Modeling Processors with the Cmpware CMP-DK FastModel Interface (Version 3.2 for Eclipse 3.2)

Cmpware, Inc.

Introduction

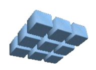
The *Cmpware Configurable Multiprocessor Development Kit (CMP-DK)* is a multiprocessor simulation and software development environment. It provides fast and efficient modeling of multiprocessor architectures as well as support for software development on such systems. The goal of supporting software development is achieved by providing an interactive, display-rich environment that permits large amounts of information to be displayed in a fast, simple and intuitive format. Such capabilities are essential in analyzing the behavior of complex multiprocessor systems.

The *Cmpware CMP-DK* version 3.2 for *Eclipse* 3.2 and higher contains customizable models for various popular processors, memories and communication links. These models transparently interface to a powerful *Eclipse*-based Integrated Development Environment (IDE) that provides a multiprocessor architecture and software development environment. This environment features displays for:

- Source Code Tracing
- Source Code Variables
- Disassembly
- Memory Display
- Power Estimator
- Profiling
- General Purpose Registers
- Special Purpose Registers
- Command Line Interface
- Link Utilization
- Image display

One of the most useful features of the *Cmpware CMP-DK* is the ability to *extend* its functionality. The existing *Cmpware CMP-DK* can be added to and modified in a variety of ways. New processor models may be added, new multiprocessor models constructed with arbitrarily complex mixes of processors and interconnection networks, and new displays may be added for specific applications. Unlike many other systems, these enhancements and modifications do not require access to the system source code. In particular the modeling of processors has been simplified and the

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



performance increased by as much as 800% by the new *FastModel* interface. This document describes the *FastModel* for modeling processors in the *Cmpware CMP-DK*.

Modeling Processors in the Cmpware CMP-DK

The *Cmpware CMP-DK* is used to model multiple processors communicating across shared memory or direct communication channels. In general, the processors are standard microprocessor cores such as *MIPS32*, *Sparc-8* and others. The underlying power of the *Cmpware CMP-DK* is that while each processor can execute a different instruction set, they all share a common interface. This permits the processors to be manipulated in a standard fashion and permits uniform display of data even across heterogeneous multiprocessor systems.

The original modeling environment for the *Cmpware CMP-DK* is based on a simple but flexible interface. It defines five functions (methods) which must be implemented by the model builder to provide the functionality for a processor. Figure 1 shows the interface for the *Cmpware Processor* class.

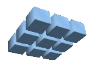
```
public abstract int decode(int instr)
      throws IllegalOpcodeException;

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

public abstract int getPC();

public abstract void setPC(int pc);

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

Figure 1: The *Cmpware CMP-DK Processor* interface.

While this is a flexible and powerful interface, it provides little structure to the model builder and requires that some potentially complex code be written. While the *ProcGen* tool aids further in this effort, it simply provides further structure for the programmer to 'fill in the blanks'. In particular, the decode functionality can become complex in some architectures and providing for support for more irregular structures such as variable instruction lengths can require a more substantial effort.

The *FastModel* interface is a new layer of abstraction placed on top of the existing

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



Processor model that allows the user to supply a minimal set of code and data describing the architecture. *FastModel* uses this data to produce structures such as the **`decode(), dasm()`** and **`execute()`** methods at run-time. In general, the *FastModel* interface takes basic information from a databook describing a processor architecture and permits it to be entered in a compact and highly structured format. This data is then used to drive the processor simulation and IDE display.

This approach results in smaller, simpler and more reliable *Processor* models, allowing faster development and easier modification. And because the *FastModel* is a superset of the existing *Processor* interface, all lower level functionality is still available to the modeling should such functionality be required. Because the system interface to the model remains the same, *FastModel* and standard *Processor* models may be freely intermixed in a multiprocessor model. Complete backward compatibility with the existing *Processor* model is also maintained.

Additionally, the *FastModel* also provides a simple stand-alone assembler and disassembler driven by the processor model description input. There are several reasons why this is valuable, particularly to anyone developing custom processors.

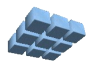
First, the testing of a new processor, even in the *Cmpware CMP-DK* will require some sort of programming tool. Typically this takes the form of a simple assembler. The development of an assembler or other programming tool can be a significant effort at least as large, if not larger, than the architecture modeling. And because the implementation of the assembler and other tools must necessarily be in tight synchronization with the processor design, its development can significantly extend the overall development time.

Secondly, the manually produced assembler will almost certainly contain programming errors, which will further delay progress, since it is typically difficult to determine if the error is in the simulation model or in the assembler code. Finally, as even small changes to the architecture occur, keeping the assembler and other tools in version synchronization can become difficult. Using outdated versions of tools is a common problem and can further consume valuable development time.

With the *FastModel* approach, the model and the programming tools are generated at the same time from the same model description. This not only provides a programming tool suitable for use with the *Cmpware CMP-DK* immediately, but provides a tool which is highly reliable and always completely in sync with the most current version of the architecture model. Such a tool can dramatically reduce the development cycle time and permit significant refinement of the architecture in a much shorter time than with other approaches.

While standard processor models will often have more mature and powerful

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



programming tools available, including high level language compilers, the *FastModel* approach to building *Cmpware* simulation models can significantly accelerate the process of developing custom processors.

The FastModel Interface

The *FastModel* programming interface is significantly smaller than the *Processor* programming interface. In fact, it contains a single function call as indicated in Figure 2. While this interface is extremely simple, the data structures used by the interface are more complex. One way to view the *FastModel* interface is as a data-driven model, where much of the information describing the architecture to be simulated is defined in data structures, as opposed to executable code. This provides a more rigid structure for the processor definition and reduces the overall model development effort, simplifying both development and debug.

