

AES: Demonstration Application for the Cmpware CMP-DK (Demo Version 2.0 for Eclipse 3.0)

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 uncluttered format. Such capabilities are essential in analyzing the behavior of multiprocessor systems.

This demonstration version of the *Cmpware CMP-DK* (version 2.0) for Eclipse 3.0 and higher contains all features of the standard toolkit, but restricts the simulation model to a 3 x 3 heterogeneous array of MIPS32 and SPARC-8 processors. All simulation capabilities and displays are included. This includes:

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

Demonstration Applications

Available for use with the *Cmpware CMP-DK* version 2.0 is a series of demonstration applications which are presented to introduce some of the features in the *CMP-DK*. These applications start with small, simple programs gradually building up to more complex applications exploiting relatively low-level parallelism. These demonstrations stand alone and can be studied in any order, but it is best to start with the early examples, which are smaller and simpler and build up to the larger ones. This provides



a tutorial-like introduction to the features in the *Cmpware CMP-DK*.

While these demonstrations cover the application development aspects of this tool, much of the power in the *Cmpware CMP-DK* is in the ability to quickly model relatively complex multiprocessor systems. This modeling activity is reserved for licensed copies of the software. For more information on getting licensed copies of the *Cmpware CMP-DK*, contact Cmpware at info@cmpware.com.

The groups of files in this tutorial package are as follows:

- **Introduction** - An introduction to all of the applications
- **Simple** - A simple, single processor test application
- **Ping Pong** - a simple two processor application
- **Hetero** - the Ping Pong application on two different types of processors
- **FIR Filter** - A multiprocessor Finite Impulse Response (FIR) Filter
- **AES Encryption** - A multiprocessor AES encryption implementation
- **FFT Filter** - a multiprocessor FFT filter using shared memory
- **FFT Filter 2** - a multiprocessor FFT filter using communication channels

These example applications assume that the *Cmpware CMP-DK* has already been successfully installed on your system. For more information on acquiring and installing either the free demonstration version or the fully licensed version, see the Cmpware web site.

The source and compiled code for these demonstration applications can be downloaded from the Cmpware Web site as a compressed ZIP archive at:

http://www.cmpware.com/Apps/CmpwareApps_2_0.zip

The AES Encryption Application

The *Advanced Encryption Standard (AES)* application extends the ideas in the previous example applications. In particular this application builds on the implementation in the FIR filter example. It is recommended that the FIR filter example be examined before moving on to *AES*. Many of the programming ideas and the *Cmpware CMP-DK* tool usages are similar and will not be repeated in any detail in this demonstration. In particular, this application shows another example of exploiting fine grained parallelism as well as some interesting parameterization techniques.

The *AES* code is all in the compressed ZIP archive under the *AES* directory, and



contains source code, Makefile, linker directives file, compiled relocatable object files and finally, fully linked executable ELF files. In fact, all of the demonstration applications will contain these types of files. All have been built using the *Gnu GCC* compiler with a version higher than 3.0. If you have access to a *MIPS32* compiler which produces standard *ELF* executable with *DWARF2* debug information, you may modify these files and re-compile them and test the results and use them in the *Cmpware CMP-DK*.

Running the Self Hosted AES Application

The most of the previous demonstration applications were all very simple and produced no particular result. They were mostly used as vehicle for demonstrating features in the *Cmpware CMP-DK*. The *FIR* application, however, was more complex and used real input data to produce a real output. The AES application is similar in structure to the FIR filter application.

In order to simplify the development process and to create a 'benchmark' for future comparisons, a single processor version of the AES application is first developed. This approach has several benefits. First, developing a single processor version of any algorithm is typically much easier than building a multiprocessor version. This allows the processes of algorithm design and implementation to be separated from the process of parallelization.

Once the algorithm is working correctly, efforts to parallelize it can proceed. Experience has shown that this process is a much simpler path than attempting to simultaneously code and parallelize an algorithm. In particular, debugging is simplified, since the single processor implementation can be tested for logical correctness, then the results of the parallel version can be compared to the results generated by the single processor version.

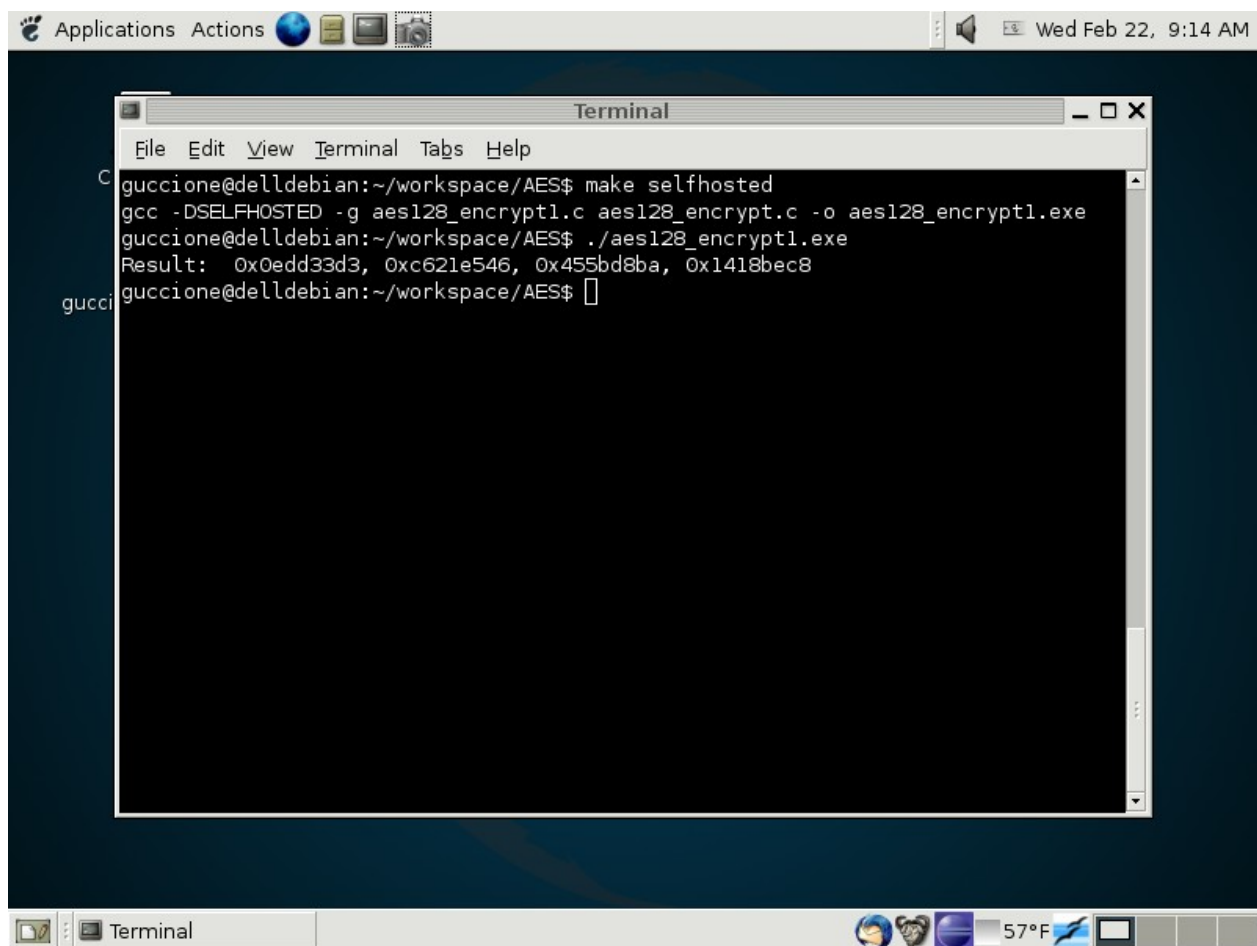
Figure 1 shows the building and execution of the single processor AES application. This is compiled and run on a standard workstation, in this case a Linux system. The flag "**-DSELFHOSTED**" is passed to the compiler and is used to indicate that the single processor is a full development system, in particular, one containing an operating system and display capabilities. This permits the output to be printed to the console as in Figure 1.

The source code to this single processor version is in *Appendix B* at the end of this document. This single processor version of the code uses the routines in *Appendix A*. These routines are completely serial code and are based on the standard AES implementation downloaded from the AES web site:



<http://csrc.nist.gov/CryptoToolkit/aes/rijndael/>

The code used in this example is not the full encryption algorithm, but only the 128-bit key size and only performs the encryption portion of the algorithm. Other decryption and larger key sizes can easily be added to this example, but focusing on a single aspect of the AES standard makes the code simpler for demonstration purposes. Figure 1 shows the building and execution of the self hosted implementation. Note that the SELFHOSTED flag is used to control the output. Specifically, in a self-hosted system, the output is sent to a local variable and printed to the standard output.



```
Applications Actions [Globe] [Folder] [Terminal] [Network] [Speaker] Wed Feb 22, 9:14 AM

Terminal
File Edit View Terminal Tabs Help
guccione@delldebian:~/workspace/AES$ make selfhosted
gcc -DSELFHOSTED -g aes128_encrypt1.c aes128_encrypt.c -o aes128_encrypt1.exe
guccione@delldebian:~/workspace/AES$ ./aes128_encrypt1.exe
Result: 0x0edd33d3, 0xc621e546, 0x455bd8ba, 0x1418bec8
guccione@delldebian:~/workspace/AES$
```

Figure 1: The self hosted AES application.

The result output by this self-hosted implementation is a series of four 32-bit integers,



printed in hexadecimal. From the source code in *Appendix B*, we see that the plaintext is zero, and that the key used is a 128-bit value with only the high bit set. The NIST website supporting the AES standard also supplies a large amount of verification data, including outputs for various input and key combinations. From the file `ecb_vk.txt` as partially reproduced in Figure 2, the encrypted text ("CT") value matches the output from the self-hosted code in Figure 1.

```
=====
FILENAME:  "ecb_vk.txt"

Electronic Codebook (ECB) Mode
Variable Key Known Answer Tests

Algorithm Name: Rijndael
Principal Submitter: Joan Daemen

=====

KEYSIZE=128

PT=00000000000000000000000000000000

I=1
KEY=80000000000000000000000000000000
CT=0EDD33D3C621E546455BD8BA1418BEC8

I=2
...
```

Figure 2: The `ecb_vk.txt` file

Once the AES filter code for a single processor has been successfully developed in a friendly self-hosted environment, the code may be moved to the *Cmpware CMP-DK* development environment. The first step will be to verify that the uniprocessor code still operates correctly on the uniprocessor model in the *Cmpware CMP-DK*. Once this is verified, parallelizing the code into a multiprocessor implementation can begin.

Rather than compiling the code with the standard GCC compiler on the self hosted system, a cross-targeted compiler running on the host system, but generating code for the *MIPS32* processor is used. Additionally, since this embedded *MIPS32* processor does not have dedicated operating system or IO support, the **-DSELFHOSTED** flag is not used. This produces code which sends the AES results to a pre-specified memory location. The compiled code for a *MIPS32* processor is supplied in the *AES* directory



as `AES1.elf`.

Running the Single Processor AES Application

To execute this uniprocessor AES code, the *Cmpware* perspective in Eclipse must first be opened. This is typically done from the Eclipse main menu using the **Window --> Open Perspective --> Cmpware** menu command. If you have problems getting this view to come up, or have not installed the *Cmpware CMP-DK*, see the installation guide available on the *Cmpware* web site. It will guide you in installing the software.

