Building a Processor Model

Cmpware, Inc.

Introduction

The *Cmpware Multiprocessor Development Environment* 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. One of the features of the *Cmpware* system is the relative ease with which even fairly sophisticated processor models can be built. This document is provided to give an introduction to the process of building models.

The Processor Abstract 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 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. All that is required to produce a new processor model is to create a new class which extends `Processor`.

Because `Processor` is a "abstract" class, it also contains some methods which the new class will have to supply. Figure 1 shows these specific methods. Each of these methods must be supplied by the `Processor` subclass. Each will be discussed briefly below.

`Decode()`: The `decode()` method supplies the instruction decode for the processor. The input parameter is an instruction represented as an integer. This method will use the instruction to return the decoded opcode for the instruction represented by this instruction. In some cases, decoding is very simple. In the case of many simple RISC machines, this method may be as simple as returning a field (say, the first eight bits) of the instruction. In other cases, a more complicated decode is required. One comment about the decode method is that the value returned may not necessarily be related to the exact bit pattern of the opcode. It must only return a unique instruction code that the `execute()` method can recognize. Additionally, this method must throw an

`IllegalOpcodeException` for all possible illegal opcodes. Note that breakpoints are implemented as an illegal opcode, so it is important that the breakpoint opcode throw this exception.

```
public abstract int decode(int instr) throws
IllegalOpcodeException;

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

public abstract int getPC();

public abstract void setPC(int pc);

public abstract String  dasm(byte  instr[]);
```

Figure 1: The `Processor` class abstract methods.

`Execute()`: The `execute()` method is responsible for the execution of the instruction in the processor. It takes in as its only parameter an integer representation of the instruction to be executed. This method will typically call the `decode()` method and use the resulting opcode to select which instruction functionality will be executed. In smaller, simpler processors, this may all be done inside of the `execute()` method, typically with a `switch()` statement. With more sophisticated processors, and perhaps for maintainability of the code, it is advisable that each instruction be implemented in its own method, which is then called from a `switch()` statement in execute. Finally, `execute()` may propagate a `MemoryAccessException` or an `IllegalRegisterException`. These will be thrown by other pieces of the `Processor` class and should not be of direct interest when constructing the `execute()` method.

`GetPC()` and **`setPC()`**: The `getPC()` and `setPC()` methods are supplied to give the simulator access to the model's Program Counter (PC). These should be extremely simple methods returning a register or portion of a register. The reason these are specified as methods and not as simply as some sort of pointer to a register is that many processors only use a subset of bits in a register in a variety of ways. Some processors will use, for instance, the lower 20 bits in a status register. Others may shift the value, ignoring the lower bits. Some processors may do both. At the most complex, it is expected that these methods will perform some shifting and masking of a

register.

`Dasm()`: The `dasm()` method takes in as its only parameter a byte array representing an instruction. It is expected to return a string representing the text assembly language for the instruction. While this method is not completely essential to the operation of the simulator, it provides very useful information to the user interfaces. It is strongly advised that the time be taken to make this method as complete and accurate as possible.

The ProcGen Processor Model Generator

Building *Cmpware* processor models is not particularly difficult. With an instruction set definition at hand, it becomes mostly tedious typing of opcodes and instruction functions. To assist in this effort and to help provide more accurate models, a processor generator, *ProcGen,* has been built. *ProcGen* is in no way a model specification language and has a very simple input format. This tool can dramatically reduce the effort required to build a model, while simultaneously improving the accuracy and the quality of the code.

The *ProcGen* tool is a stand-alone utility and may be downloaded as a single compiled Java class file, P*rocGen.class* To execute this program, from a command-line prompt, type the command below:

```
$ java ProcGen
Usage:  java ProcGen <processor name> <template file> <data
file>
```

ProcGen has responded with a “usage” help. *ProcGen* requires three parameters: the name of the processor being modeled, a template file and a data file. The processor name is just a string containing the name of the processor being modeled, for instance “RISC16”. The template data file is supplied by *Cmpware* and is available as *ProcTemplate.j* This file contains some Java code with tags used by *ProcGen* to customize the file. Finally, the last parameter is the name of a data file which contains a processor definition. This file must be entered by the person constructing the model. The format is described in the section below.