```
public void defineInstructions(Instruction i[]);
```

Figure 2: The *Cmpware CMP-DK FastModel* interface.

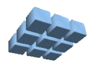
The example used in this document to describe the *FastModel* tool is the '*Simple*' processor example used to describe the original *Processor* models. The *Simple* processor and its model are described in the Cmpware Inc. *Processor Modeling Guide* available on the Cmpware, Inc. web site.

Appendix A at the end of this document gives the full source code for the *FastModel* version of the *Simple* processor, called *FastSimple*. Because the data structures are nested, it is perhaps best to explore the data structures from the end of the file, working backward.

Figure 3 shows the *Instruction* arrays used by the **`defineInstructions()`** interface. While any technique used to produce an array of Instruction elements can be used by the *FastModel* interface, this is perhaps the simplest and most manageable. An array of new *Instruction* objects is allocated as a table using the standard class constructor for the *Instruction* class.

Note that for demonstration purposes, two arrays are declared here. In this case the **`defineInstructions()`** method is called twice, once for each group of instructions. This ability to define groups of instructions is useful in complex processors that may

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



have optional instructions that may be enabled or disabled for various similar versions of the processor hardware.

```
/** A group of Simple processor instructions */
private final Instruction instructions[] = {
    new Instruction("add",  Decoder.primary(f_opcode, 1), new ADD()),
    new Instruction("addi", Decoder.primary(f_opcode, 2), new ADDI()),
    new Instruction("sub",  Decoder.primary(f_opcode, 3), new SUB()),
    new Instruction("xor",  Decoder.primary(f_opcode, 4), new XOR()),
    new Instruction("not",  Decoder.primary(f_opcode, 5), new NOT_()),
    new Instruction("or",   Decoder.primary(f_opcode, 6), new OR()),
    new Instruction("and",  Decoder.primary(f_opcode, 7), new AND()),
};

/** Another group of Simple processor instructions */
private final Instruction instructions2[] = {
    new Instruction("shl", Decoder.primary(f_opcode, 8), new SHL()),
    new Instruction("shr", Decoder.primary(f_opcode, 9), new SHR()),
    new Instruction("br",  Decoder.primary(f_opcode, 10), new BR()),
    new Instruction("bnz", Decoder.primary(f_opcode, 11), new BNZ()),
    new Instruction("bz",  Decoder.primary(f_opcode, 12), new BZ()),
    new Instruction("ld",  Decoder.primary(f_opcode, 13), new LD()),
    new Instruction("st",  Decoder.primary(f_opcode, 14), new ST())
};
```

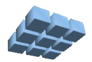
Figure 3: The *FastSimple Instruction* definition arrays.

In this constructor, three parameters which describe the instruction are passed as input parameters. These are:

Name: The first parameter to the *Instruction* class constructor is the instruction name. This is simply a text string describing the instruction. This is typically the mnemonic for the instruction and is used in places such as the assembler, disassembler and other instruction displays.

Decode: The second parameter to the *Instruction* class constructor is the *Decode*. This defines the constants used to decode the particular instruction. In cases of a simple one field decode as in the *FastSimple* processor, the statement **`Decode.primary(f_opcode, 14)`**, will suffice. Here the first parameter defines the field (in this case the field is named **`f_opcode`**) and the second parameter defines the value of this field in this particular instruction. More complex decoding schemes with multiple fields are relatively simple. The secondary fields may be added in using the **`add()`** function. An example of this syntax would be:

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



```
Decode.primary(f_opcode, 14).add(opcode2, 5)
```

Additional fields can be added by chaining additional **add()** methods in this same manner.

Function: The fourth and final field describing an *Instruction* is the *Function*. It contains all of the code implementing the behavior of this instruction. These functions typically make use of local variables and other data in the *FastModel* and are usually very small. Each of the instructions in the *FastSimple* processor have a function containing a single line of code, which is typical. The *Function* classes can use any name, but this example uses the instruction name in all upper case letters.

With the instruction name, decode and function all of the information required to fully describe the instruction has been provided. The last item in each instruction, the function definition, represents the implementation of each instruction. This relies on further interfaces and pre-defined data structures that simplify the task of implementing instructions.

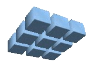
```
/** The ADD instruction */
public class ADD extends RegToReg {
   public void exec(long addr) {
      r[c] = r[a] + r[b];
      }  /* end exec() */
}  /* end class ADD */
```

Figure 4: The *FastSimple* ADD instruction.

The first instruction defined in the *Instruction* array is *ADD*. Figure 4 gives the actual implementation of the *ADD* instruction. A single function, **exec()**, is defined to describe the execution of the instruction. In this case, register a is added to register b and the result stored in register c. This functionality is typically defined in some sort of pseudocode in processor definition documents. The implementation should usually look very similar to the pseudocode definition in these documents.

Of course, this is not the complete instruction implementation. The class *ADD* is derived from another class called *RegToReg* and the local variables *a*, *b* and *c* need to be defined. The *RegToReg* class is shown in figure 5. This class defines all of the common functionality for the register to register operations in the *FastSimple* processor. Most processors typically have groups of instructions that share common decode fields and have somewhat similar functionality. While not required, making use of this

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



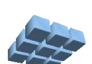
grouping of instructions will help keep the *FastModel* code simple and easy to implement and debug.

```
/**
**  The Register to Resister instruction type
*/

public abstract class RegToReg extends Function {

   public Field[] getFormat() {
      return (new Field[]{f_c, f_a, f_b});
      }  /* end getFields() */

   public int getSize() {return (16);}

   public void extractFields(BigInteger  instr) {
      a = (int) f_a.get(instr);
      b = (int) f_b.get(instr);
      c = (int) f_c.get(instr);
      }  /* end extractFields() */

   public String toString() {
      return ("r"+c+", r"+a+", r"+b);
      }  /* end toString() */

   /** The 'a' field */
   protected int  a;

   /** The 'b' field */
   protected int  b;

   /** The 'c' field */
   protected int  c;