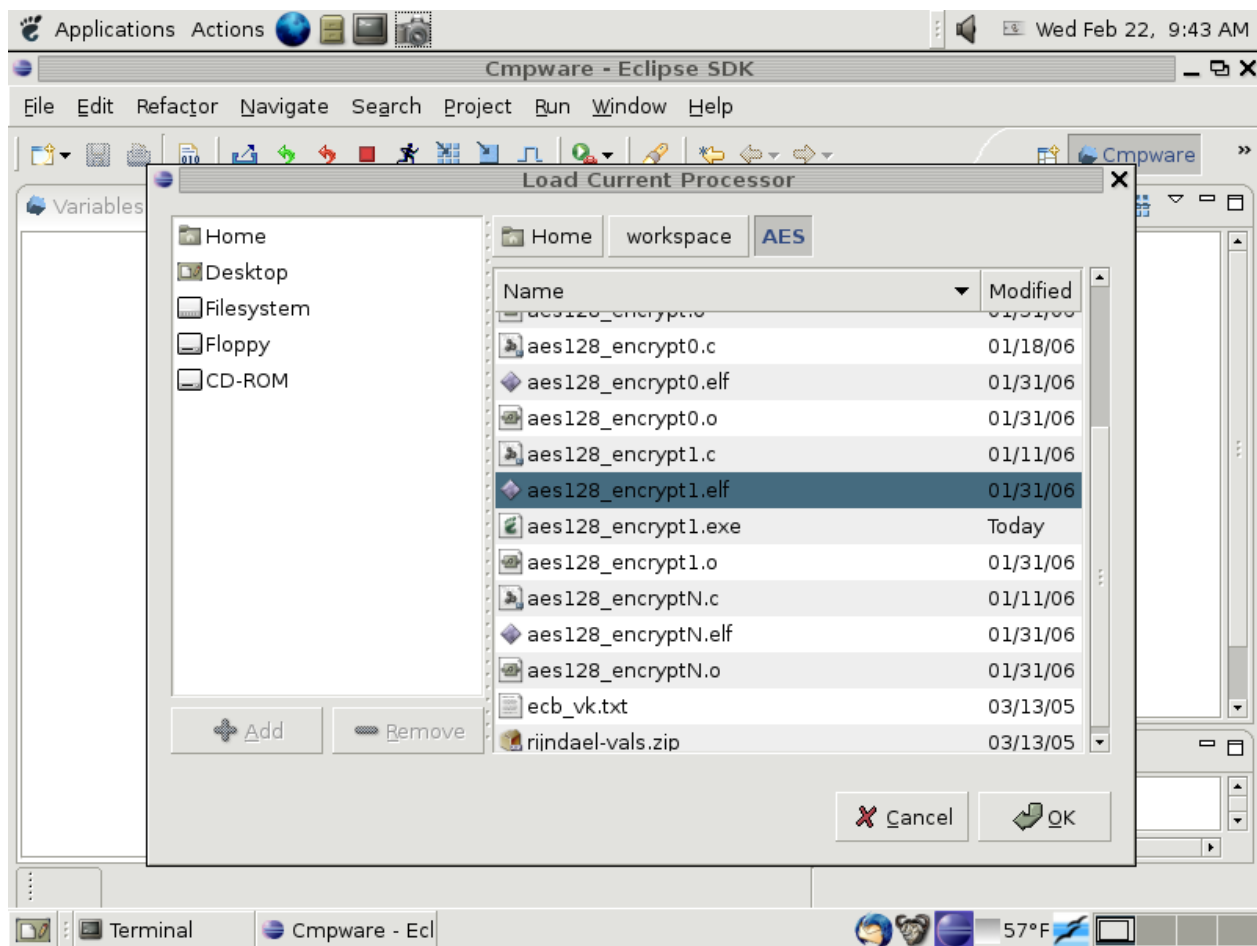


Figure 3: Loading `aes128_encrypt1.elf` into node (0,0).

The *Cmpware CMP-DK* used in this example is the demonstration version of the



software and begins with the default 3 x 3 array of processors. The first row contains three *MIPS32* processors, the second row three *Sparc-8* processors, and the third row contains another three *MIPS32* processors.

Like the previous examples, executable code is loaded into the first processor in the upper left corner of the array. To load this processor with executable code, select the processor with the mouse. It should be highlighted with a grey background and the **Status** window at the bottom should indicate that the processor **MIPS32(0,0)** is selected.

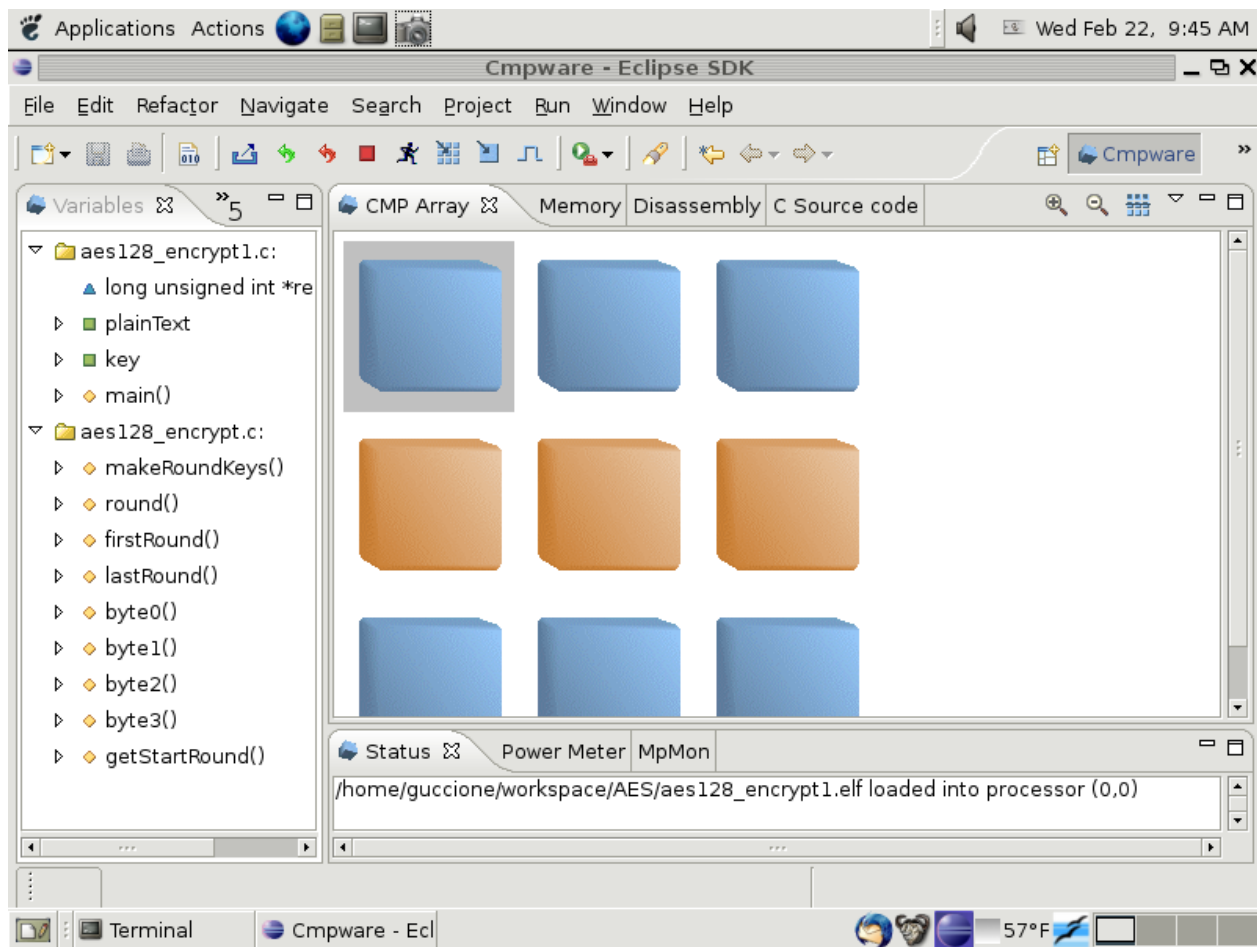




Figure 4: *aes128_encrypt1.elf* loaded into node (0,0).

Use the **Load** button () to bring up a file selection dialog. Using this file selection dialog, select the *aes128_encrypt1.elf* file from the list of files for the AES



demonstration as shown in Figure 3. A message in the **Status** window at the bottom of the IDE should indicate that the file was successfully loaded into the MISP32 processor at location (0,0) as in Figure 4.

At this point, the executable file *aes128_encrypt1.elf* is loaded into the processor in the upper left corner of the processor array. Clicking on the **Step** button () advances the global clock in the simulation and updates the displays in the *Cmpware CMP-DK*. In the view in Figure 5, the multiprocessor has been stepped through 26 cycles, as indicated by the **Status** window.

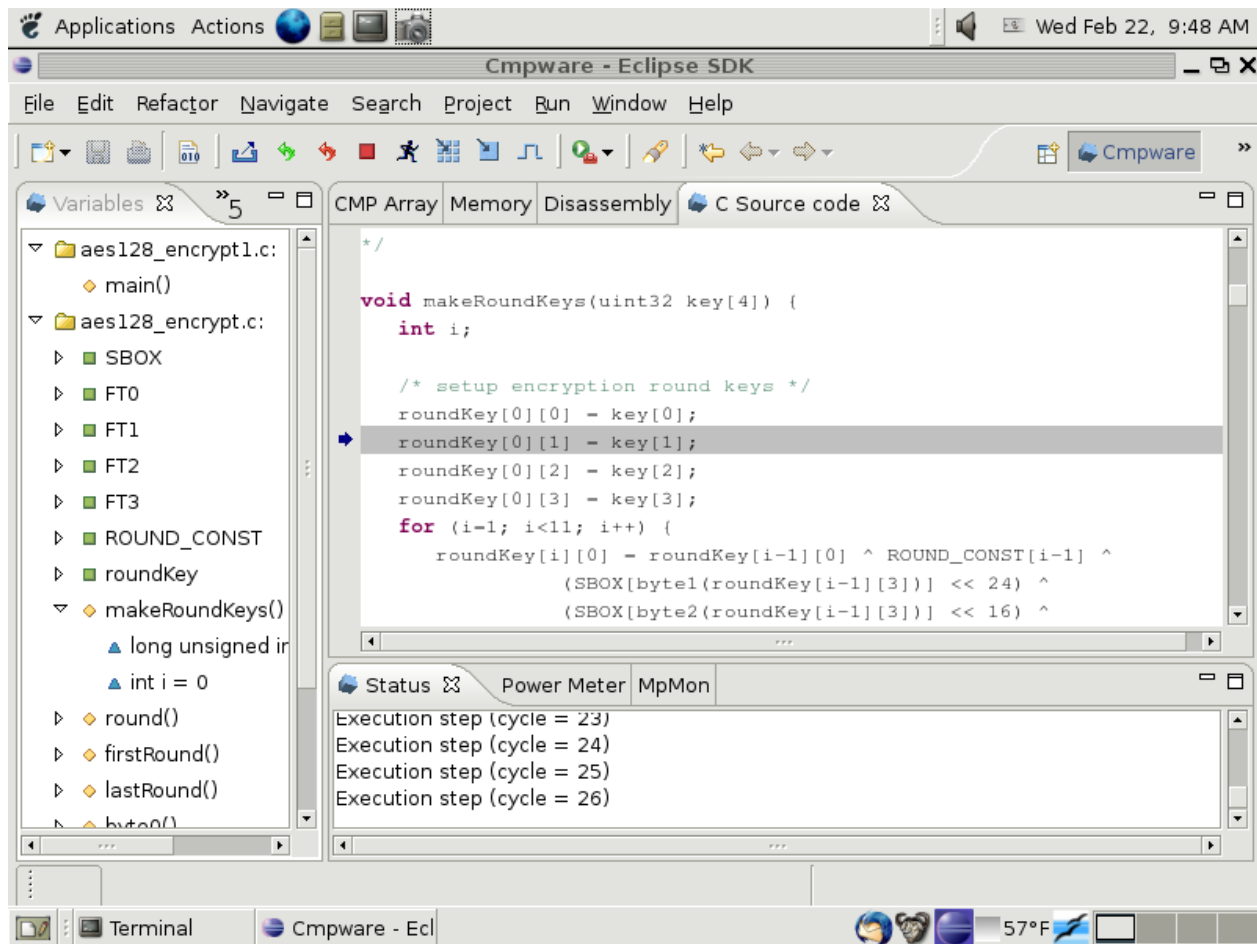



Figure 5: *aes128_encrypt1.elf* executing on node (0,0).

