

FFT II: 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 Fast Fourier Transform (FFT) Application II

The second implementation of the *Fast Fourier Transform (FFT)* application extends the ideas in the previous example applications. In particular this application revisits the FFT application using only Shared Register communication channels instead of shared memory. While some algorithms such as the FFT lend themselves well to shared memory implementations, the synchronization and control issues tend to make correct implementation and debug more difficult. This version of the FFT algorithm uses Shared Register communications channels to copy the blocks of data into local memory instead of using the shared memory approach. At the end of this document, the performance impact of this approach will be evaluated.



The *FFT II* code is all in the compressed ZIP archive under the *FFT-2* 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 FFT II Application

The FFT algorithm is used extensively in digital signal processing. It is used to convert time domain signals to the frequency domain where various types of frequency oriented filtering can take place. The FFT algorithm is more compute intensive than the other examples discussed for the *Cmpware CMP-DK* and also tends to have a more complex communication pattern. The details of the FFT algorithm are beyond the scope of this document and will not be covered here, but any standard textbook on digital signal processing should cover the FFT adequately.

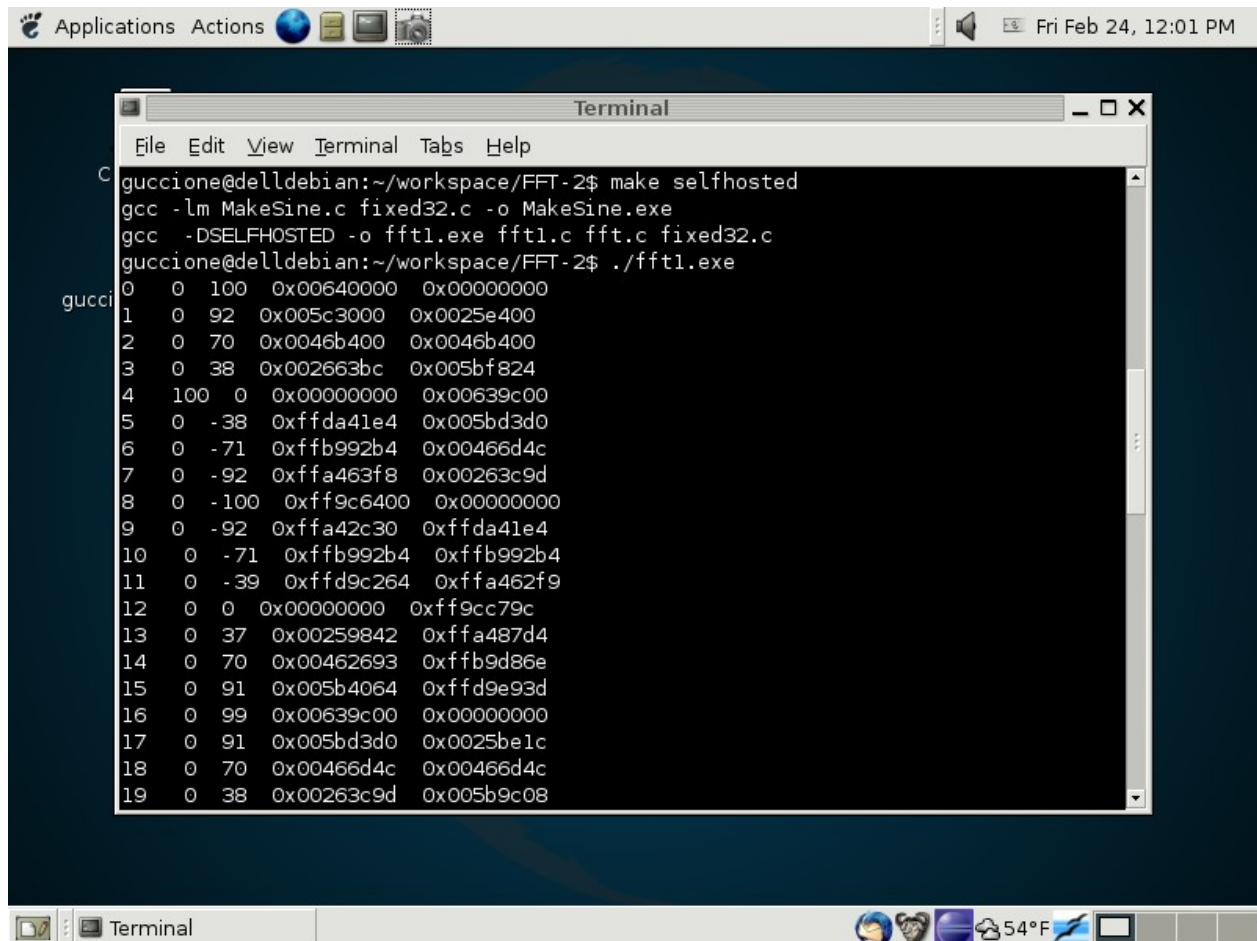
The source code for the implementation used in this example is include in the appendicies at the end of this document. The implementation is fairly straight forward and every attempt was made to keep the code simple and readable. Some study of this code may be useful for readers more interested in implementation details. Again, more efficient approaches are possible; this code was intended primarily to be used for demonstration purposes.