   }  /* end class RegToReg */
```

Figure 5: The *FastSimple RegToReg* instruction group class.

The functionality in the *RegToReg* class is relatively simple, but very important. There are four methods in this class and three local variables. These provide the following functionality:

- **getFormat()**: This method is optional and supplies information on the instruction format and is used only by the assembler. If you have no need for the *FastModel* assembler, then this method can be ignored. If it is implemented, an array of *Field* definitions is returned. These fields represent all of the non-decode fields used in the

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.


assembly language for the instruction. These fields should be in the same order as the assembly language and should have a one to one correspondence to the text fields in the assembly language definition for the instruction. The *Field* data structures will be discussed in more detail below.

- **getSize()**: This method simply returns the number of bits in the instruction. This method typically returns a constant integer value.

- **extractFields()**: This method is used to take the binary representation of the instruction and break it into pieces that can be more easily used by the **`exec()`** method. It is highly recommended that this method make use of locally defined values. This method both simplifies the final **`exec()`** operation and helps accelerate execution. In general, this method should always look very much like the one in this example. *Field* data structures are used to split the instruction into pieces and saved in the local variables.

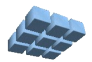
- **toString()**: Finally, the **`toString()`** method is a typical Java method used to output a string representation of a class. In this case, the disassembly of the instruction is returned. Note that this disassembly only contains the parameters to the instruction, since the instruction name, or mnemonic, is not defined at this point. This method, along with the instruction mnemonic, are used to produce the complete instruction disassembly. While this method is not crucial to the operation of the simulator, it is very useful and care should be taken to implement it correctly. Additionally, it is recommended that this method be kept simple and use primarily the local variables.

While the *RegToReg* class defines the register to register instruction, other similar classes are used to define other groups of instructions. In the *FastSimple* processor, these instruction groups are defined by the classes *Immediate*, *Unary*, *Branch*, *BranchImmediate* and *LoadStore*. This is a relatively large number of groups for such a simple processor with a small number of instructions. Other processors will typically have many instructions in a group.

The final element of the *FastModel* is the *Field* definitions. These are used to define the various bit fields in the processor instruction word. These are typically shared and may be declared globally to the *FastModel* class as in Figure 6. These six fields are all of the fields used in the processor definition. They are used in various combinations in the individual instructions.

The *Field* class constructor takes two parameters, the field width and the start bit of the field. A third optional parameter is a symbol table used by the *FastModel* built-in assembler. If your processor already has tools available and will not be using the assembler feature, these symbol tables are not necessary.

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



In the *FastSimple* processor fields, a symbol table is used to translate register names to the binary values used in the instruction word.

```
/* The Simple instruction set decode fields */
private final static Field f_opcode = new Field(4, 12);
private final static Field f_c = new Field(4, 8, regSymbols);
private final static Field f_a = new Field(4, 4, regSymbols);
private final static Field f_b = new Field(4, 0, regSymbols);
private final static Field f_imm8 = new Field(8, 0);
private final static Field f_imm12 = new Field(12, 0);
```

Figure 6: The *FastSimple* Instruction fields.

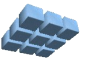
Figure 7 shows the *regSymbols* symbol table used in the *FastSimple* processor. This maps the register name strings to the appropriate integer value used in the instruction. Note that the *Symbol* constructor takes two parameters, a string and its corresponding integer value. Again, these tables may be implemented in a variety of ways, but this approach is the most common. Also note that more than one string may be given per integer value. This will permit different 'alias' values of a register to be used in the assembler. For instance, the first special register may be referred to as the *Program Counter* ('*pc*') or simply as *Special Register Zero* ('*sr0*').

```
/** The Registers Names / Assembler Symbols */
private final static Symbol regSymbols[] = {
   new Symbol("r0", 0),   new Symbol("r1", 1),
   new Symbol("r2", 2),   new Symbol("r3", 3),
   new Symbol("r4", 4),   new Symbol("r5", 5),
   new Symbol("r6", 6),   new Symbol("r7", 7),
   new Symbol("r8", 8),   new Symbol("r9", 9),
   new Symbol("r10", 10),   new Symbol("r11", 11),
   new Symbol("r12", 12),   new Symbol("r13", 13),
   new Symbol("r14", 14),   new Symbol("r15", 15)
   };
```

Figure 7: The FastSimple register symbol table.

While this describes all of the basic data structures and code to produce a working *FastModel* of a processor, there is some additional support available to enhance the flexibility of the model and to make the coding simpler. Some of this additional support

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



is in the form of available Application Program Interface (API) functions that can be overridden and used in a variety of ways. These are all documented in the on-line Java package documentation for the *FastModel* classes, but some of the more useful extensions and techniques are discussed briefly in the following section.

Additional Support in the FastModel Interface

The basic *FastModel* programming interface is sufficient for producing a simulation model and associated stand-alone assembler and disassembler tools. Note that while the *FastModel* interface replaces much of the older *Processor* interface, much of that interface is still exposed and may still be used. While the use of this interface is largely optional, some methods should be used in most processor definitions. The *FastSimple* constructor shown in *Appendix A* makes several calls to this interface to define the register sets and to resize the memory. This constructor is typical and similar definitions should be used in all *FastModel* processor models.

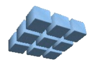
Finally, some methods in the *Processor* model may be overridden just as in normal *Processor* models. **`setPC()`** and **`getPC()`** are two candidates for methods from the *Processor* interface that may be overloaded and redefined. The default value in *FastModel* sets the *Program Counter* to *Special Register zero* (**`sr[0]`**). If this is not correct, then these methods should be overridden.

It should be noted that the overriding of internal methods in the *Cmpware API* is a powerful technique, and care should be taken to completely understand the effect of specifying these methods. In particular, the *FastModel* approach hides this lower level interface in order to provide a simpler processor specification. Mixing *FastModel* and *Processor* methods should only be attempted after a thorough understanding of the function and interaction of these components. If such low-level control is required, it is possible that the *FastModel* approach is not suitable for a particular implementation. Consider using the lower level *Processor* model directly in such cases.

The FastModel Assembler and Disassembler Interface

An additional benefit of the *FastModel* programming interface is that a simple set of programming tools in the form of a stand alone assembler and disassembler is generated automatically. In fact, these tools are a part of the *FastModel* processor model and do not even exist as separate files. This ensures that the simulation model and its tools are properly in sync and that the correct version of the tools are being used with the correct version of the simulation model. This can be very valuable in environments where the architecture or the instruction set of the processor is under development and is changing rapidly.
In the example of the *FastSimple* processor, an assembly language syntax is implicitly

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



defined by the definition of the *Fields* in the **`getFormat()`** methods and by the addition of symbol tables to selected *Fields*. Figure 8 shows a very simple assembly language program for the *FastSimple* processor. It adds a constant value to register r3, a zero to r0 and branches back to the beginning of the program at address zero.