For larger applications such as the AES filter, the single stepping with the **Step** button () one cycle at a time quickly becomes tedious. The *Cmpware CMP-DK* is designed



to permit a configurable step size for the simulation. This permits larger sections of code to be executed with a single step.

Like most parameters in the Cmpware *CMP-DK* the step size is set in the Cmpware **Preferences Page**. This is set from the menu items **Windows --> Preferences**. This brings up the **Preference** dialog box in Figure 6. Selecting **Cmpware** from the list on left brings up the preferences for the *Cmpware CMP-DK*. In this case, the step size is set to 1000. Pressing the **[OK]** button will accept this value.

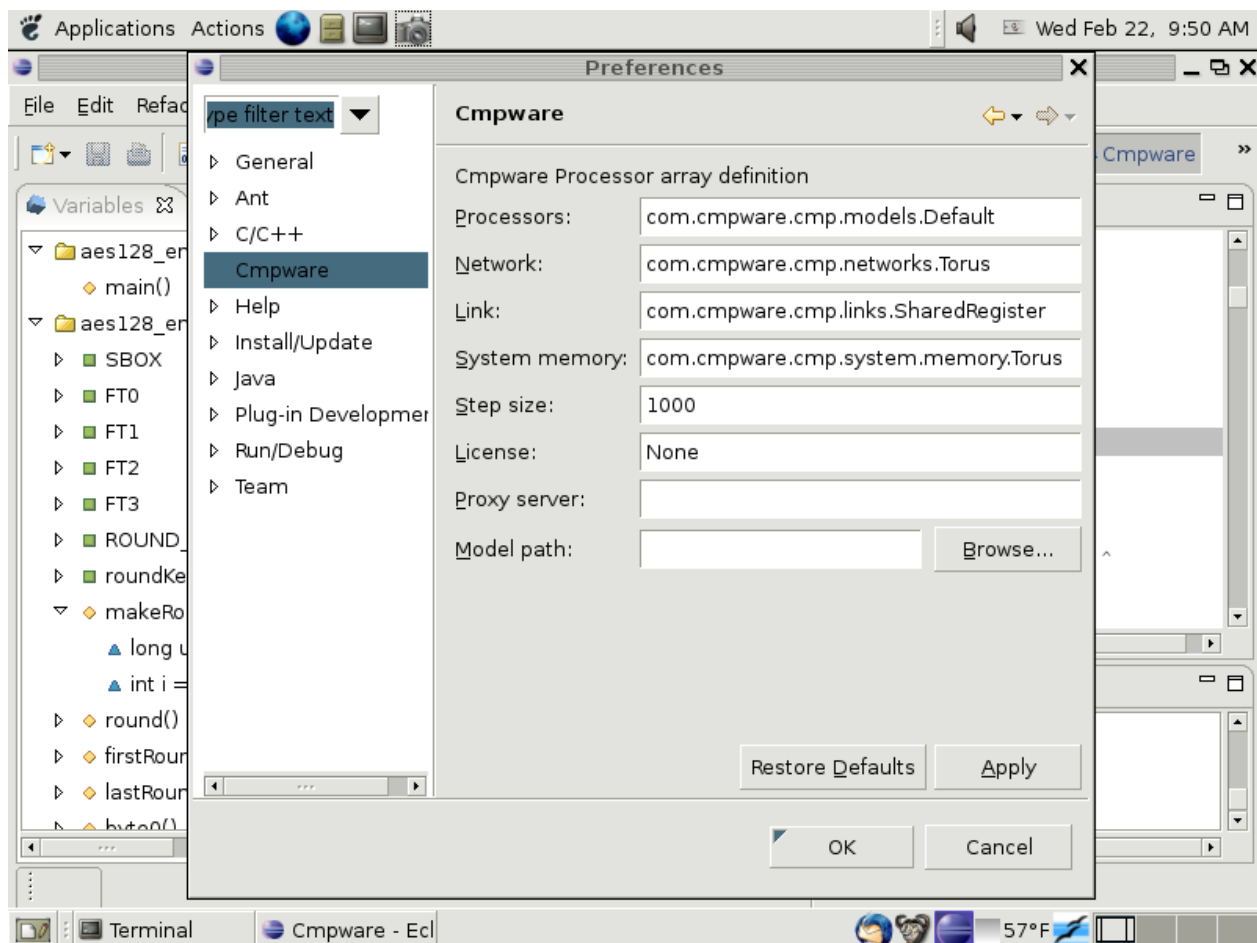



Figure 6: The *Cmpware CMP-DK* Preferences page.

Now when the **Step** button () is pressed, the simulation steps 1000 cycles and the display updates. This coarser display granularity permits simulation to proceed at a faster pace. Note that the simulation may be suspended and the displays updated



before 1000 cycles if a breakpoint, illegal opcode or other system error occurs. Also note that this parameter is also used by the **Run** button (⌘). Steps of 1000 are used between display updates. This sets the 'speed' of the 'animated' display.

At this point it is useful to switch to the main **Memory** display and scroll down to address 0x00003000 as in Figure 7. This is the address in the source code where the results will be placed. Stepping the simulation, 128 bytes of data identical to those printed in the self hosted version can be seen being written to the memory.

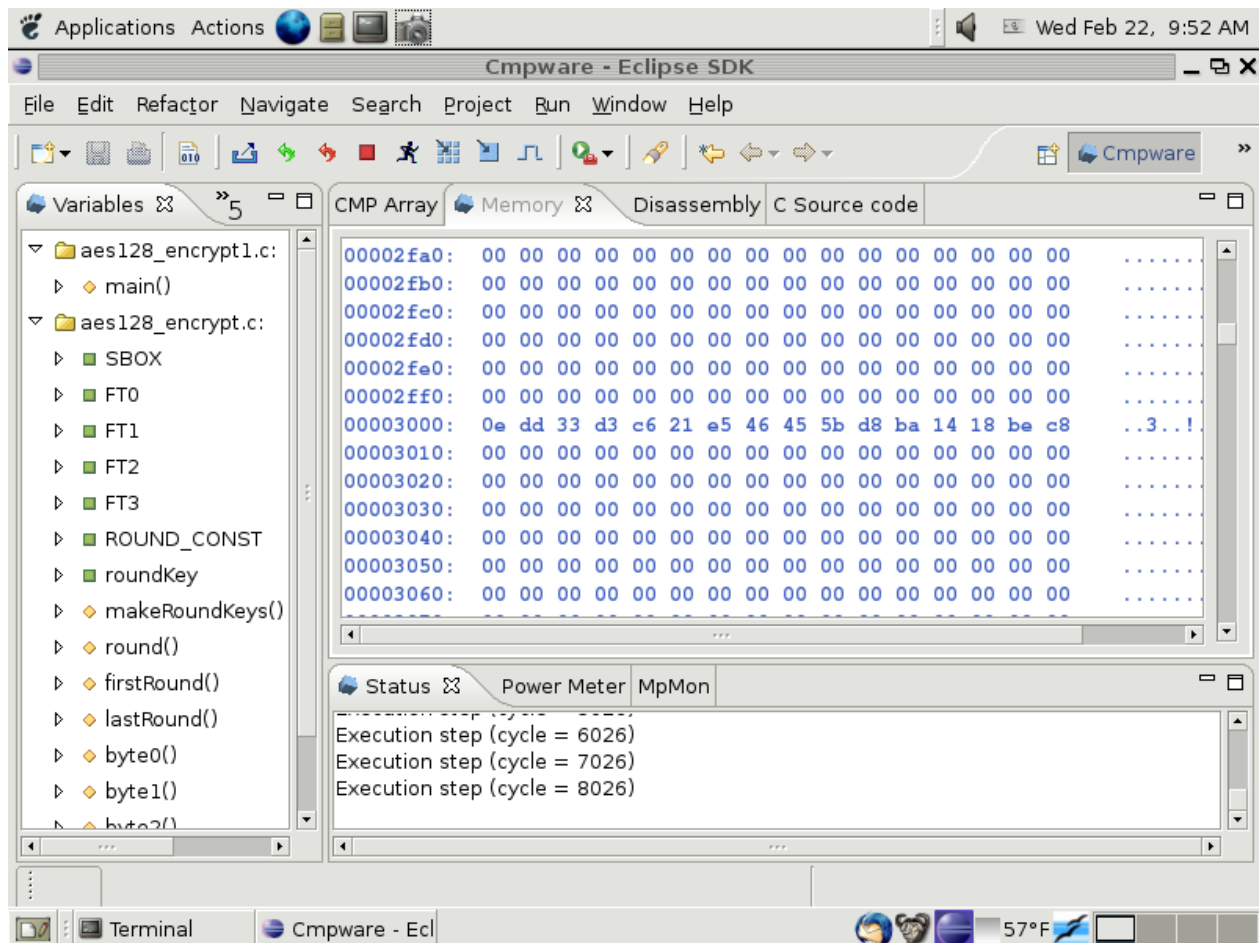


Figure 7: The AES results in local memory.

While the execution can be single stepped until new values appear in the memory display, a breakpoint could be set in the **Source Code** window. This is done by clicking on the vertical bar to the left of the source code text. A small round blue icon (●)



should appear along with a message in the **Status Window** indicating that the breakpoint was set.

Using the **Run** button (⌘), the simulation should execute until the breakpoint is reached. The breakpoint may be removed by clicking on the breakpoint icon to the left of the source code. A message in the **Status Window** should indicate that the breakpoint was removed.

Inspecting the results at address 0x00003000 in the **Memory** display window, it is clear that the AES filter is indeed functioning correctly on the *MIPS32* model and generating the correct results.

Parallelizing the AES Application

The *Cmpware CMP-DK* supports a wide variety of inter-processor communication mechanisms. The default network configuration for the demonstration software is a 2D torus, which is just a 2D nearest-neighbor grid with the ends folded around on itself. This folding makes the topology a 'doughnut', but mostly just serves to keep the network from having any dangling ends.

The nearest neighbor torus communication consists of a shared block of memory and bi-directional *Shared Register* communication channels. These Shared Registers are 32 bit data registers memory mapped at some address in the memory space. They are also fully synchronized, meaning that no data will be written to a Shared Register until any previously written data is read out. And no data will be read until data has been made available by a write. If reads or writes cannot be performed, the processor stalls. This can be thought of as a one word FIFO. Such communication channels may also be found in theoretical models such as *Communicating Sequential Processes* (CSP). These types of channels have the useful feature that they are easy to debug and analyze.

The software definitions in the *CmpwareTorus.h* include file describe these communication resources and the memory map in the simulation models associated with this network. *Appendix E* contains the source code for this default inter-processor communication network configuration. Primarily of interest in this application are the *east* and *west Shared Registers*.

Also note that because these registers exist in the processor memory map, they can be accessed in high level languages as a simple address pointer. No new language constructs or libraries are required. The source code in *Appendix C* and *Appendix D* demonstrate how communication across processors is performed by a simple



assignment to or from an address pointer.

Because of this pointer access to the communication channels, it becomes fairly simple to parallelize this code. The regular serial functions in *aes128_encrypt.c* do not need to be modified at all. These will be executed in parallel across two or more nodes to calculate the final result in parallel. All that is required is that intermediate results, rather than being sent to a local variable, get sent to the next processor for the next stage of processing.