The ProcGen Model Format

The format for the `Processor` model was designed to be as simple as possible. This is because this data is not a complete modeling language, but rather a helper to get an

initial Java source file constructed. This Java source file can then be edited as necessary.

The reason for this approach is twofold. First, a complex language for specifying models would put an additional burden on the modeler. This language would have to be learned and understood. Since some compilable code is inevitably generated by such tools, it was decided that the effort would go into providing a clean, simple and efficient interface in the native language (Java) which would be easy to use. The second reason for avoiding a processor modeling language is that such languages tend to become complicated as they attempt to cover all the possibilities in processor design. It has been our experience that all the possibilities can never really be covered, and it is best to provide a good structure to model these architectural features instead.

```
--
--  This describes a simple
--  (fictional) processor
--
--  Copyright (c) 2004 Cmpware, Inc.
--  All Rights Reserved.
--

add    1   r[c] = r[a] + r[b];
addi   2   r[c] = r[c] + imm8;
sub    3   r[c] = r[a] - r[b];
xor    4   r[c] = r[a] ^ r[b];
not    5   r[c] = ~r[b];
or     6   r[c] = r[a] | r[b];
and    7   r[c] = r[a] & r[b];
shl    8   r[c] = r[a] << r[b];
shr    9   r[c] = r[a] >> r[b];
br    10   branch(imm12);
bnz   11   if (r[a] != 0) branch(r[b]);
bz    12   if (r[a] == 0) branch(r[b]);
ld    13   r[c] = read32(r[b]);
st    14   write32(r[c], r[b]);
```

Figure 2: A simple processor.

Figure 2 gives an example of a simple processor modeled with this description. The format it very straight forward. The file is line-based with three fields per line recognized. The first field contains the opcode name, the second field, the numeric code associated with this opcode. The remainder of the line consists of Java code

which implements this instruction. Note that this code is intended to model the typically simple functionality of a processor instruction. If the operation is more complex and requires several lines, a function call placeholder may be used. Once the Java code is generated by *ProcGen*, the function can me implemented in the resulting Java source code file.

Figure 2 describes a simple processor that provides a variety of typical instructions. It is in no way a complete processor, but it can be used to illustrate the model building process. This file has been saved to *Simple.txt* and used to drive the *ProcGen* generation of the model. The command below shows *ProcGen* being run with this file for its input. Note that *ProcGen* writes to the standard output, so the resulting model will be piped to the file *Simple.java*.

```
$ java ProcGen Simple ProcTemplate.j Simple.txt >
Simple.java
```

Fixing Compile-time Errors

Now we have the basic Java file for the *Simple* processor model. But running a Java compiler on this file will immediately produce several errors. *ProcGen* has done the tedious work and generated dozens of lines of well-structured and commented code. But some work remains to configure and flesh out this model.

This is a good time to have a look at the generated code. All of the required abstract methods are there, as well as some definitions and data structures to make the process of building the model easier. This Java source code file also contains some basic instructions for customizing your model.

Perhaps the first and largest source of errors are the missing variables used in the instruction descriptions. The *ProcGen* description uses elements such as `a` `b`, `c`, `imm8` and `imm12`, but never defines these. By simply defining private integers variables for each of these elements in the *Simple.java* source code file, nearly all of the errors disappear.

The next step is to resolve the import dependencies from the rest of the *Cmpware* code. This code can be found in a Java JAR file in the *Eclipse* plugin. If you are using Eclipse to develop this model, then you will have to go to the **Properties** and click on 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 these

missing dependencies. Other Java development systems should have a similar method for including external JAR files. Consult the documentation for your particular system.