```
//  This tests the Simple processor assember.

#define incr  2

addi r3, incr
addi r0, 0
br 0
```

Figure 8: A small assembly language program for the *FastSimple* processor.

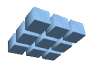
This example also demonstrates two additional features of the built in assembler. First, comments are indicated by the "//" character pair. Other comment markers are the double dash ("--") and the semicolon character (";") Any characters in a line following these comment markers are ignored by the *FastModel* assembler.

```
C:\Cmpware> java com.cmpware.cmp.models.FastSimple -asm Test.asm Test.out
2302 --> addi r3, incr
2000 --> addi r0, 0
A000 --> br 0
3 instructions processed.    (1 #define / 3 comment / 3 blank lines).

C:\Cmpware>
```

Figure 9: Assembling an assembly language program for the *FastSimple* processor.

The second feature is the constant definition indicated by the **`#define`** statement. This is not a general purpose macro facility as in higher level languages, but simply a text string substitution. This permits constants and variables to be defined using more meaningful names. In this example, the increment value is indicated by the text string "incr" and is set to a value of "2".

The assembler command in Figure 9 produces a binary file named *Test.out* that contains the result of the assembly. This file is a 'raw' binary and contains only the assembled code in binary form. This file may be disassembled using the stand alone disassembler created in the *FastModel* for the *FastSimple* processor. Figure 10 shows

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.


the disassembler running on this output file.

```
C:\Cmpware> java com.cmpware.cmp.models.FastSimple -dasm Test.out
2302   addi  r3, 2
2000   addi  r0, 0
A000   br  0

   (3 instructions disassembled).

C:\Cmpware>
```

Figure 10: Disassembling a binary file for the *FastSimple* processor.

Note that the output should be the same as the input to the assembler. This disassembler, in addition to being a simple stand alone tool has other uses. It may be used to help test and verify the model and to be sure that the instruction decode portion of the simulation model is operating correctly. Since this is typically the most error-prone and most difficult part of the model to debug, it is useful to have independent methods of verifying the correctness of this code.

```
   ...
   new Instruction("bz",   branch,    Decode.primary(opcode, 12), new BZ()),
   new Instruction("ld",   loadStore, Decode.primary(opcode, 13), new LD()),
   new Instruction("st",   loadStore, Decode.primary(opcode, 13), new ST())
};
   ...
```

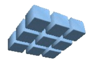
Figure 11: A duplicate opcode error inserted into the *FastSimple* model.

Finally, this stand alone disassembler uses the same code used in the *Cmpware CMP-DK* user interface. If some changes to the disassembly format need to be made, the *FastModel* disassembler gives a simple way to test this portion of the system.

One final note on the *FastModel* assembler / disassembler tools. It is useful to run these at some point after model development just to do some further verification on the models. In particular, the instruction decode functionality is tested in these modules for consistency. If two instructions specified in the model use the same decode values, the assembler and disassembler will produce a warning message.

This can be very useful when developing *FastModel* models of processors. The typing

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



in of the decode values can be tedious and typographical errors can easily go unnoticed. Such errors can be difficult to find in the final simulation model running under the *Cmpware CMP-DK* environment.

```
C:\Cmpware> java com.cmpware.tools.AutoSimple -dasm Test.out
Warning:  Instruction 'ld' has the same decode as instruction 'st'.
2302   addi  r3, 2
2000   addi  r0, 0
A000   br  0

   (3 instructions disassembled).