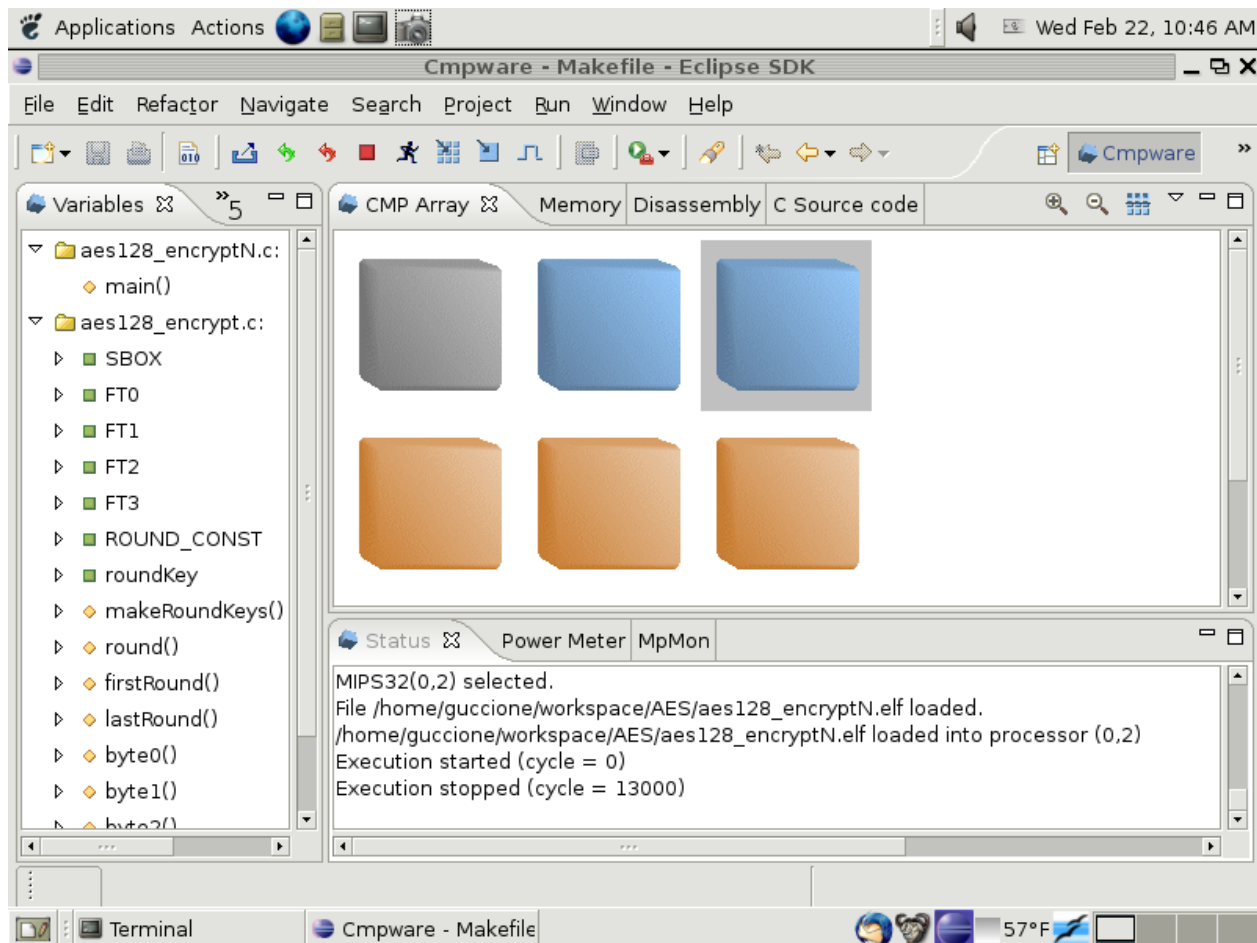



Figure 8: The two node AES implementation executing.

The main loop in the *aes128_encryptN.c* source code in *Appendix D* is very similar to the single processor code. All that has changed is that partial result inputs now come from a pointer (communication channel) and are sent to another pointer



(communication channel). There are some other features of this code, but these will be discussed after the demonstration execution.

As shown in Figure 8, executable code must be loaded into the three *MIPS32* processors in the top row. Again this is done using the **Load** button () to bring up a file selection dialog. The executable *ELF* files are then selected and loaded. In this application, the first processor at (0,0) is loaded with *aes128_encrypt0.elf*. The next two processors, (0,1) and (0,2) are loaded with the *aes128_encryptN.elf* executable file. It is very important that each of these files is loaded into the correct processor with no other operations performed in the interim.

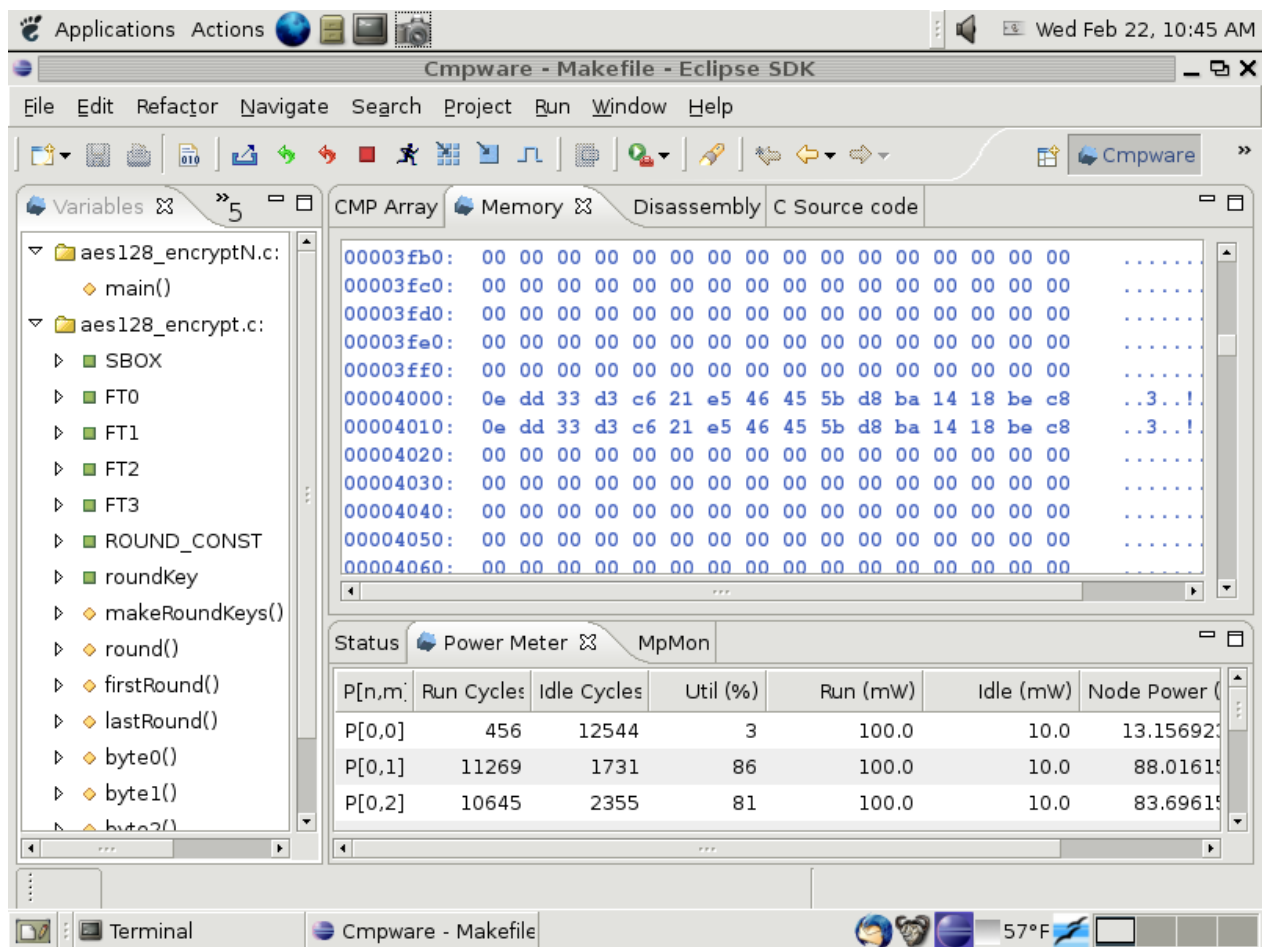


Figure 9: The two node AES implementation results.

Unlike the uniprocessor versions of this software, the synchronous communication of



the FIR application will permit the processors to begin execution when data is available, and stop execution, or at least stall, when no further data is available. The only involvement of processor (0,0) in the calculation is to send data to the next two processors. These two processors split up the task of performing the AES calculation.

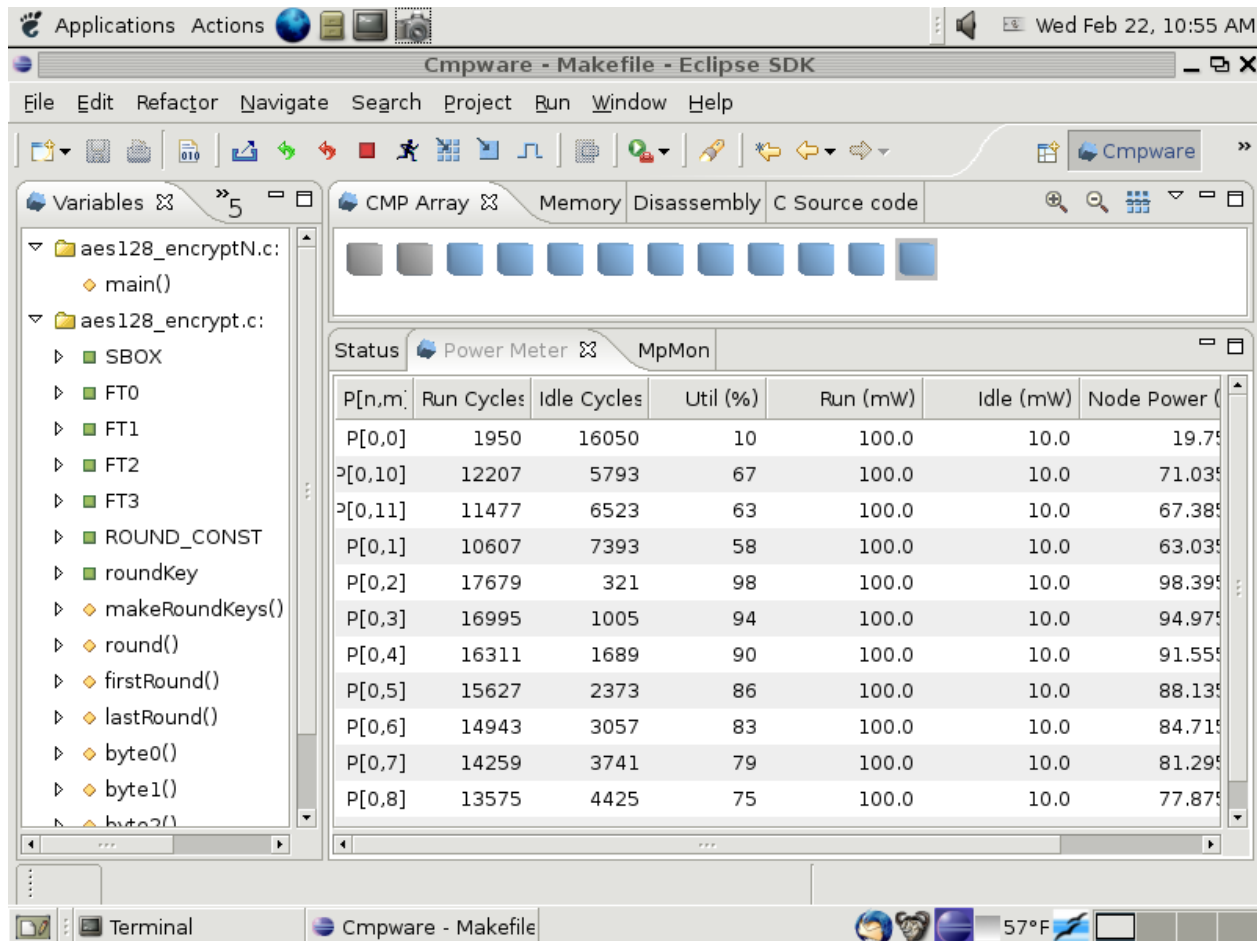


Figure 10: The twelve node AES encryption executing.