As with the other demonstration algorithms, a single processor version of the FFT 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 seperated from the process of parallelization. It also creates a 'benchmark' for future comparisons,

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. Note that this single processor version of the FFT is identical to the implementation discussed in the previous FFT document.



Figure 1 shows the building and execution of the single processor FFT 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.



```
guccione@delldebian:~/workspace/FFT-2$ make selfhosted
gcc -lm MakeSine.c fixed32.c -o MakeSine.exe
gcc -DSELFHOSTED -o fft1.exe fft1.c fft.c fixed32.c
guccione@delldebian:~/workspace/FFT-2$ ./fft1.exe
0 0 100 0x00640000 0x00000000
1 0 92 0x005c3000 0x0025e400
2 0 70 0x0046b400 0x0046b400
3 0 38 0x002663bc 0x005bf824
4 100 0 0x00000000 0x00639c00
5 0 -38 0xffda41e4 0x005bd3d0
6 0 -71 0xffb992b4 0x00466d4c
7 0 -92 0xffa463f8 0x00263c9d
8 0 -100 0xff9c6400 0x00000000
9 0 -92 0xffa42c30 0xffda41e4
10 0 -71 0xffb992b4 0xffb992b4
11 0 -39 0xffd9c264 0xffa462f9
12 0 0 0x00000000 0xff9cc79c
13 0 37 0x00259842 0xffa487d4
14 0 70 0x00462693 0xffb9d86e
15 0 91 0x005b4064 0xffd9e93d
16 0 99 0x00639c00 0x00000000
17 0 91 0x005bd3d0 0x0025be1c
18 0 70 0x00466d4c 0x00466d4c
19 0 38 0x00263c9d 0x005b9c08
```

Figure 1: The self hosted FFT II application.

The source code to this single processor version is in *Appendix E* at the end of this document. Note that this code contains only the main routines. The supporting FFT routines as well as the 32-bit fixed point support is used extensively and can be found in *Appendices A* through *D*. Note that all of these routines are standard serial,



uniprocessor function calls which can be typically found in any text or existing source code for the FFT algorithm implementation.

The output in Figure 1 has five columns and 64 rows of numbers. The first column is just the integers zero through 63. This is used to graph the X axis of the data. The next two columns are the real and imaginary portions of the FFT data. Only the real portion will be used in the graphing. Finally, the last two columns are the real and imaginary values repeated in hexadecimal format. The format of these values is 16 bits of integral and sixteen bits of fractional data.