C:\Cmpware>
```

Figure 12: The *FastSimple* disassembler warning message from a duplicate opcode error in the model.

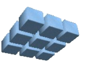
As an example, the instruction opcode for the **st** operation is changed from the value of 14 to the value 13. This causes a decode collision with the **ld** instruction. If this change is made as in Figure 11, a warning message as indicated in Figure 12 is displayed. Note that no such warning message will be produced by the *Cmpware CMP-DK* environment. This is only a feature in the stand-alone assembler and disassembler tools.

Using the FastSimple Processor

Once the *FastSimple* processor has been compiled, it can immediately be used in the *Cmpware CMP-DK*. Since this demonstration processor is included with the other models in the *Cmpware CMP-DK* library, it can be used just like any other standard processor. By selecting the *Cmpware* preferences from the **Windows --> Preferences ...** menu, the *Processors* field can be changed to, in this case, a 4 x 4 array of *FastSimple* processors as shown in Figure 13.

Pressing the **[OK]** button in the *Preferences* window allocates the new array and produces the display as shown in Figure 14. Selecting the *Registers* tab on the left verifies that the number of registers and their names are the same as the ones given in the model.

At his point, the **reset()** code in the model has loaded the first few addresses in memory with some simple instructions, in particular two immediate additions and a loop back to address zero. See the **reset()** code in *Appendix A* for more details on this

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.


piece of embedded executable code. Note that including such executable code after a reset can help to speed up development of processor models. This operates much like the Read Only Memory (ROM) typically used to help boot up most computing systems. In practical terms, it also saves the additional step of manually loading code using the **Load All** ( ) or **Load** ( ) buttons.

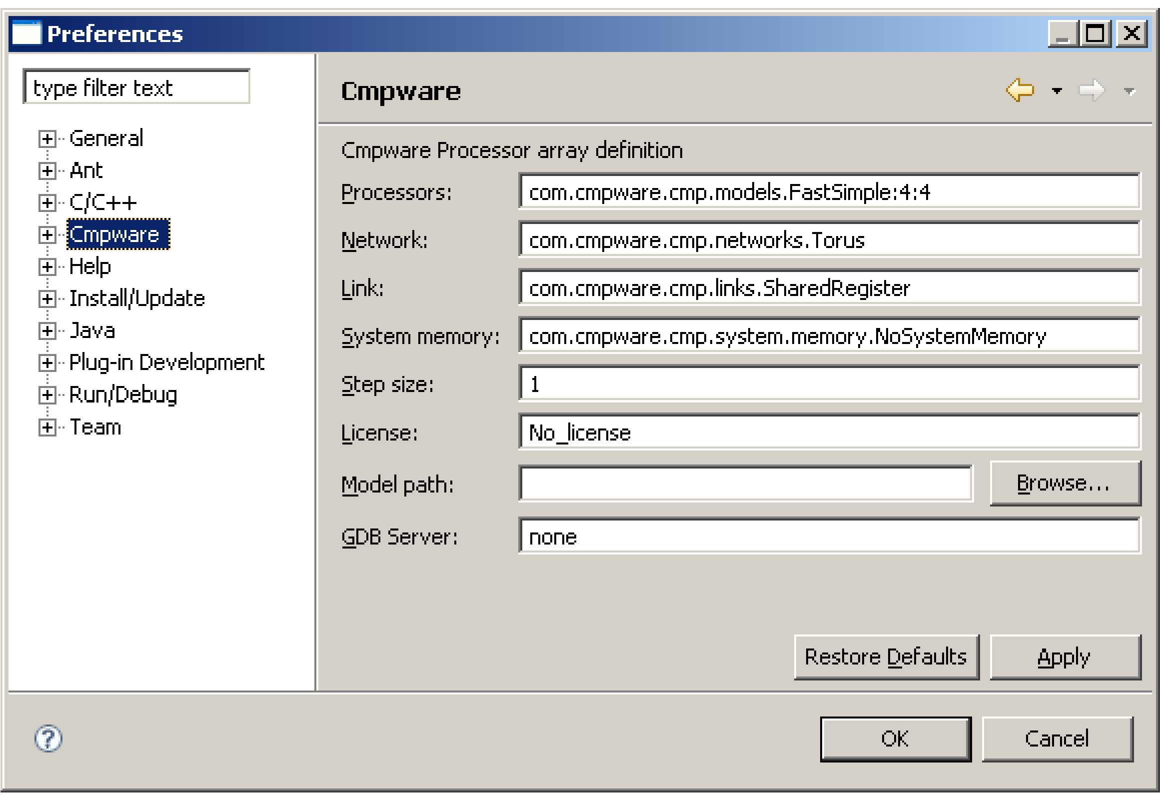
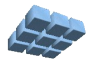


Figure 13: Creating a 4 x 4 array of *FastSimple* processors.

Pressing the **Step** button ( ) will step the multiprocessor simulation through a simulation cycle. Repeated pressing of the Step button will continue to single step the simulation. The code in this example will increment registers and loop infinitely. In this example, each *FastSimple* core operates independently; there is no communication or other interaction between the processor cores. But this will still allow some verification that the model is operating correctly.

Figure 15 shows the *Cmpware* IDE after the *Disassemble* tab has been selected. This switches the main display from the multiprocessor array to a display of the disassembly of memory for the selected processor. As expected, the code defined in the model

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.


`reset()` method can be seen in the disassembly of the first few words of memory.

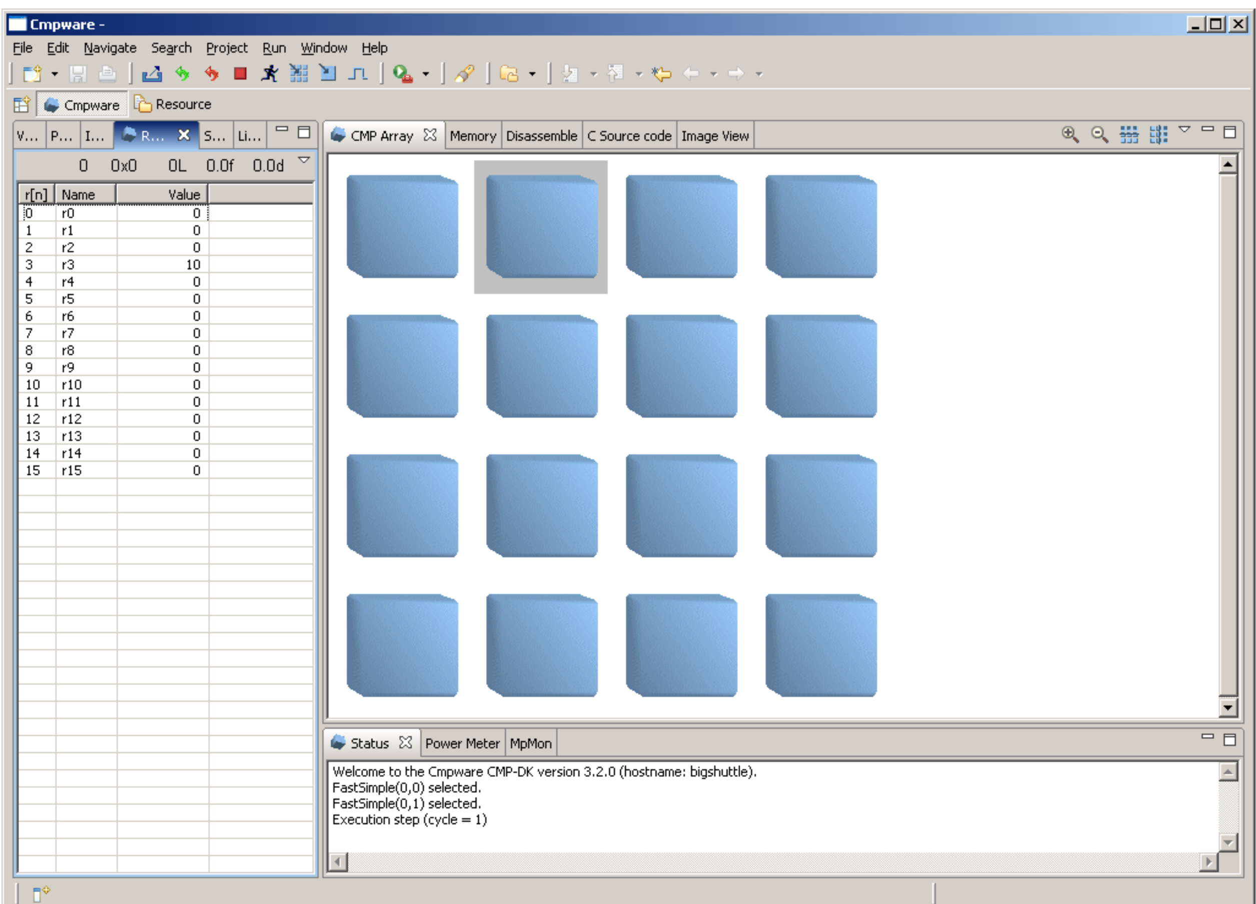
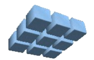


Figure 14: Creating a 4 x 4 array of *FastSimple* processors.