At this point, there are still two more outstanding errors. The op_ld() and op_st() methods have unhandled MemoryAccessExceptions. These are the load and store operations, respectively, so we will have to tell the model that these operations will potentially throw these memory access exceptions. Figure 3 shows these two methods with the added **throws MemoryAccessException** in the method interface.

```
/** The LD operation */
private void op_ld() throws MemoryAccessException {
   r[c] = read32(rb);
   }  /* end op_ld() */

/** The ST operation */
private void op_st() throws MemoryAccessException {
   write32(rc, rb);
   }  /* end op_st() */
```

Figure 3: Fixing the unhandled exceptions.

Note that the load and store operation use the predefined Memory access methods `read32()` and `write32()`. These access methods should always be used to read and write memory. See the `com.cmpware.cmp.Memory` class documentation for other memory access methods.

Configuring the Model

At this point, the model compiles correctly, but there are still some missing pieces that need to be supplied. First, as mentioned previously is the `decode()` method. This brings up the subject of the instruction format, which we have ignored until now.

Rather than trying to specify the instruction formats as part of the *ProcGen* modeling language, this part of the model is left to be defined in the model Java code. This approach was taken primarily because instruction formats vary widely across processor architectures. It is believed that trying to support this variety in the modeling language would unnecessarily complicate its syntax.

The instruction for used for the *Simple* processor is, well, simple. All instructions are 16 bits. The first four bits are the opcode. For most of the instructions, the remaining bits

will be broken up into three 4-bit fields. Each of these fields will address a register in the 16-entry register file. Corresponding to the specification given to *ProcGen*, these fields will be called `c`, `a`, and `b`. These represent the destination and two source registers, respectively.

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

   /* *** implement decode logic here *** */
   opcode = ((instr >> 12) & 0x0f);
   c = ((instr >> 8) & 0x0f);
   a = ((instr >> 4) & 0x0f);
   b = (instr & 0x0f);
   imm8 = (instr & 0x00ff);
   imm12 = (instr & 0x0fff);

   /* Check for illegal opcodes (incl. breakpoint) */
   switch (opcode) {
      case 0:
      case 15:
         throw(new IllegalOpcodeException(instr));
      }  /* end switch{} */

   return (opcode);

   }  /* end decode() */
```

Figure 4: The `decode()` method.

The two other fields used in the branch operations are the immediate instructions are `imm8` and `imm12`. These are the last 8 and 12 bits in the instruction word, respectively.

This gives all of the information necessary to implement the `decode()` method. The actual decode is rather simple, with the first four bits being the returned opcode. The other instruction values will also be extracted and stored to the local variables for later use. Figure 4 shows the implementation of the `decode()` method.

Of course, this is a relatively simple processor. But even more complex processors may have a simple decode and may not be any more complicated to implement than this method. Of note here is that all fields, even overlapping ones, are all decoded.

This is somewhat inefficient, since no instruction will make use of all of these fields. But it is much simpler to implement, and a small price to pay in performance, as opposed to decoding only the necessary fields in each of the instruction implementation methods.

The next major piece of implementation is the `dasm()` method. This involves filling in the supplied `switch()` statement with string representations of the instruction. Since there are only three instruction formats here, the implementation is relatively simple. This code is not reproduced here but may be viewed in the final *Simple.java* source file.

Now all that is left is to be sure that all of the defaults provided by *ProcGen* are appropriate. Earlier the `setPC()` and `getPC()` abstract methods were mentioned. These by default use Special Register 0 (`sr[0]`) as the Program Counter. This is an acceptable value.

The `Simple()` constructor also contains several run-time configurable values. Most of these are acceptable for this processor, but at least one should be changed. The instruction size is currently set to four bytes (32 bits). This should be changed to **`defineInstructionSize(2)`** to indicate a 2-byte (16 bit) instruction word.

The Special Register size is defaulted to 16 entries. This is harmless, but there is only one Special Register defined in this architecture, so we may which to change this value from **`16`** to **`1`**. If this is done, the **`sregName[]`** string array should also me modified to have a single element, **`"pc"`**.