Execution may be controlled either manually with the **Step** button (⏏) or as an animation with the **Run** button (⏏). The end of execution will be obvious from the main **CMP Array** window. When all of the top row of processors are 'greyed' and no longer progressing in execution, the calculation is complete. If the **Run** button (⏏) was used, the **Stop** button (■) should be used to halt execution at this point.

Figure 9 shows the **Memory** view in processor (0,2) after execution has completed.



This is exactly the results from the single processor code, as expected. Two rows of data are shown, with the encrypted values repeating. This be because the *aes125_encrypt0.elf* file sends the plain text data in a loop 100 times, for benchmarking purposes.

Also of interest is the **Power Meter** view. It indicates that the AES algorithm executed nearly twice as fast on two processors as on one. Figure 8 shows that the two processing nodes remained busy 86% and 83% of the time. The first processor at (0,0) is busy only 3% of the time because it is simply sending test data to the first processing node, and doing no processing itself. Note that the 86% and 83% numbers include some start-up overheads. For longer runs, these numbers will increase into the mid-90% range.

Figure 10 shows an twelve processor version of the AES algorithm. This is not available with the demonstration version of the *Cmpware CMP-DK* and can only be executed on a licensed version of the software. A glance at the Power Meter view indicates that even at twelve processors, the utilization remains as high as 98% and a large amount of processing in parallel occurs. And as with the two node version, longer runs will increase these number into the mid-90% range. What is perhaps more interesting is that fairly low level parallelism is easily extracted and put to use in a single chip multiprocessor using existing tools and simple software techniques.

Parameterization of the AES Application

The code used to perform the basic computation in the AES algorithm filter is exactly the same in the uniprocessor and multiprocessor implementations. This code can be found in the *aes128_encrypt.c* file and in *Appendix B* at the end of this document. This code is basic, serial version of the functions such as `makeKeys()` and `round()` as they may be found in common uniprocessor implementations. Large tables for the S-boxes are also included here as well as utility functions for byte-level manipulation of the data.

In the uniprocessor version of the code, *aes128_encrypt1.c*, these functions are used inside of the main loop to perform the AES algorithm. Similarly, these same functions are used inside of the main loop of the multiprocessor AES code in the *aes128_encryptN.c* source code file. However, the multiprocessor code in *aes128_encryptN.c* contains some additional functionality.

Much like the FIR filter example, Figure 11 shows three 'parameters' being read in to the processor from the input port `*input` and then being modified and sent to the output port `*output`. What this does is establish three global parameters used to



describe the AES implementation. The first parameter, **processors**, is the number of processors in the calculation. The second is **thisProcessor**, or the current processor number. The last parameter is **key**, the 128 bit encryption key. This is read in as four 32-bit integers. These parameters are read in and sent to the next node, with the **thisProcessor** parameter being incremented.

In this case, the number of nodes can dynamically vary from one to eleven. Eleven is the number of 'rounds' in the AES algorithm and is the basic unit of parallelization for this algorithm. Other implementations could conceivably parallelize this algorithm at a lower level and use more processors and further increase performance.

```
/* Get the parameters */
processors = *input; // Get the number of processors
thisNode = *input; // Get this node number
for (i=0; i<4; i++) // Get the key
    key[i] = *input;

/* Find which rounds are being done on this processor */
startRound = getStartRound(thisNode, processors, rounds);
endRound = getStartRound(thisNode+1, processors, rounds) - 1;

/* If this is the last processor, set output to dev_null */
if (thisNode >= (processors-1))
    output = dev_null;

/* Send the parameters on to the next node*/
*output = processors; // Number of processors
*output = thisNode + 1; // The next node number
for (i=0; i<4; i++)
    *output = key[i];
```

Figure 11: The AES parameterization code.

This parameterization approach is useful for the same reason parameters to function calls are: they permit one piece of code to be compiled and run for a variety of conditions. The *aes128_encryptN.elf* executable from the *aes128_encryptN.c* source code file can be used to build an AES implementation of any size. This not only helps with the reuse of multiprocessor code, but also helps to simplify the loading, debug and management of the multiprocessor software. Judicious use of multiprocessor parameters, much like judicious use of uniprocessor parameters in function calls, will help define the efficiency and flexibility of the code.

In particular, two function calls which define the **startRound** and **endRound** variables



are derived from the input parameters. These do a simple partition of the rounds among the processors. In the case of one processor, all eleven rounds go on that processor, with `startRound` being set to zero and `endRound` being set to eleven. Two processors will split up the rounds with one processor getting five rounds and the other getting six. Details of how exactly these are partitioned can be surmised from looking at the source code for the `getStartRound()` function.

Finally, there is a conditional line of code between the parameters being read the written. This checks to see if the current processor is indeed the last processor in the multiprocessor. If it is, it should not be sending parameters or results to the next node (because there is no next node!). To attempt to do so will simply stall the system and execution will not proceed.

One way to solve this problem is with 'if' statements in places where data is output. But this can quickly become cumbersome. The approach used here is to reassign the `output` port from its default of `east` to `dev_null`. The `dev_null` port is a default input and output port in each processor node that is unsynchronized and can be read and written at any time and will never stall the calculation. This is somewhat similar to the `"/dev/null"` device in Unix file systems which permits data to be sent to a device which is always guaranteed to exist, and never to fail.

This also points out how the use of a small piece of hardware in a multiprocessor system can greatly simplify the software. Without the `dev_null` port, the code would be filled with complex if-then structures to take into account the final node in the sequence. Also note that it is more likely that this application will be used in some application that will take the output data and perform some further functions on it. In this case, the `dev_null` port is really only used for software development and not in the final deployment of the AES algorithm.

What is important here is that the `aes128_encryptN.c` implementation takes the multiprocessor parameterization to a new level. At run-time, parameters can be sent to the nodes defining how many processors are to participate in the calculation. With some additional code, this can even be changed dynamically at run-time. This provides a very interesting range of power and performance behavior, all in a single piece of multiprocessor code.

Conclusions

The *Cmpware CMP-DK* is a rich display environment combining fast simulation and flexible multiprocessor modeling. This makes it an ideal environment for architecture modeling and software development for these systems.



While the execution and display features of the *Cmpware CMP-DK* are notable, much of the power of the system lies in its ability to quickly and flexibly construct processor, network, link and multiprocessor models. This modeling capability is a large part of the commercial version of the *Cmpware CMP-DK*.

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

<http://www.cmpware.com/>

or send an email to:

info@cmpware.com



Appendix A: aes128_encrypt.c Source Code

```
/*
**
** These are all of the routines used to implement the AES algorithm.
** These are all exclusively serial and are used by both serial
** and parallel implementations as well as self-hosted test code.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#ifndef uint32
#define uint32 unsigned long int
#endif

/* Function prototypes */
void lastNode(void);
void makeRoundKeys(uint32 key[4]);
void round(int round, uint32 in[4], uint32 out[4]);
void firstRound(uint32 in[4], uint32 out[4]);
void lastRound(uint32 in[4], uint32 out[4]);
int getStartRound(int thisProcessor, int processors, int rounds);

int byte0(int i);
int byte1(int i);
int byte2(int i);
int byte3(int i);

/* The SBOX table (see end of file) */
extern const uint32 SBOX[256];

/* The FT tables (see end of file) */
extern const uint32 FT0[256];
extern const uint32 FT1[256];
extern const uint32 FT2[256];
extern const uint32 FT3[256];

/* Round constants */
static const uint32 ROUND_CONST[10] = {
    0x01000000, 0x02000000, 0x04000000, 0x08000000,
    0x10000000, 0x20000000, 0x40000000, 0x80000000,
    0x1B000000, 0x36000000
};

/* The round keys */
```



```

static uint32 roundKey[11][4];

/*
** The AES key scheduling routine. This method takes
** in the 128 bit key and produces the keys for the
** various rounds.
**
** @param key The 128 bit encryption key.
**
*/

void makeRoundKeys(uint32 key[4]) {
    int i;

    /* setup encryption round keys */
    roundKey[0][0] = key[0];
    roundKey[0][1] = key[1];
    roundKey[0][2] = key[2];
    roundKey[0][3] = key[3];
    for (i=1; i<11; i++) {
        roundKey[i][0] = roundKey[i-1][0] ^ ROUND_CONST[i-1] ^
            (SBOX[byte1(roundKey[i-1][3])] << 24) ^
            (SBOX[byte2(roundKey[i-1][3])] << 16) ^
            (SBOX[byte3(roundKey[i-1][3])] << 8) ^
            (SBOX[byte0(roundKey[i-1][3])]));
        roundKey[i][1] = roundKey[i-1][1] ^ roundKey[i][0];
        roundKey[i][2] = roundKey[i-1][2] ^ roundKey[i][1];
        roundKey[i][3] = roundKey[i-1][3] ^ roundKey[i][2];
    } /* end for() */

} /* end makeRoundKeys() */

/*
** This method implements a round of the 128 bit
** AES encryption.
**
** @param round The round number.
**
** @param in The 128 bits of input.
**
** @param out The 128 bits of output from this round.
**
*/

void round(int round, uint32 in[4], uint32 out[4]) {

    out[0] = roundKey[round][0] ^
        FT0[byte0(in[0])] ^
        FT1[byte1(in[1])] ^
        FT2[byte2(in[2])] ^
        FT3[byte3(in[3])];
}

```



```
    out[1] = roundKey[round][1] ^
            FT0[byte0(in[1])] ^
            FT1[byte1(in[2])] ^
            FT2[byte2(in[3])] ^
            FT3[byte3(in[0])];

    out[2] = roundKey[round][2] ^
            FT0[byte0(in[2])] ^
            FT1[byte1(in[3])] ^
            FT2[byte2(in[0])] ^
            FT3[byte3(in[1])];

    out[3] = roundKey[round][3] ^
            FT0[byte0(in[3])] ^
            FT1[byte1(in[0])] ^
            FT2[byte2(in[1])] ^
            FT3[byte3(in[2])];

} /* end round() */

/*
** This method implements the first round of the 128 bit
** AES encryption.
**
** @param in  The 128 bits of input.
**
** @param out The 128 bits of output from this round.
**
** */
void firstRound(uint32 in[4], uint32 out[4]) {

    out[0] = in[0] ^ roundKey[0][0];
    out[1] = in[1] ^ roundKey[0][1];
    out[2] = in[2] ^ roundKey[0][2];
    out[3] = in[3] ^ roundKey[0][3];

} /* end firstRound() */

/*
** This method implements the last round of the 128 bit
** AES encryption.
**
** @param in  The 128 bits of input.
**
** @param out The 128 bits of output from this round.
**
** */
void lastRound(uint32 in[4], uint32 out[4]) {
```



```

    out[0] = roundKey[10][0] ^
        (SBOX[byte0(in[0])] << 24) ^
        (SBOX[byte1(in[1])] << 16) ^
        (SBOX[byte2(in[2])] << 8) ^
        (SBOX[byte3(in[3])]);

    out[1] = roundKey[10][1] ^
        (SBOX[byte0(in[1])] << 24) ^
        (SBOX[byte1(in[2])] << 16) ^
        (SBOX[byte2(in[3])] << 8) ^
        (SBOX[byte3(in[0])]);

    out[2] = roundKey[10][2] ^
        (SBOX[byte0(in[2])] << 24) ^
        (SBOX[byte1(in[3])] << 16) ^
        (SBOX[byte2(in[0])] << 8) ^
        (SBOX[byte3(in[1])]);

    out[3] = roundKey[10][3] ^
        (SBOX[byte0(in[3])] << 24) ^
        (SBOX[byte1(in[0])] << 16) ^
        (SBOX[byte2(in[1])] << 8) ^
        (SBOX[byte3(in[2])]);

} /* end lastRound() */

/*
** This method returns the first (high) byte of
** an integer.
**
** @param i The integer input.
**
** @return The high byte of the integer input.
**
** */
int byte0(int i) {return ((i >> 24) & 0x000000ff);}

/*
** This method returns the second byte of
** an integer.
**
** @param i The integer input.
**
** @return The second highest byte of the integer input.
**
** */
int byte1(int i) {return ((i >> 16) & 0x000000ff);}

/*
** This method returns the third highest byte of