In Figure 15, a single step cycle has been performed and the first instruction has been executed. As expected from the code in the disassembly, the value in register 10 (r10) has been incremented by ten. This is further confirmation that the model is functioning correctly. While it is likely that there will be some errors (bugs) in custom models, the structures and displays in the *Cmpware CMP-DK* make these problems easy to identify and isolate.

It should also be noted that the *Disassembly* display in Figure 15 is produced directly by the *FastSimple* simulation model. In previous versions of the *Cmpware CMP-DK*, this disassembly had to be implemented manually and was a potential source of disagreement between what the display indicated and the function the actual binary code. The *FastModel* approach eliminates this gap, producing not only more reliable and robust simulation models, but also smaller models with less code.

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



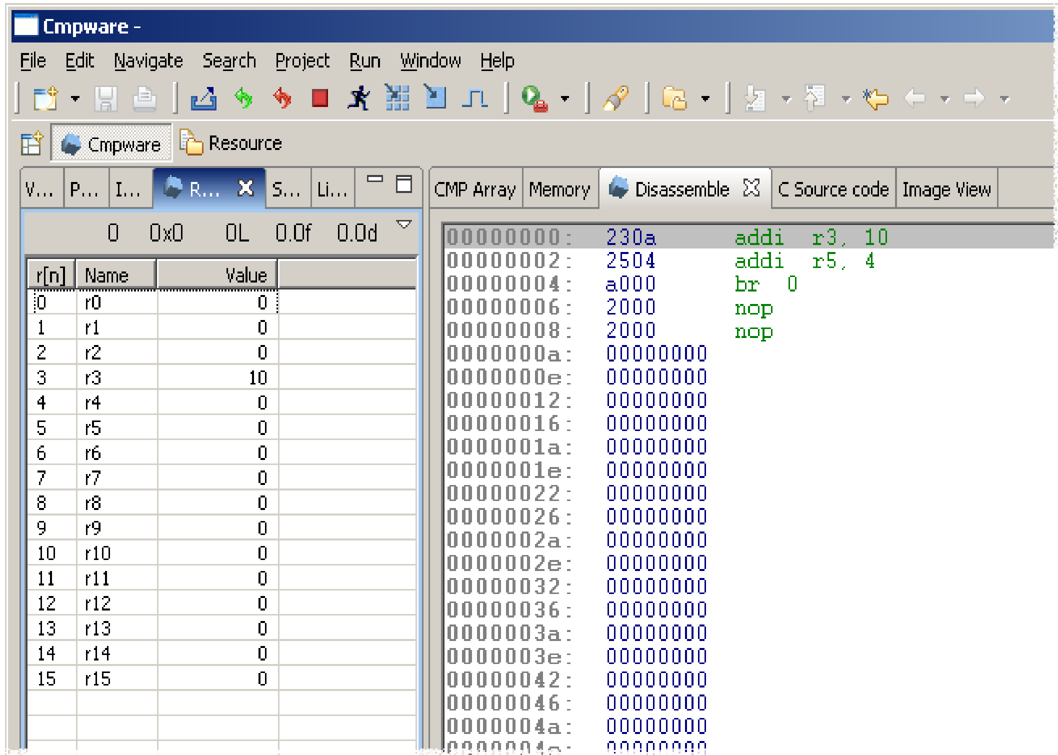


Figure 15: Execution of code in the *FastModel* processors.

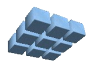
Lastly, it should be mentioned that the disassembly shows the third and fourth instructions as a 'nop'. This 'no-operation' value is set in the *FastSimple* constructor using the **`defineNoop()`** method. The model defines the instruction '`addi r0, 0`' to indicate a '`nop`', and this is the instruction that the disassembler in the IDE displays as the default nop operation. This '`nop`' indicator is only a display convenience, since many other instructions could just as easily been used to produce the '`nop`' functionality.

Conclusions

The *FastModel* package in the *Cmpware CMP-DK* permits users to quickly and easily define and modify processor models. *FastModel* provides a simple, yet flexible framework to provide all of the information necessary to define an instruction set processor. Once this description is provided, *FastModel* makes use of this information in a variety of ways.

While the ability to produce the simulation model that interfaces directly to the

Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



Cmpware CMP-DK IDE is the primary goal, the *FastModel* approach also produces both an assembler and disassembler for the processor. This has several advantages.

First, the effort required to produce these tools is eliminated. Second, these tools are part of the model itself, and in fact all share the same Java class file. This guarantees that the tools will always be in sync with the models. This can be very important in environments where the design in changing rapidly. Finally, the use of the same data to perform simulation, assembly and disassembly provides the opportunity to do a variety of 'sanity' checking, thus greatly reducing the model debug burden.

FastModel represents a significant new feature in the 3.2 release and is recommended for all future processor modeling in the *Cmpware CMP-DK*.

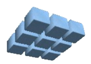
For more information on the *Cmpware CMP-DK* see our web site at:

http://www.cmpware.com/

or send an email to:

info@cmpware.com

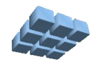
Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



Appendix A: FastSimple.java Source Code

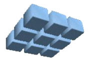
```
package com.cmpware.cmp.models;

