = 124;
private final static int  XOR = 94;
private final static int  NOT = 126;
private final static int  SHR = 62;
private final static int  SHL = 60;
private final static int  CONST = 99;
private final static int  INCR = 105;
private final static int  DEC = 100;
```

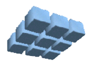
```
/**
**  This method translates the byte / characters "N",
**  'S', 'E' and 'W' into port memory addresses for
**  the Torus network (although these addresses could be
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



```
**  re-used in other networks).  This is just used to
**  help permit a text file to be loaded as a binary
**  'program'.
**
** @param port  The port byte / char.  This should be
**        an upper or lower case 'n', 'e', 's', or 'w'.
**
** @return  This methos returns the port address associated
**          with the input port parameter.
*/

private final int getPort(byte port) {

    switch(port) {
       case  'n':
       case  'N':  return (Torus.NORTH);
       case  's':
       case  'S':  return (Torus.SOUTH);
       case  'e':
       case  'E':  return (Torus.EAST);
       case  'w':
       case  'W':  return (Torus.WEST);
       default:  return (-1);
       }  /* end switch{} */

   }  /* end () */


/* *** put any additional data structures here *** */
private byte  port_d = 0;
private byte  port_a = 0;
private byte  port_b = 0;


private int d = 0;
private int a = 0;
private int b = 0;
private int imm8 = 0;

};  /* end class ALU */
```

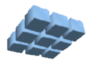

Copyright © 2005 Cmpware, Inc. All rights reserved.



Appendix C: Torus.java

```
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.
**
**  <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);
```


Copyright © 2005 Cmpware, Inc. All rights reserved.



```
          /* 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 'east' IO register */
public static int EAST = (IO_ANCHOR + 4);

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

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

}  /* end class Torus */
```


Copyright © 2005 Cmpware, Inc. All rights reserved.