```




```
** an integer.
**
** @param i The integer input.
**
** @return The third byte of the integer input.
**
*/

int byte2(int i) {return ((i >> 8) & 0x000000ff);}

/*
** This method returns the last (low) byte of
** an integer.
**
** @param i The integer input.
**
** @return The last (low) byte of the integer input.
**
*/

int byte3(int i) {return (i & 0x000000ff);}

/*
** This method is used to spread rounds across processors
** evenly, even when the number of processors is not an
** even multiple of the number of rounds. This method gives
** the number of the first round to be executed on a node.
**
** @param thisProcessor The processor number, starting
** at zero.
**
** @param processors The number of processors.
**
** @param rounds The number of rounds.
**
** @return The number of first round to be performed on
** this processor.
**
*/

int getStartRound(int thisProcessor, int processors, int rounds) {
    return ((thisProcessor * rounds) / processors);
} /* end getStartRound() */

/* S-box (encrypt) */
static const uint32 SBOX[256] = {
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5,
    0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0,
```



```

0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC,
0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A,
0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0,
0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B,
0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85,
0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5,
0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17,
0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88,
0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C,
0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9,
0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6,
0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E,
0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94,
0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68,
0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};

```

```

static const uint32 FT0[256] = {
0xC66363A5, 0xF87C7C84, 0xEE777799, 0xF67B7B8D,
0xFFF2F20D, 0xD66B6BBB, 0xDE6F6FB1, 0x91C5C554,
0x60303050, 0x02010103, 0xCE6767A9, 0x562B2B7D,
0xE7FEFE19, 0xB5D7D762, 0x4DABABE6, 0xEC76769A,
0x8FCACA45, 0x1F82829D, 0x89C9C940, 0xFA7D7D87,
0EFFAFA15, 0xB25959EB, 0x8E4747C9, 0xFBF0F00B,
0x41ADADEC, 0xB3D4D467, 0x5FA2A2FD, 0x45AFAFEA,
0x239C9CBF, 0x53A4A4F7, 0xE4727296, 0x9BC0C05B,
0x75B7B7C2, 0xE1FDFD1C, 0x3D9393AE, 0x4C26266A,
0x6C36365A, 0x7E3F3F41, 0xF5F7F702, 0x83CCCC4F,
0x6834345C, 0x51A5A5F4, 0xD1E5E534, 0xF9F1F108,
0xE2717193, 0xABD8D873, 0x62313153, 0x2A15153F,
0x0804040C, 0x95C7C752, 0x46232365, 0x9DC3C35E,
0x30181828, 0x379696A1, 0x0A05050F, 0x2F9A9AB5,
0x0E070709, 0x24121236, 0x1B80809B, 0xDFE2E23D,
0xCDEBEB26, 0x4E272769, 0x7FB2B2CD, 0xEA75759F,
0x1209091B, 0x1D83839E, 0x582C2C74, 0x341A1A2E,
0x361B1B2D, 0xDC6E6EB2, 0xB45A5AEE, 0x5BA0A0FB,
0xA45252F6, 0x763B3B4D, 0xB7D6D661, 0x7DB3B3CE,
0x5229297B, 0xDDE3E33E, 0x5E2F2F71, 0x13848497,
0xA65353F5, 0xB9D1D168, 0x00000000, 0xC1EDED2C,

```



```

0x40202060, 0xE3FCFC1F, 0x79B1B1C8, 0xB65B5BED,
0xD46A6ABE, 0x8DCBCB46, 0x67BEBED9, 0x7239394B,
0x944A4ADE, 0x984C4CD4, 0xB05858E8, 0x85CFCF4A,
0xBBD0D06B, 0xC5EFEF2A, 0x4FAAAAE5, 0xEDFBFB16,
0x864343C5, 0x9A4D4DD7, 0x66333355, 0x11858594,
0x8A4545CF, 0xE9F9F910, 0x04020206, 0xFE7F7F81,
0xA05050F0, 0x783C3C44, 0x259F9FBA, 0x4BA8A8E3,
0xA25151F3, 0x5DA3A3FE, 0x804040C0, 0x058F8F8A,
0x3F9292AD, 0x219D9DBC, 0x70383848, 0xF1F5F504,
0x63BCBCDF, 0x77B6B6C1, 0xAFDADA75, 0x42212163,
0x20101030, 0xE5FFFF1A, 0xFDF3F30E, 0xBF2D2D6D,
0x81CDCD4C, 0x180C0C14, 0x26131335, 0xC3ECEC2F,
0xBE5F5FE1, 0x359797A2, 0x884444CC, 0x2E171739,
0x93C4C457, 0x55A7A7F2, 0xFC7E7E82, 0x7A3D3D47,
0xC86464AC, 0xBA5D5DE7, 0x3219192B, 0xE6737395,
0xC06060A0, 0x19818198, 0x9E4F4FD1, 0xA3DCDC7F,
0x44222266, 0x542A2A7E, 0x3B9090AB, 0x0B888883,
0x8C4646CA, 0xC7EEEE29, 0x6BB8B8D3, 0x2814143C,
0xA7DEDE79, 0xBC5E5EE2, 0x160B0B1D, 0xADDBDB76,
0xDBE0E03B, 0x64323256, 0x743A3A4E, 0x140A0A1E,
0x924949DB, 0x0C06060A, 0x4824246C, 0xB85C5CE4,
0x9FC2C25D, 0xBDD3D36E, 0x43ACACEF, 0xC46262A6,
0x399191A8, 0x319595A4, 0xD3E4E437, 0xF279798B,
0xD5E7E732, 0x8BC8C843, 0x6E373759, 0xDA6D6DB7,
0x018D8D8C, 0xB1D5D564, 0x9C4E4ED2, 0x49A9A9E0,
0xD86C6CBA, 0xAC5656FA, 0xF3F4F407, 0xCFEAEA25,
0xCA6565AF, 0xF47A7A8E, 0x47AEAEE9, 0x10080818,
0x6FBABAD5, 0xF0787888, 0x4A25256F, 0x5C2E2E72,
0x381C1C24, 0x57A6A6F1, 0x73B4B4C7, 0x97C6C651,
0xCBE8E823, 0xA1DDDD7C, 0xE874749C, 0x3E1F1F21,
0x964B4BDD, 0x61BDBDDC, 0x0D8B8B86, 0x0F8A8A85,
0xE0707090, 0x7C3E3E42, 0x71B5B5C4, 0xCC6666AA,
0x904848D8, 0x06030305, 0xF7F6F601, 0x1C0E0E12,
0xC26161A3, 0x6A35355F, 0xAE5757F9, 0x69B9B9D0,
0x17868691, 0x99C1C158, 0x3A1D1D27, 0x279E9EB9,
0xD9E1E138, 0xEBF8F813, 0x2B9898B3, 0x22111133,
0xD26969BB, 0xA9D9D970, 0x078E8E89, 0x339494A7,
0x2D9B9BB6, 0x3C1E1E22, 0x15878792, 0xC9E9E920,
0x87CECE49, 0xAA5555FF, 0x50282878, 0xA5DFDF7A,
0x038C8C8F, 0x59A1A1F8, 0x09898980, 0x1A0D0D17,
0x65BFBFDA, 0xD7E6E631, 0x844242C6, 0xD06868B8,
0x824141C3, 0x299999B0, 0x5A2D2D77, 0x1E0F0F11,
0x7BB0B0CB, 0xA85454FC, 0x6DBBBBD6, 0x2C16163A
};

```

```

static const uint32 FT1[256] = {
0xA5C66363, 0x84F87C7C, 0x99EE7777, 0x8DF67B7B,
0x0DFFF2F2, 0xBDD66B6B, 0xB1DE6F6F, 0x5491C5C5,
0x50603030, 0x03020101, 0xA9CE6767, 0x7D562B2B,
0x19E7FEFE, 0x62B5D7D7, 0xE64DABAB, 0x9AEC7676,
0x458FCACA, 0x9D1F8282, 0x4089C9C9, 0x87FA7D7D,
0x15EFFAFA, 0xEBB25959, 0xC98E4747, 0x0BFBF0F0,
0xEC41ADAD, 0x67B3D4D4, 0xFD5FA2A2, 0xEA45AFAF,
0xBF239C9C, 0xF753A4A4, 0x96E47272, 0x5B9BC0C0,
0xC275B7B7, 0x1CE1FDFD, 0xAE3D9393, 0x6A4C2626,