import  com.cmpware.cmp.MemoryAccessException;
import  com.cmpware.cmp.Util;
import  com.cmpware.fastmodel.Instruction;
import  com.cmpware.fastmodel.Symbol;
import  com.cmpware.fastmodel.Decoder;
import  com.cmpware.fastmodel.Field;
import  com.cmpware.fastmodel.Function;
import  com.cmpware.fastmodel.FastModel;

import  java.math.BigInteger;


/**
**   This is the 'Simple' processor produced using the
**   FastModel data-driven interface.  Note that this
**   model relies on the use of local (as opposed to global)
**   data in the exec() methods.  The exec() method may
**   reference globlal data such as registers, but all
**   fields extracted from the instruction word must be local
**   to the Instruction class.
**
** <p>
** Copyright (c) 2008 Cmpware, Inc. All Rights Reserved.
** <p>
**
** @author SAG
*/


public class FastSimple extends FastModel {


/** Copyright string */
public final static String copyright =
   "Copyright (c) 2008 Cmpware, Inc.  All Rights Reserved.";


/**
**   This main() method is optional and enables the
**   stand-alone assembler and disassembler.  The
**   command line should be either:
**
**   java com.cmpware.cmp.models.FastSimple -dasm <infile>
**   or
**   java com.cmpware.cmp.models.FastSimple -asm <infile> <outfile>
*
```


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



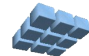
```
**  @param args  the command line arguments.
**
*/

public static void main(String[] args) {
   FastModel.main(new FastSimple(), args);
   }  /* end main() */


/**
**  The constructor
*/

public FastSimple() {

   try {

   String  sregNames[] = {"pc"};
   String  regNames[];

   /* Define the instructions first */
   defineInstructions(instructions);
   defineInstructions(instructions2);

   /* Get some string tables from the instruction data */
   regNames = getSymbolNames(regSymbols);

   /* Some good defaults */
   defineName("FastSimple");
   defineInstructionSize(2);
   defineRegisters(16);
   defineSpecialRegisters(1);
   defineBranchDelay(0);
   defineRegisterNames(regNames);
   defineSpecialRegisterNames(sregNames);
   defineNoop(NOOP_INSTRUCTION);

   /* Resize the memory */
   getLocalMemory().resize(1024);
   getLocalMemory().setEndian(BIG);

   /* Reset the processor */
   reset();

   } catch (Exception e) {
      e.printStackTrace();
   }  /* end try{} */

   }  /* end FastSimple() */


/**
**  This method resets the  processor.  Note that
**  super.reset() must me called to completely intialize
```


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



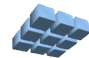
```
**  the processor.
**
*/

public void  reset() {

   /* Always call Processor.reset() first */
   super.reset();

   /* Write the test code to address 0 */
   byte testCode[] = {
      (byte) 0x23, (byte) 0x0a,   // ADDI  r3, 10
      (byte) 0x25, (byte) 0x04,   // ADDI  r5, 4
      (byte) 0xa0, (byte) 0x00,   // BR 0
      (byte) 0x20, (byte) 0x00,   // NOOP
      (byte) 0x20, (byte) 0x00    // NOOP
      };

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

   }  /* end reset() */


/** The Registers Names / Assembler Symbols */
private final static Symbol regSymbols[] = {
   new Symbol("r0", 0),  new Symbol("r1", 1),
   new Symbol("r2", 2),  new Symbol("r3", 3),
   new Symbol("r4", 4),  new Symbol("r5", 5),
   new Symbol("r6", 6),  new Symbol("r7", 7),
   new Symbol("r8", 8),  new Symbol("r9", 9),
   new Symbol("r10", 10),  new Symbol("r11", 11),
   new Symbol("r12", 12),  new Symbol("r13", 13),
   new Symbol("r14", 14),  new Symbol("r15", 15)
   };


/* The Simple instruction set decode fields */
private final static Field f_opcode = new Field(4, 12);
private final static Field f_c = new Field(4, 8, regSymbols);
private final static Field f_a = new Field(4, 4, regSymbols);
private final static Field f_b = new Field(4, 0, regSymbols);
private final static Field f_imm8 = new Field(8, 0);
private final static Field f_imm12 = new Field(12, 0);


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


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



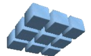
```
/**
**  The Register to Resister instruction type
*/

public abstract class RegToReg extends Function {

   public Field[] getFormat() {
      return (new Field[]{f_c, f_a, f_b});
      }  /* end getFields() */

   public int getSize() {return (16);}

   public void extractFields(BigInteger  instr) {
      a = (int) f_a.get(instr);
      b = (int) f_b.get(instr);
      c = (int) f_c.get(instr);
      }  /* end extractFields() */

   public String toString() {
      return ("r"+c+", r"+a+", r"+b);
      }  /* end toString() */

   /** The 'a' field */
   protected int  a;

   /** The 'b' field */
   protected int  b;

   /** The 'c' field */
   protected int  c;

   }  /* end class RegToReg */


/**
**  The Immediate instruction type
*/

public abstract class Immediate extends Function {

   public Field[] getFormat() {
      return (new Field[]{f_c, f_imm8});
      }  /* end getFields() */

   public int getSize() {return (16);}

   public void extractFields(BigInteger  instr) {
      int  tmp;
      tmp = (int) f_imm8.get(instr);
      simm8 = Util.signExtend(tmp, 8);
      c = (int) f_c.get(instr);
      }  /* end extractFields() */
```


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



```
   public String toString() {
       return ("r"+c+", "+simm8);
       }  /* end toString() */

   /** The signed 8-bit immediate value */
   protected int  simm8;

   /** The 'c' field */
   protected int  c;

   }  /* end class Immediate */


/**
**  The Unary instruction type
*/

public abstract class Unary extends Function {

   public Field[] getFormat() {
      return (new Field[]{f_c, f_b});
      }  /* end getFields() */

   public int getSize() {return (16);}

   public void extractFields(BigInteger  instr) {
      b = (int) f_b.get(instr);
      c = (int) f_c.get(instr);
      }  /* end extractFields() */

   public String toString() {
       return ("r"+c+", r"+b);
       }  /* end toString() */

   /** The 'b' field */
   protected int  b;

   /** The 'c' field */
   protected int  c;

   }  /* end class Unary */


/**
**  The Branch instruction type
*/

public abstract class Branch extends Function {

   public Field[] getFormat() {
      return (new Field[]{f_a, f_b});
      }  /* end getFields() */
```


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.