```
/* Some simple test code loaded at reset */
private final static byte testCode[] = {
    (byte) 0x01, (byte) 0x20,  // ADDI 0,1
    (byte) 0x00, (byte) 0x20,  // noop
    (byte) 0x00, (byte) 0xa0   // BR 0
    };
```

Figure 5: Some hand-compiled Simple code.

Lastly, there a noop and a breakpoint instruction must be defined. The defaults are 32-bit values set to `0x00000000` and `0xffffffff`, respectively. Since both 0 and 0xf (15) are undefined, we can keep `0xffffffff` as a breakpoint, but it must be shortened to 16 bits. The NOOP instruction, however, should be re-defined to be something which does not change the state of the processor. One such instruction would be to add zero to a register. So **`ADDI r[0],0`** would be an appropriate NOOP. This must be defined as a a 16-bit value. Since the ADDI opcode is 2, the binary code

for the NOOP instruction is `0x2000`. Note that like disassembly, this is not a crucial component of the model, but it will help improve its usability.

This is all that is required to produce the model for the *Simple* processor. But because we have just made up this architecture, there are no compilers or assemblers available for it. In order to perform some basic testing, the model will be modified slightly to pre-load a piece of hand-compiled code. This is done by defining a local byte array containing some *Simple* binary code as in Figure 5.

In order to load this code at reset, we will just overload the standard `Processor.reset()` method and load this code at address zero. The code to perform this is shown in Figure 6. It will load this code fragment at address 0 in the memory each time the processor is reset. This technique is also useful for emulation a Read-Only Memory (ROM) image that may be either permanently in memory or loaded upon reset.

```
public void  reset() {

   /* Call Processor.reset() first */
   super.reset();

   /* Write the test code to address 0 */
   try {
      write(0, testCode);
   } catch (MemoryAccessException mae) {
      System.out.println("Warning:  Could not load
test code.");
   }

   }  /* end reset() */
```

Figure 6: Loading the hand-compiled Simple code at reset.

Testing the Model

At this point we have a compiled Java class file that will serve as the simulation model for the *Simple* processor. 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 requires 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 processor 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 1 x1 array of *Simple* processors, then attempt to execute the pre-loaded code. Figure 7 below give a partial trace that demonstrates the Simple processor running.

```
$ java -classpath ide.jar com.cmpware.cmp.MpMon
(null)> a 1 1 Simple
[0,0]Simple> d 0
0000:   0120    addi   r[0], 1
0002:   0020    nop
0004:   00a0    br   0

[0,0]Simple> t
0000:   0120    addi   r[0], 1

[0,0]Simple> s 100
[0,0]Simple> r
r0:00000022  r1:00000000  r2:00000000  r3:00000000
r4:00000000  r5:00000000  r6:00000000  r7:00000000
r8:00000000  r9:00000000  r10:00000000  r11:00000000
r12:00000000  r13:00000000  r14:00000000  r15:00000000