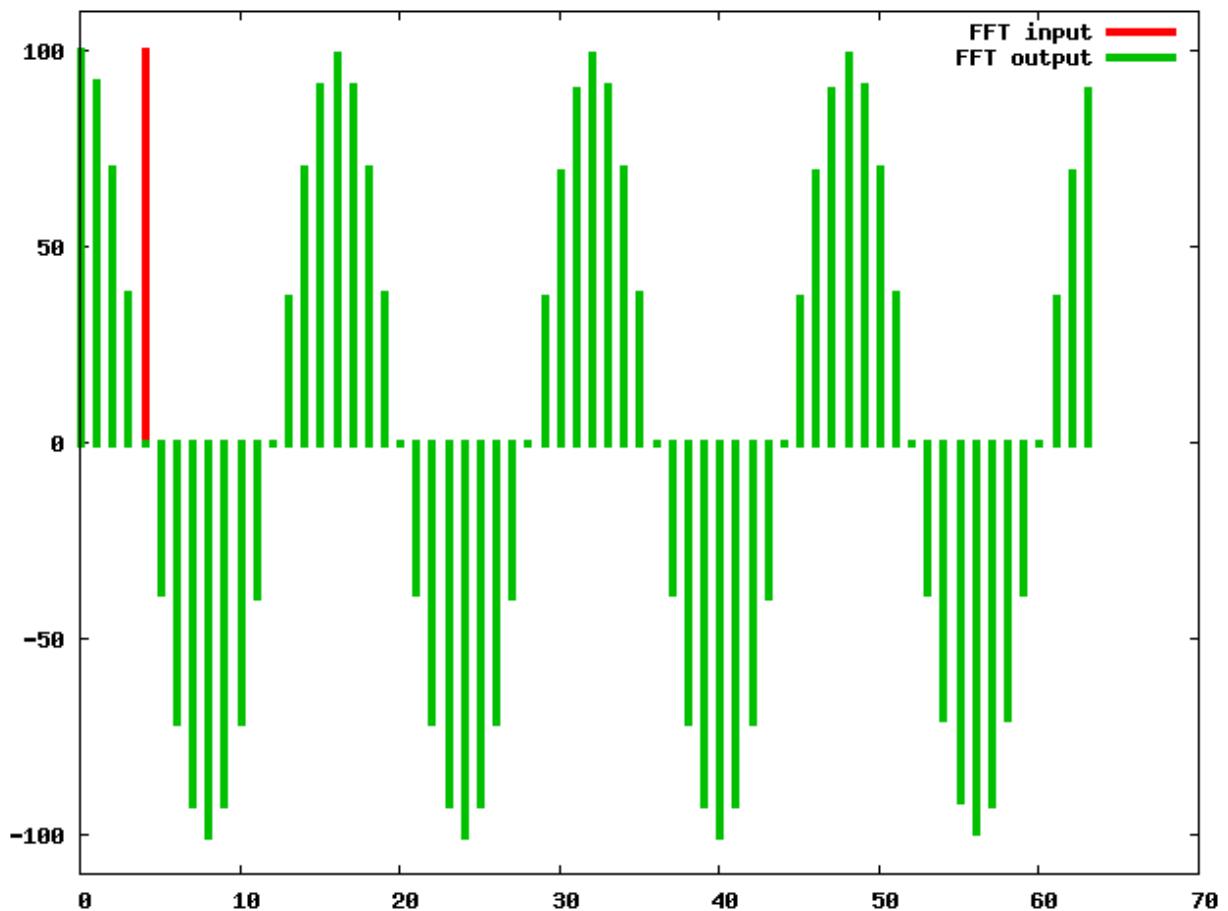


Figure 2: The self-hosted FFT filter input and output.

Also included in the source code for the FFT filter are files to graphically plot the output from this execution. If the results of the *fft1.exe* file are piped to a file named *fft.dat*, the



gnuplot plotting application may be used to graph the output of the FFT to verify that the functionality is correct. These files are supplied in the FFT demonstration directory, including a saved bitmap of the plot.

Figure 2 shows the plot generated with the command:

\$ gnuplot -persist fft.p

This plot shows that the input data is a sine wave going through four cycles with a peak of 100. The FFT filtered output data is a single impulse of height 100 at the value of four on the X axis, as expected.