```



```

0x5A6C3636, 0x417E3F3F, 0x02F5F7F7, 0x4F83CCCC,
0x5C683434, 0xF451A5A5, 0x34D1E5E5, 0x08F9F1F1,
0x93E27171, 0x73ABD8D8, 0x53623131, 0x3F2A1515,
0x0C080404, 0x5295C7C7, 0x65462323, 0x5E9DC3C3,
0x28301818, 0xA1379696, 0x0F0A0505, 0xB52F9A9A,
0x090E0707, 0x36241212, 0x9B1B8080, 0x3DDFE2E2,
0x26CDEBEB, 0x694E2727, 0xCD7FB2B2, 0x9FEA7575,
0x1B120909, 0x9E1D8383, 0x74582C2C, 0x2E341A1A,
0x2D361B1B, 0xB2DC6E6E, 0xEEB45A5A, 0xFB5BA0A0,
0xF6A45252, 0x4D763B3B, 0x61B7D6D6, 0xCE7DB3B3,
0x7B522929, 0x3EDDE3E3, 0x715E2F2F, 0x97138484,
0xF5A65353, 0x68B9D1D1, 0x00000000, 0x2CC1EDED,
0x60402020, 0x1FE3FCFC, 0xC879B1B1, 0xEDB65B5B,
0xBED46A6A, 0x468DCBCB, 0xD967BEBE, 0x4B723939,
0xDE944A4A, 0xD4984C4C, 0xE8B05858, 0x4A85CFCF,
0x6BBBD0D0, 0x2AC5EFEF, 0xE54FAAAA, 0x16EDFBFB,
0xC5864343, 0xD79A4D4D, 0x55663333, 0x94118585,
0xCF8A4545, 0x10E9F9F9, 0x06040202, 0x81FE7F7F,
0xF0A05050, 0x44783C3C, 0xBA259F9F, 0xE34BA8A8,
0xF3A25151, 0xFE5DA3A3, 0xC0804040, 0x8A058F8F,
0xAD3F9292, 0xBC219D9D, 0x48703838, 0x04F1F5F5,
0xDF63BCBC, 0xC177B6B6, 0x75AFDADA, 0x63422121,
0x30201010, 0x1AE5FFFF, 0x0EFD3F3F, 0x6DBFD2D2,
0x4C81CDCD, 0x14180C0C, 0x35261313, 0x2FC3ECEC,
0xE1BE5F5F, 0xA2359797, 0xCC884444, 0x392E1717,
0x5793C4C4, 0xF255A7A7, 0x82FC7E7E, 0x477A3D3D,
0xACC86464, 0xE7BA5D5D, 0x2B321919, 0x95E67373,
0xA0C06060, 0x98198181, 0xD19E4F4F, 0x7FA3DCDC,
0x66442222, 0x7E542A2A, 0xAB3B9090, 0x830B8888,
0xCA8C4646, 0x29C7EEEE, 0xD36BB8B8, 0x3C281414,
0x79A7DEDE, 0xE2BC5E5E, 0x1D160B0B, 0x76ADDBDB,
0x3BDBE0E0, 0x56643232, 0x4E743A3A, 0x1E140A0A,
0xDB924949, 0x0A0C0606, 0x6C482424, 0xE4B85C5C,
0x5D9FC2C2, 0x6EBDD3D3, 0xEF43ACAC, 0xA6C46262,
0xA8399191, 0xA4319595, 0x37D3E4E4, 0x8BF27979,
0x32D5E7E7, 0x438BC8C8, 0x596E3737, 0xB7DA6D6D,
0x8C018D8D, 0x64B1D5D5, 0xD29C4E4E, 0xE049A9A9,
0xB4D86C6C, 0xFAAC5656, 0x07F3F4F4, 0x25CFEAEA,
0xAFCA6565, 0x8EF47A7A, 0xE947AEAE, 0x18100808,
0xD56FBABA, 0x88F07878, 0x6F4A2525, 0x725C2E2E,
0x24381C1C, 0xF157A6A6, 0xC773B4B4, 0x5197C6C6,
0x23CBE8E8, 0x7CA1DDDD, 0x9CE87474, 0x213E1F1F,
0xDD964B4B, 0xDC61BDBD, 0x860D8B8B, 0x850F8A8A,
0x90E07070, 0x427C3E3E, 0xC471B5B5, 0xAACC6666,
0xD8904848, 0x05060303, 0x01F7F6F6, 0x121C0E0E,
0xA3C26161, 0x5F6A3535, 0xF9AE5757, 0xD069B9B9,
0x91178686, 0x5899C1C1, 0x273A1D1D, 0xB9279E9E,
0x38D9E1E1, 0x13EBF8F8, 0xB32B9898, 0x33221111,
0xBBD26969, 0x70A9D9D9, 0x89078E8E, 0xA7339494,
0xB62D9B9B, 0x223C1E1E, 0x92158787, 0x20C9E9E9,
0x4987CECE, 0xFFAA5555, 0x78502828, 0x7AA5DFDF,
0x8F038C8C, 0xF859A1A1, 0x80098989, 0x171A0D0D,
0xDA65BFBF, 0x31D7E6E6, 0xC6844242, 0xB8D06868,
0xC3824141, 0xB0299999, 0x775A2D2D, 0x111E0F0F,
0xCB7BB0B0, 0xFCA85454, 0xD66DBBBB, 0x3A2C1616
    
```



```
};  
  
static const uint32 FT2[256] = {  
    0x63A5C663, 0x7C84F87C, 0x7799EE77, 0x7B8DF67B,  
    0xF20DFFF2, 0x6BBDD66B, 0x6FB1DE6F, 0xC55491C5,  
    0x30506030, 0x01030201, 0x67A9CE67, 0x2B7D562B,  
    0xFE19E7FE, 0xD762B5D7, 0xABE64DAB, 0x769AEC76,  
    0xCA458FCA, 0x829D1F82, 0xC94089C9, 0x7D87FA7D,  
    0xFA15E7FA, 0x59EBB259, 0x47C98E47, 0xF00BFBF0,  
    0xADEC41AD, 0xD467B3D4, 0xA2FD5FA2, 0xAFEA45AF,  
    0x9CBF239C, 0xA4F753A4, 0x7296E472, 0xC05B9BC0,  
    0xB7C275B7, 0xFD1CE1FD, 0x93AE3D93, 0x266A4C26,  
    0x365A6C36, 0x3F417E3F, 0xF702F5F7, 0xCC4F83CC,  
    0x345C6834, 0xA5F451A5, 0xE534D1E5, 0xF108F9F1,  
    0x7193E271, 0xD873ABD8, 0x31536231, 0x153F2A15,  
    0x040C0804, 0xC75295C7, 0x23654623, 0xC35E9DC3,  
    0x18283018, 0x96A13796, 0x050F0A05, 0x9AB52F9A,  
    0x07090E07, 0x12362412, 0x809B1B80, 0xE23DDFE2,  
    0xEB26CDEB, 0x27694E27, 0xB2CD7FB2, 0x759FEA75,  
    0x091B1209, 0x839E1D83, 0x2C74582C, 0x1A2E341A,  
    0x1B2D361B, 0x6EB2DC6E, 0x5AEEB45A, 0xA0FB5BA0,  
    0x52F6A452, 0x3B4D763B, 0xD661B7D6, 0xB3CE7DB3,  
    0x297B5229, 0xE33EDDE3, 0x2F715E2F, 0x84971384,  
    0x53F5A653, 0xD168B9D1, 0x00000000, 0xED2CC1ED,  
    0x20604020, 0xFC1FE3FC, 0xB1C879B1, 0x5BEDB65B,  
    0x6ABED46A, 0xCB468DCB, 0xBED967BE, 0x394B7239,  
    0x4ADE944A, 0x4CD4984C, 0x58E8B058, 0xCF4A85CF,  
    0xD06BBBD0, 0xEF2AC5EF, 0xAAE54FAA, 0xFB16EDFB,  
    0x43C58643, 0x4DD79A4D, 0x33556633, 0x85941185,  
    0x45CF8A45, 0xF910E9F9, 0x02060402, 0x7F81FE7F,  
    0x50F0A050, 0x3C44783C, 0x9FBA259F, 0xA8E34BA8,  
    0x51F3A251, 0xA3FE5DA3, 0x40C08040, 0x8F8A058F,  
    0x92AD3F92, 0x9DBC219D, 0x38487038, 0xF504F1F5,  
    0xBCDF63BC, 0xB6C177B6, 0xDA75AFDA, 0x21634221,  
    0x10302010, 0xFF1AE5FF, 0xF30EFD3, 0xD26DBFD2,  
    0xCD4C81CD, 0x0C14180C, 0x13352613, 0xEC2FC3EC,  
    0x5FE1BE5F, 0x97A23597, 0x44CC8844, 0x17392E17,  
    0xC45793C4, 0xA7F255A7, 0x7E82FC7E, 0x3D477A3D,  
    0x64ACC864, 0x5DE7BA5D, 0x192B3219, 0x7395E673,  
    0x60A0C060, 0x81981981, 0x4FD19E4F, 0xDC7FA3DC,  
    0x22664422, 0x2A7E542A, 0x90AB3B90, 0x88830B88,  
    0x46CA8C46, 0xEE29C7EE, 0xB8D36BB8, 0x143C2814,  
    0xDE79A7DE, 0x5EE2BC5E, 0x0B1D160B, 0xDB76ADDB,  
    0xE03BDBE0, 0x32566432, 0x3A4E743A, 0x0A1E140A,  
    0x49DB9249, 0x060A0C06, 0x246C4824, 0x5CE4B85C,  
    0xC25D9FC2, 0xD36EBDD3, 0xACEF43AC, 0x62A6C462,  
    0x91A83991, 0x95A43195, 0xE437D3E4, 0x798BF279,  
    0xE732D5E7, 0xC8438BC8, 0x37596E37, 0x6DB7DA6D,  
    0x8D8C018D, 0xD564B1D5, 0x4ED29C4E, 0xA9E049A9,  
    0x6CB4D86C, 0x56FAAC56, 0xF407F3F4, 0xEA25CFEA,  
    0x65AFCA65, 0x7A8EF47A, 0xAEE947AE, 0x08181008,  
    0xBAD56FBA, 0x7888F078, 0x256F4A25, 0x2E725C2E,  
    0x1C24381C, 0xA6F157A6, 0xB4C773B4, 0xC65197C6,  
    0xE823CBE8, 0xDD7CA1DD, 0x749CE874, 0x1F213E1F,  
    0x4BDD964B, 0xBDDC61BD, 0x8B860D8B, 0x8A850F8A,
```



```

0x7090E070, 0x3E427C3E, 0xB5C471B5, 0x66AACC66,
0x48D89048, 0x03050603, 0xF601F7F6, 0x0E121C0E,
0x61A3C261, 0x355F6A35, 0x57F9AE57, 0xB9D069B9,
0x86911786, 0xC15899C1, 0x1D273A1D, 0x9EB9279E,
0xE138D9E1, 0xF813EBF8, 0x98B32B98, 0x11332211,
0x69BBD269, 0xD970A9D9, 0x8E89078E, 0x94A73394,
0x9BB62D9B, 0x1E223C1E, 0x87921587, 0xE920C9E9,
0xCE4987CE, 0x55FFAA55, 0x28785028, 0xDF7AA5DF,
0x8C8F038C, 0xA1F859A1, 0x89800989, 0x0D171A0D,
0xBFDA65BF, 0xE631D7E6, 0x42C68442, 0x68B8D068,
0x41C38241, 0x99B02999, 0x2D775A2D, 0x0F111E0F,
0xB0CB7BB0, 0x54FCA854, 0xBBD66DBB, 0x163A2C16
};

```

```

static const uint32 FT3[256] = {
0x6363A5C6, 0x7C7C84F8, 0x777799EE, 0x7B7B8DF6,
0xF2F20DFE, 0x6B6BBDD6, 0x6F6FB1DE, 0xC5C55491,
0x30305060, 0x01010302, 0x6767A9CE, 0x2B2B7D56,
0xFEFE19E7, 0xD7D762B5, 0xABABE64D, 0x76769AEC,
0xCACA458F, 0x82829D1F, 0xC9C94089, 0x7D7D87FA,
0xFAFA15EF, 0x5959EBB2, 0x4747C98E, 0xF0F00BFB,
0xADADEC41, 0xD4D467B3, 0xA2A2FD5F, 0xAFAFEA45,
0x9C9CBF23, 0xA4A4F753, 0x727296E4, 0xC0C05B9B,
0xB7B7C275, 0xFDFD1CE1, 0x9393AE3D, 0x26266A4C,
0x36365A6C, 0x3F3F417E, 0xF7F702F5, 0xCCCC4F83,
0x34345C68, 0xA5A5F451, 0xE5E534D1, 0xF1F108F9,
0x717193E2, 0xD8D873AB, 0x31315362, 0x15153F2A,
0x04040C08, 0xC7C75295, 0x23236546, 0xC3C35E9D,
0x18182830, 0x9696A137, 0x05050F0A, 0x9A9AB52F,
0x0707090E, 0x12123624, 0x80809B1B, 0xE2E23DDF,
0xEBEB26CD, 0x2727694E, 0xB2B2CD7F, 0x75759FEA,
0x09091B12, 0x83839E1D, 0x2C2C7458, 0x1A1A2E34,
0x1B1B2D36, 0x6E6EB2DC, 0x5A5AAEEB4, 0xA0A0FB5B,
0x5252F6A4, 0x3B3B4D76, 0xD6D661B7, 0xB3B3CE7D,
0x29297B52, 0xE3E33EDD, 0x2F2F715E, 0x84849713,
0x5353F5A6, 0xD1D168B9, 0x00000000, 0xEDED2CC1,
0x20206040, 0xFCFC1FE3, 0xB1B1C879, 0x5B5BBEDB6,
0x6A6ABED4, 0xCBCB468D, 0xBEBED967, 0x39394B72,
0x4A4ADE94, 0x4C4CD498, 0x5858E8B0, 0xCF4A85,
0xD0D06BBB, 0xEFEF2AC5, 0xAAAAE54F, 0xFBFB16ED,
0x4343C586, 0x4D4DD79A, 0x33335566, 0x85859411,
0x4545CF8A, 0xF9F910E9, 0x02020604, 0x7F7F81FE,
0x5050F0A0, 0x3C3C4478, 0x9F9FBA25, 0xA8A8E34B,
0x5151F3A2, 0xA3A3FE5D, 0x4040C080, 0x8F8F8A05,
0x9292AD3F, 0x9D9DBC21, 0x38384870, 0xF5F504F1,
0xBCBCDF63, 0xB6B6C177, 0xDADA75AF, 0x21216342,
0x10103020, 0xFFFF1AE5, 0xF3F30EFD, 0xD2D26DBF,
0xCDCD4C81, 0x0C0C1418, 0x13133526, 0xECEC2FC3,
0x5F5FE1BE, 0x9797A235, 0x4444CC88, 0x1717392E,
0xC4C45793, 0xA7A7F255, 0x7E7E82FC, 0x3D3D477A,
0x6464ACC8, 0x5D5DE7BA, 0x19192B32, 0x737395E6,
0x6060A0C0, 0x81819819, 0x4F4FD19E, 0xDCDC7FA3,
0x22226644, 0x2A2A7E54, 0x9090AB3B, 0x8888830B,
0x4646CA8C, 0xEEEE29C7, 0xB8B8D36B, 0x14143C28,
0xDEDE79A7, 0x5E5EE2BC, 0x0B0B1D16, 0xDBDB76AD,