pc:00000000

[0,0]Simple>
```

Figure 7: The *Simple* processor running under the command line debug monitor.

From this brief test, it is clear that the processor is behaving properly. Code is executing and sequencing properly, and registers are being updated and displayed as expected. Further testing, including the use of such features as breakpoints, is

recommended at this point.

Finally, a very simple performance test indicates that this simulaton moder, on this portion of code, is capable of executing ten million instructions is approximately four seconds on a standard AMD Athlon 1800+ PC running Windows XP. While this is a relatively simple processor, this is a good indication of the performance one may expect with these models. In fact, adding instructions will do little to slow the performance of this sort of architectural model.

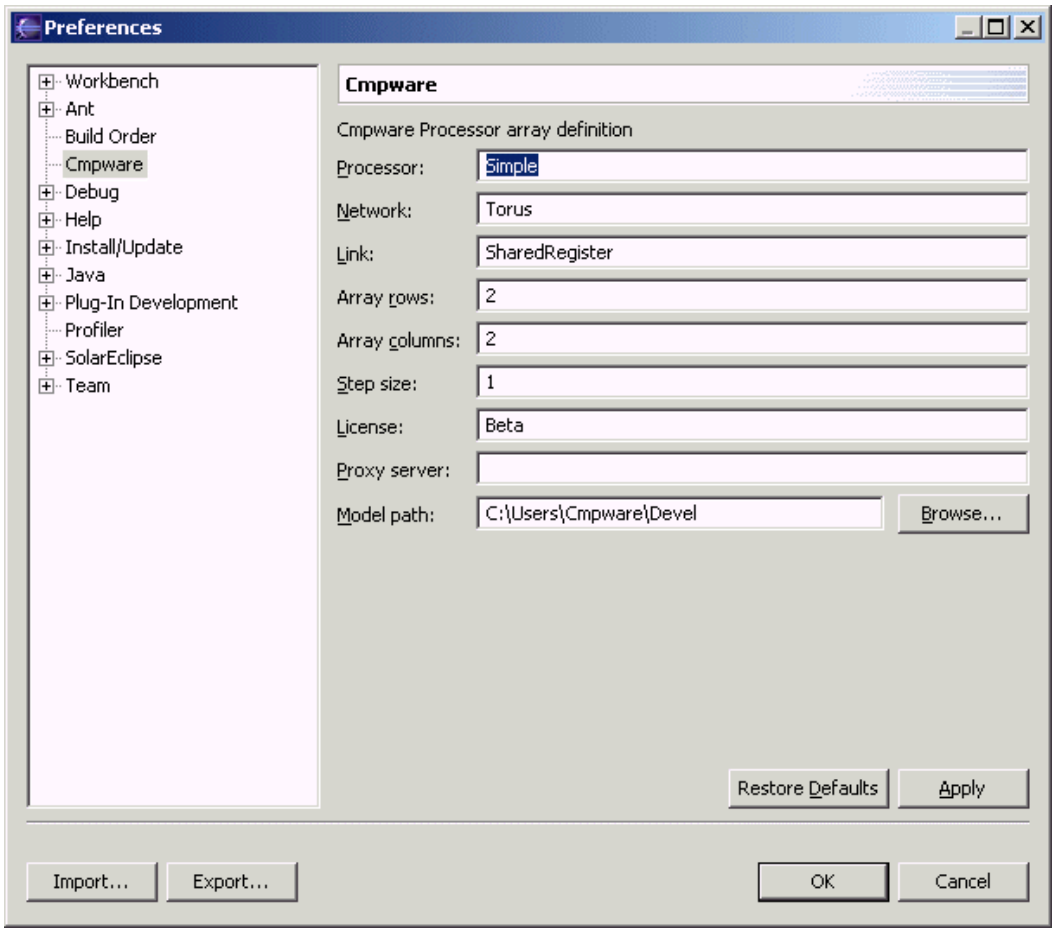

Figure 8: Setting preference for the *Simple* processor in the Eclipse IDE

Using the Model in the Eclipse IDE

If the command line test works, the Eclipse IDE should also function. Once the *Cmpware* Eclipse IDE is brought up, the *Simple* processor must be selected from the

Windows --> Preferences --> Cmpware preference page. This is done by changing the Processor field to **Simple**. 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 *Simple* processor array is allocated and initialized. Figure 9 shows the *Cmpware* IDE using the new *Simple* model. This view shows the registers and the disassembly after several steps. Checking the other views verifies that the processor is indeed functioning correctly.

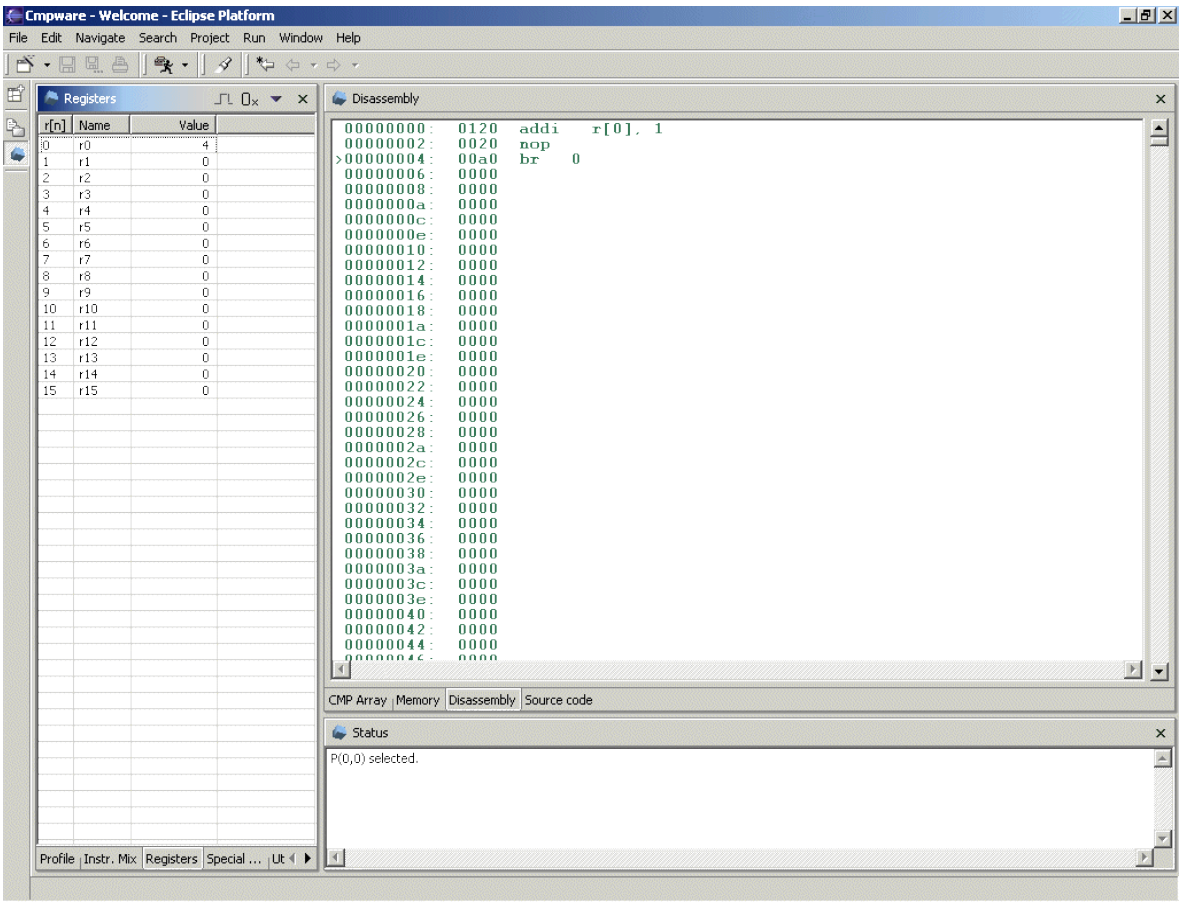

Figure 9: The *Simple* processor running under the Eclipse IDE

Some Comments on Architecture Modeling

So far we have used the *ProcGen* tool to produce the basic simulation model which we then modified to complete describe the *Simple* architecture. But in the real world, architectures are sometimes more complex. This is the reason that *Cmpware* has take the approach of providing a source-level (Java) interface to its model rather than relying on a more complicated specification language based approach. In most real-world architectures there are features which may not be easily modeled using other less flexible techniques.

There are many examples, but a simple and interesting one is the Register 0 in the Altera *NIOS II* processor. This register is always set to zero. Writing to it has no effect, and all reads return a zero. This is unusual, but potentially useful behavior for a register. It does, however, break most of the assumptions one would have concerning the simulation of a register file.