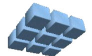


```
   public int getSize() {return (16);}

   public void extractFields(BigInteger  instr) {
      a = (int) f_a.get(instr);
      b = (int) f_b.get(instr);
      }  /* end extractFields() */

   public String toString() {
       return ("r"+a+", r"+b);
       }  /* end toString() */

   /** The 'a' field */
   protected int  a;

   /** The 'b' field */
   protected int  b;

   }  /* end class Branch */


/**
**  The Branch Immediate instruction type
*/

public abstract class BranchImm extends Function {

   public Field[] getFormat() {
      return (new Field[]{f_imm12});
      }  /* end getFields() */

   public int getSize() {return (16);}

   public void extractFields(BigInteger  instr) {
      int  tmp;
      tmp = (int) f_imm12.get(instr);
      simm12 = Util.signExtend(tmp, 12);
      }  /* end extractFields() */

   public String toString() {
       return (""+simm12);
       }  /* end toString() */

   /** The signed 12-bit immediate value */
   protected int  simm12;

   }  /* end class BranchImm */


/**
**  The Load Store instruction type
*/

public abstract class LoadStore extends Function {
```


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.

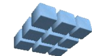
```
   public Field[] getFormat() {
      return (new Field[]{f_c, f_b});
      }  /* end getFields() */

   public int getSize() {return (16);}

   public void extractFields(BigInteger  instr) {
      b = (int) f_b.get(instr);
      c = (int) f_c.get(instr);
      }  /* end extractFields() */

   public String toString() {
       return ("r"+c+", r"+b);
       }  /* end toString() */

   /** The 'b' field */
   protected int  b;

   /** The 'c' field */
   protected int  c;

   }  /* end class LoadStore */



/*
**  The Instruction Functions
*/

/** The ADD instruction */
public class ADD extends RegToReg {
   public void exec(long addr) {
      r[c] = r[a] + r[b];
      }  /* end exec() */
}  /* end class ADD */

/** The ADDI instruction */
public class ADDI extends Immediate {
   public void exec(long addr) {
      r[c] = r[c] + simm8;
      }  /* end exec() */
}  /* end class ADDI */

/** The NOT_ instruction */
public class NOT_ extends Unary {
   public void exec(long addr) {
      r[c] = ~r[b];
      }  /* end exec() */
}  /* end class NOT_ */

/** The SUB instruction */
public class SUB extends RegToReg {
   public void exec(long addr) {
      r[c] = r[a] - r[b];
```


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



```
      }  /* end exec() */
}  /* end class SUB */

/** The XOR instruction */
public class XOR extends RegToReg {
   public void exec(long addr) {
      r[c] = r[a] ^ r[b];
      }  /* end exec() */
}  /* end class XOR */

/** The OR instruction */
public class OR extends RegToReg {
   public void exec(long addr) {
      r[c] = r[a] | r[b];
      }  /* end exec() */
}  /* end class OR */

/** The AND instruction */
public class AND extends RegToReg {
   public void exec(long addr) {
      r[c] = r[a] & r[b];
      }  /* end exec() */
}  /* end class AND */

/** The SHL instruction */
public class SHL extends RegToReg {
   public void exec(long addr) {
      r[c] = r[a] << r[b];
      }  /* end exec() */
}  /* end class SHL */

/** The SHR instruction */
public class SHR extends RegToReg {
   public void exec(long addr) {
      r[c] = r[a] >> r[b];
      }  /* end exec() */
}  /* end class SHR */

/** The BR instruction */
public class BR extends BranchImm {
   public void exec(long addr) {
      branch(simm12);
      }  /* end exec() */
}  /* end class BR */

/** The BNZ instruction */
public class BNZ extends Branch {
   public void exec(long addr) {
      if (r[a] != 0) branch(r[b]);
      }  /* end exec() */
}  /* end class BNZ */

/** The BZ instruction */
public class BZ extends Branch {
```


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.



```
   public void exec(long addr) {
      if (r[a] == 0) branch(r[b]);
      }  /* end exec() */
}  /* end class BZ */

/** The LD instruction */
public class LD extends LoadStore {
   public void exec(long addr) throws MemoryAccessException {
      r[c] = read32(r[b]);
      }  /* end exec() */
}  /* end class LD */

/** The ST instruction */
public class ST extends LoadStore {
   public void exec(long addr) throws MemoryAccessException {
      write32(r[c], r[b]);
      }  /* end exec() */
}  /* end class ST */


/** A group of Simple processor instructions */
private final Instruction instructions[] = {
   new Instruction("add",  Decoder.primary(f_opcode, 1), new ADD()),
   new Instruction("addi", Decoder.primary(f_opcode, 2), new ADDI()),
   new Instruction("sub",  Decoder.primary(f_opcode, 3), new SUB()),
   new Instruction("xor",  Decoder.primary(f_opcode, 4), new XOR()),
   new Instruction("not",  Decoder.primary(f_opcode, 5), new NOT_()),
   new Instruction("or",   Decoder.primary(f_opcode, 6), new OR()),
   new Instruction("and",  Decoder.primary(f_opcode, 7), new AND()),
};

/** Another group of Simple processor instructions */
private final Instruction instructions2[] = {
   new Instruction("shl", Decoder.primary(f_opcode, 8), new SHL()),
   new Instruction("shr", Decoder.primary(f_opcode, 9), new SHR()),
   new Instruction("br",  Decoder.primary(f_opcode, 10), new BR()),
   new Instruction("bnz", Decoder.primary(f_opcode, 11), new BNZ()),
   new Instruction("bz",  Decoder.primary(f_opcode, 12), new BZ()),
   new Instruction("ld",  Decoder.primary(f_opcode, 13), new LD()),
   new Instruction("st",  Decoder.primary(f_opcode, 14), new ST())
};

}  /* end class FastSimple */
```


Copyright © 2004-2008 Cmpware, Inc. All rights reserved.