```



```
0xE0E03BDB, 0x32325664, 0x3A3A4E74, 0x0A0A1E14,  
0x4949DB92, 0x06060A0C, 0x24246C48, 0x5C5CE4B8,  
0xC2C25D9F, 0xD3D36EBD, 0xACACEF43, 0x6262A6C4,  
0x9191A839, 0x9595A431, 0xE4E437D3, 0x79798BF2,  
0xE7E732D5, 0xC8C8438B, 0x3737596E, 0x6D6DB7DA,  
0x8D8D8C01, 0xD5D564B1, 0x4E4ED29C, 0xA9A9E049,  
0x6C6CB4D8, 0x5656FAAC, 0xF4F407F3, 0xEAEA25CF,  
0x6565AFCA, 0x7A7A8EF4, 0xAEAEE947, 0x08081810,  
0xBABAD56F, 0x787888F0, 0x25256F4A, 0x2E2E725C,  
0x1C1C2438, 0xA6A6F157, 0xB4B4C773, 0xC6C65197,  
0xE8E823CB, 0xDDDD7CA1, 0x74749CE8, 0x1F1F213E,  
0x4B4BDD96, 0xBDBDDC61, 0x8B8B860D, 0x8A8A850F,  
0x707090E0, 0x3E3E427C, 0xB5B5C471, 0x6666AACC,  
0x4848D890, 0x03030506, 0xF6F601F7, 0x0E0E121C,  
0x6161A3C2, 0x35355F6A, 0x5757F9AE, 0xB9B9D069,  
0x86869117, 0xC1C15899, 0x1D1D273A, 0x9E9EB927,  
0xE1E138D9, 0xF8F813EB, 0x9898B32B, 0x11113322,  
0x6969BBD2, 0xD9D970A9, 0x8E8E8907, 0x9494A733,  
0x9B9BB62D, 0x1E1E223C, 0x87879215, 0xE9E920C9,  
0xCECE4987, 0x5555FFAA, 0x28287850, 0xDFDF7AA5,  
0x8C8C8F03, 0xA1A1F859, 0x89898009, 0x0D0D171A,  
0xBFBFDA65, 0xE6E631D7, 0x4242C684, 0x6868B8D0,  
0x4141C382, 0x9999B029, 0x2D2D775A, 0x0F0F111E,  
0xB0B0CB7B, 0x5454FCA8, 0BBBBD66D, 0x16163A2C  
};
```



Appendix B: aes128_encrypt1.c Source Code

```
/*
** This implements the single processor version of the
** AES encryption algorithm. It can be compiled to run
** either self-hosted (on a PC or workstation) or on a
** single embedded processor. It uses the same routines
** used in the multiprocessor AES and is used primarily
** to test the algorithm.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#ifdef uint32
#define uint32 unsigned long int
#endif

/* Define SELFHOSTED to run on standard host */
/* undefine to run on simulator / hardware */
/* (Or use the -DSELFHOSTED flag to gcc) */
// #define SELFHOSTED

/* Function prototypes */
void makeRoundKeys(uint32 key[4]);
void round(int round, uint32 in[4], uint32 out[4]);
void firstRound(uint32 in[4], uint32 out[4]);
void lastRound(uint32 in[4], uint32 out[4]);

/* A place to put the result */
#ifdef SELFHOSTED
    static uint32 result[4];
#else
    static uint32 *result = (uint32 *) 0x3000;
#endif

/* The plain text */
uint32 plainText[4] =
    {0x00000000, 0x00000000, 0x00000000, 0x00000000};

/* The key */
uint32 key[4] =
    {0x80000000, 0x00000000, 0x00000000, 0x00000000};

int main(int argc, char *argv[]) {
    uint32 tmp[4];
    uint32 out[4];

    makeRoundKeys(key);
```



```
firstRound(plainText, out);
round(1, out, tmp);
round(2, tmp, out);
round(3, out, tmp);
round(4, tmp, out);
round(5, out, tmp);
round(6, tmp, out);
round(7, out, tmp);
round(8, tmp, out);
round(9, out, tmp);
lastRound(tmp, out);

result[0] = out[0];
result[1] = out[1];
result[2] = out[2];
result[3] = out[3];

#ifdef SELFHOSTED

/* Print out the result (if we have a stdout) */
printf("Result:  0x%08x, 0x%08x, 0x%08x, 0x%08x\n",
       out[0], out[1], out[2], out[3]);

#endif

} /* end main() */
```



Appendix C: aes128_encrypt0.c Source Code

```
/*
** This implements the first node in the multiprocessor
** version of the AES code. This is just used to send the
** parameter information and the input data to the first AES
** processing node. This can be done multiple times if the
** goal is to benchmark the performance.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#include "CmpwareTorus.h"

/* Define the number of processors */
/* (Can also be done using the gcc -D flag) */
// #define PROCESSORS 2

#ifndef uint32
#define uint32 unsigned long int
#endif

/* The number of time to send the data */
const int REPEAT = 100;

/* The input (128 bits) */
static int input[4] =
    {0x00000000, 0x00000000, 0x00000000, 0x00000000};

/* The encryption key (128 bits) */
static int key[4] =
    {0x80000000, 0x00000000, 0x00000000, 0x00000000};

int main(int argc, char *argv[]) {
    int i;
    int j;
    int tmp;
    int rounds = 11;
    int thisNode = 0;
    Port out = east;

    /* Send out parameters */
    *out = PROCESSORS; // Number of processors
    *out = thisNode; // The first AES processing node
    for (i=0; i<4; i++)
        *out = key[i];
}
```



```
/* Send data out (outer loop for performance measurement) */
for (j=0; j<REPEAT; j++)
    for (i=0; i<4; i++)
        *out = input[i];

/* Stall at the end */
tmp = *out;

} /* end main() */
```



Appendix D: aes128_encryptN.c Source Code

```
/*
**
** This application implements the 128-bit AES encryption
** algorithm. It is intended to be split across multiple
** processors.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#include "CmpwareTorus.h"

#ifndef uint32
#define uint32 unsigned long int
#endif

/* Function prototypes */
void lastNode(void);
void makeRoundKeys(uint32 key[4]);
void round(int round, uint32 in[4], uint32 out[4]);
void firstRound(uint32 in[4], uint32 out[4]);
void lastRound(uint32 in[4], uint32 out[4]);
int getStartRound(int thisProcessor, int processors, int rounds);

/* A place to put the result */
static int *result = (int *) 0x4000;

int main(int argc, char *argv[]) {
    int rounds = 11;
    int i;
    int j;
    int k = 0;
    int processors;
    int thisNode;
    int startRound;
    int endRound;
    uint32 key[4];
    uint32 in[4];
    uint32 out[4];
    Port input = west;
    Port output = east;

    /* Get the parameters */
    processors = *input; // Get the number of processors
    thisNode = *input; // Get this node number
    for (i=0; i<4; i++) // Get the key
```



```
    key[i] = *input;

    /* Find which rounds are being done on this processor */
    startRound = getStartRound(thisNode, processors, rounds);
    endRound = getStartRound(thisNode+1, processors, rounds) - 1;

    /* If this is the last processor, set output to dev_null */
    if (thisNode >= (processors-1))
        output = dev_null;

    /* Send the parameters on to the next node*/
    *output = processors;    // Number of processors
    *output = thisNode + 1; // The next node number
    for (i=0; i<4; i++)
        *output = key[i];

makeRoundKeys(key);

/* Keep going as long as there is data */
for(;;) {

    /* Get input */
    for (i=0; i<4; i++)
        in[i] = *input;

    /* Do the rounds */
    for (j=startRound; j<=endRound; j++) {
        if (j == 0)
            firstRound(in, out);
        else if ((j >=1) &&(j <=9))
            round(j, in, out);
        else
            lastRound(in, out);

        /* Send result to next round */
        for (i=0; i<4; i++)
            in[i] = out[i];

    } /* end for(j) */

    /* Save the intermediate output (for debug) */
    for (i=0; i<4; i++)
        result[k++] = out[i];

    /* Send output to next node (if there is one) */
    for (i=0; i<4; i++)
        *output = out[i];

} /* end for(;;) */

} /* end main() */
```



Appendix E: CmpwareTorus.h Source Code

```
/*
**
** This defines the shared memory and links in the 'Torus' topology.
** This must agree with the values in the memory map for the
** hardware and the simulation model.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#ifndef _CMPWARETORUS_H_
#define _CMPWARETORUS_H_

/* The Memory Mapped IO Ports */
typedef volatile int *Port;

/* A shared memory address */
typedef unsigned char *Address;

/* The size of the local memory */
#define LOCAL_MEMORY_SIZE (32 * 1024)

/* The size of the shared memory */
#define SHARED_MEMORY_SIZE (8 * 1024)

/* Memory Mapped IO ports */
#ifdef SELFHOSTED
    /* For self-hosted testing, just make a pointer to an int */
    Port north[1];
    Port east[1];
    Port south[1];
    Port west[1];
    Port dev_null[1];
#else
    /* Point to links in hardware memeory map */
    Port north = (Port) 0x80000000;
    Port east = (Port) 0x80000004;
    Port south = (Port) 0x80000008;
    Port west = (Port) 0x8000000c;
    Port dev_null = (Port) 0x80000010;
#endif /* SELFHOSTED */

/* Shared Memory */
#ifdef SELFHOSTED
    /* For self-hosted testing, just allocate arrays */
    #include <stdio.h>

```




```
    Address northSharedMemory[SHARED_MEMORY_SIZE];
    Address eastSharedMemory[SHARED_MEMORY_SIZE];
    Address southSharedMemory[SHARED_MEMORY_SIZE];
    Address westSharedMemory[SHARED_MEMORY_SIZE];
#else
    /* else define shared memory addresses corresponding to the hardware */
    Address northSharedMemory = (Address) LOCAL_MEMORY_SIZE;
    Address eastSharedMemory = (Address) (LOCAL_MEMORY_SIZE +
SHARED_MEMORY_SIZE);
    Address southSharedMemory = (Address) (LOCAL_MEMORY_SIZE +
(2*SHARED_MEMORY_SIZE));
    Address westSharedMemory = (Address) (LOCAL_MEMORY_SIZE +
(3*SHARED_MEMORY_SIZE));
#endif /* SELFHOSTED */

#endif /* _CMPWARETORUS_H_ */
```