There are many approaches to modeling this behavior. An to support these various approaches the *Cmpware* interface permits re-defining of essentially all of the underlying standard processor model. The solution that was used by *Cmpware* to implement this architectural feature was to simply add a single line of code at the very end of the `execute()` method setting `r[0]` to zero. This does the trick, clearing out any data set by an instruction before the next instruction has a chance to read it. Because this is in the `execute()` method, there is some small performance penalty to be paid on each cycle, but it is small compared to more complex schemes. And the code ends up being simpler and more maintainable.

Conclusions

The document has described the modeling of a processor in the *Cmpware* environment. The process is relatively simple and can produce fast, high-quality processor simulation models with relatively little effort. It is estimated that just a few hours are required to engineer a simulation model for a modern processor.

> *Note: all of the files described in this document can be downloaded as a ZIP file from: http://www.cmpware.com/Secure/Modeling.zip*

Appendix A: The Simple Processor definition

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

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


/**
**  Describe your processor here.
**
**  <p>
**  Copyright (c) 2004 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
**    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  Simple extends Processor {


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


/**
**  The constructor
**
*/

public Simple() {

   /* Define the processor */
   defineName("Simple");
   defineInstructionSize(2);
   defineRegisters(16);
   defineSpecialRegisters(1);
   defineBranchDelay(0);
   defineRegisterNames(regName);
   defineSpecialRegisterNames(sregName);
```


```
   defineOpcodeNames(opcodeName);
   defineNoop(NOOP);
   defineBreakpoint(BREAKPOINT);

   /* Resize the memory */
   resize(64*1024);

   /* Reset the processor */
   reset();

   }  /* end Simple() */


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

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

   /* *** implement decode logic here *** */
   opcode = ((instr >> 12) & 0x0f);
   c = ((instr >> 8) & 0x0f);
   a = ((instr >> 4) & 0x0f);
   b = (instr & 0x0f);
   imm8 = (instr & 0x00ff);
   imm12 = (instr & 0x0fff);

   /* Check for illegal opcodes (incl. breakpoint) */
   switch (opcode) {
      case 0:
      case 15:
         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 *** */

   /* Execute instruction */
   switch (currentInstrCode) {
      case  ADD:   op_add(); break;
      case  ADDI:   op_addi(); break;
      case  SUB:   op_sub(); break;
      case  XOR:   op_xor(); break;
      case  NOT:   op_not(); break;
      case  OR:   op_or(); break;
      case  AND:   op_and(); break;
      case  SHL:   op_shl(); break;
      case  SHR:   op_shr(); break;
      case  BR:   op_br(); break;
      case  BNZ:   op_bnz(); break;
      case  BZ:   op_bz(); break;
      case  LD:   op_ld(); break;
      case  ST:   op_st(); 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{} */

   }  /* 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;}


/*
**  (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 *** */
   c = ((instrWord >> 8) & 0x0f);
   a = ((instrWord >> 4) & 0x0f);
   b = (instrWord & 0x0f);
   imm8 = (instrWord & 0x00ff);
   imm12 = (instrWord & 0x0fff);
```


```
   /* Decode instruction */
   switch (instrCode) {
      case  ADD:
      case  SUB:
      case  XOR:
      case  OR:
      case  AND:
      case  SHL:
      case  SHR:
         dasmStr = dasmStr + "   r[" + c + "], r[" + a + ", r[" +
 b + "]";
         break;
      case  NOT:
         dasmStr = dasmStr + "   r[" + c + "], r[" + a + ", r[" +
 b + "]";
         break;
      case  BR:
         dasmStr = dasmStr + "   " + imm12;
         break;
      case  BNZ:
      case  BZ:
         dasmStr = dasmStr + "  r[" + a + ", r[" + b + "]";
         break;
      case  ADDI:
         dasmStr = dasmStr + "   r[" + c + "], " + imm8;
         break;
      case  LD:
      case  ST:
         dasmStr = dasmStr + "  r[" + c + ", r[" + b + "]";
         break;
      default:
         dasmStr = "<error>";
         break;
      }  /* end switch{} */

   return(dasmStr);

   }  /* end dasm() */


/*
** (non-Javadoc)
```


```
** @see com.cmpware.cmp.Processor#reset()
*/

public void  reset() {

   /* Call Processor.reset() first */
   super.reset();

   /* Write the test code to address 0 */
   try {
      write(0, testCode);
   } catch (MemoryAccessException mae) {
      System.out.println("Warning:  Could not load test code.");
   }

   }  /* end reset() */



/** The ADD operation */
private void op_add() {
   r[c] = r[a] + r[b];
   }  /* end op_add() */


/** The ADDI operation */
private void op_addi() {
   r[c] = r[c] + imm8;
   }  /* end op_addi() */


/** The SUB operation */
private void op_sub() {
   r[c] = r[a] - r[b];
   }  /* end op_sub() */


/** The XOR operation */
private void op_xor() {
   r[c] = r[a] ^ r[b];
   }  /* end op_xor() */


/** The NOT operation */
```


```
private void op_not() {
   r[c] = ~r[b];
   }  /* end op_not() */


/** The OR operation */
private void op_or() {
   r[c] = r[a] | r[b];
   }  /* end op_or() */


/** The AND operation */
private void op_and() {
   r[c] = r[a] & r[b];
   }  /* end op_and() */


/** The SHL operation */
private void op_shl() {
   r[c] = r[a] << r[b];
   }  /* end op_shl() */


/** The SHR operation */
private void op_shr() {
   r[c] = r[a] >> r[b];
   }  /* end op_shr() */


/** The BR operation */
private void op_br() {
   branch(imm12);
   }  /* end op_br() */


/** The BNZ operation */
private void op_bnz() {
   if (r[a] != 0) branch(r[b]);
   }  /* end op_bnz() */


/** The BZ operation */
private void op_bz() {
   if (r[a] == 0) branch(r[b]);
```


```
   }  /* end op_bz() */


/** The LD operation */
private void op_ld() throws MemoryAccessException {
   r[c] = read32(r[b]);
   }  /* end op_ld() */


/** The ST operation */
private void op_st() throws MemoryAccessException {
   write32(r[c], r[b]);
   }  /* end op_st() */




/**  The NOOP instruction */
private final static byte NOOP[] =
    {(byte) 0x00, (byte) 0x20};

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


/** The General purpose register names */
private static final String  regName[] = {
   "r0",  "r1", "r2",  "r3",
   "r4",  "r5", "r6",  "r7",
   "r8",  "r9", "r10",  "r11",
   "r12",  "r13", "r14", "r15"
   };


/** 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>",   "add",   "addi",   "sub",
```


```
    "xor",   "not",   "or",   "and",
    "shl",   "shr",   "br",   "bnz",
    "bz",   "ld",   "st"

    };


/**  The instruction codes */
private final static int  ADD = 1;
private final static int  ADDI = 2;
private final static int  SUB = 3;
private final static int  XOR = 4;
private final static int  NOT = 5;
private final static int  OR = 6;
private final static int  AND = 7;
private final static int  SHL = 8;
private final static int  SHR = 9;
private final static int  BR = 10;
private final static int  BNZ = 11;
private final static int  BZ = 12;
private final static int  LD = 13;
private final static int  ST = 14;



/* *** put any additional data structures here *** */

private int a;
private int b;
private int c;
private int imm8;
private int imm12;

/* Some simple test code loaded at reset */
private final static byte testCode[] = {
    (byte) 0x01, (byte) 0x20,  // ADDI 0,1
    (byte) 0x00, (byte) 0x20,  // noop
    (byte) 0x00, (byte) 0xa0   // BR 0
    };

};  /* end class Simple */
```