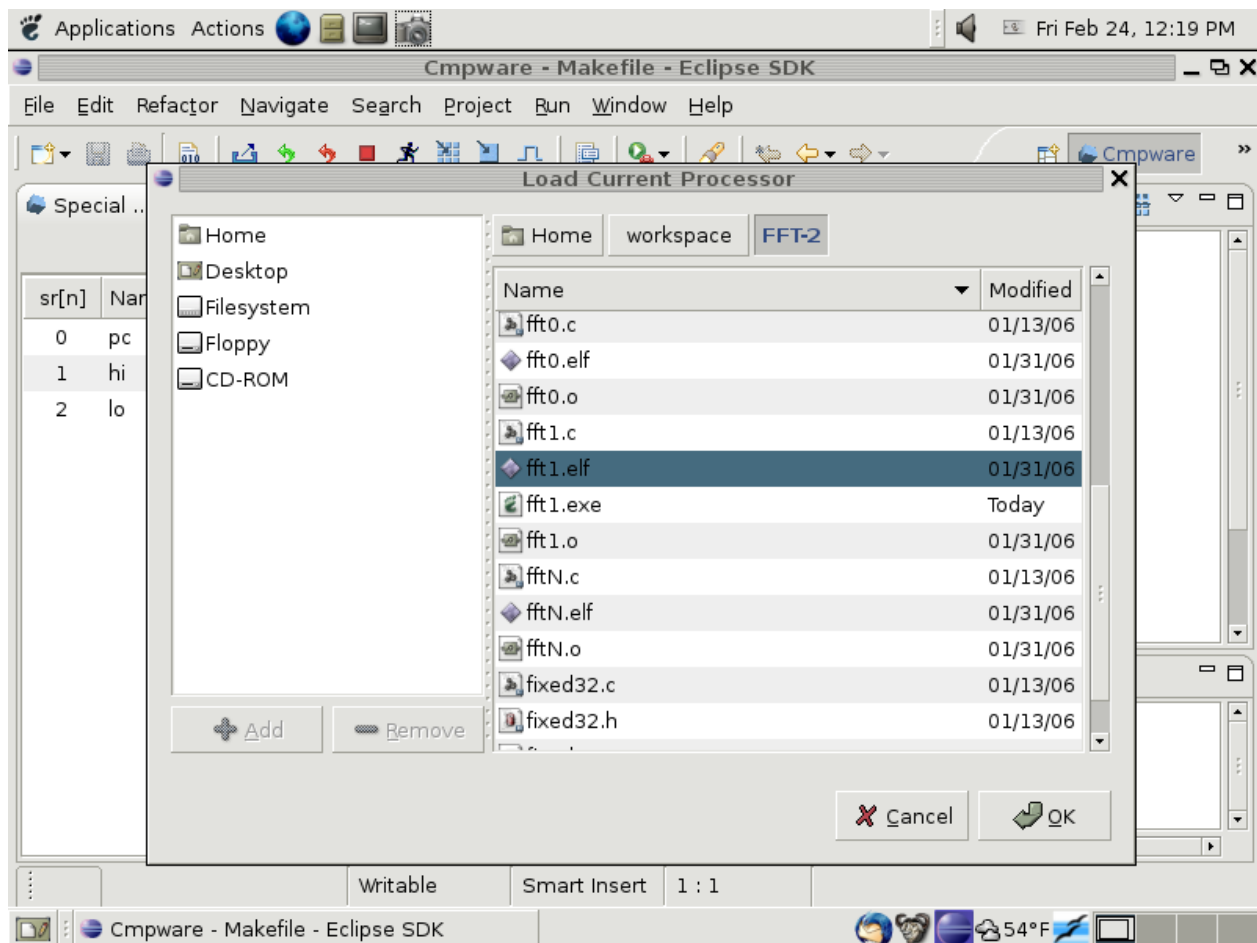


Figure 3: Loading *fft1.elf* into node (0,0).



Once the FFT filter code for a single processor has been successfully developed in a self-hosted uniprocessor 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.

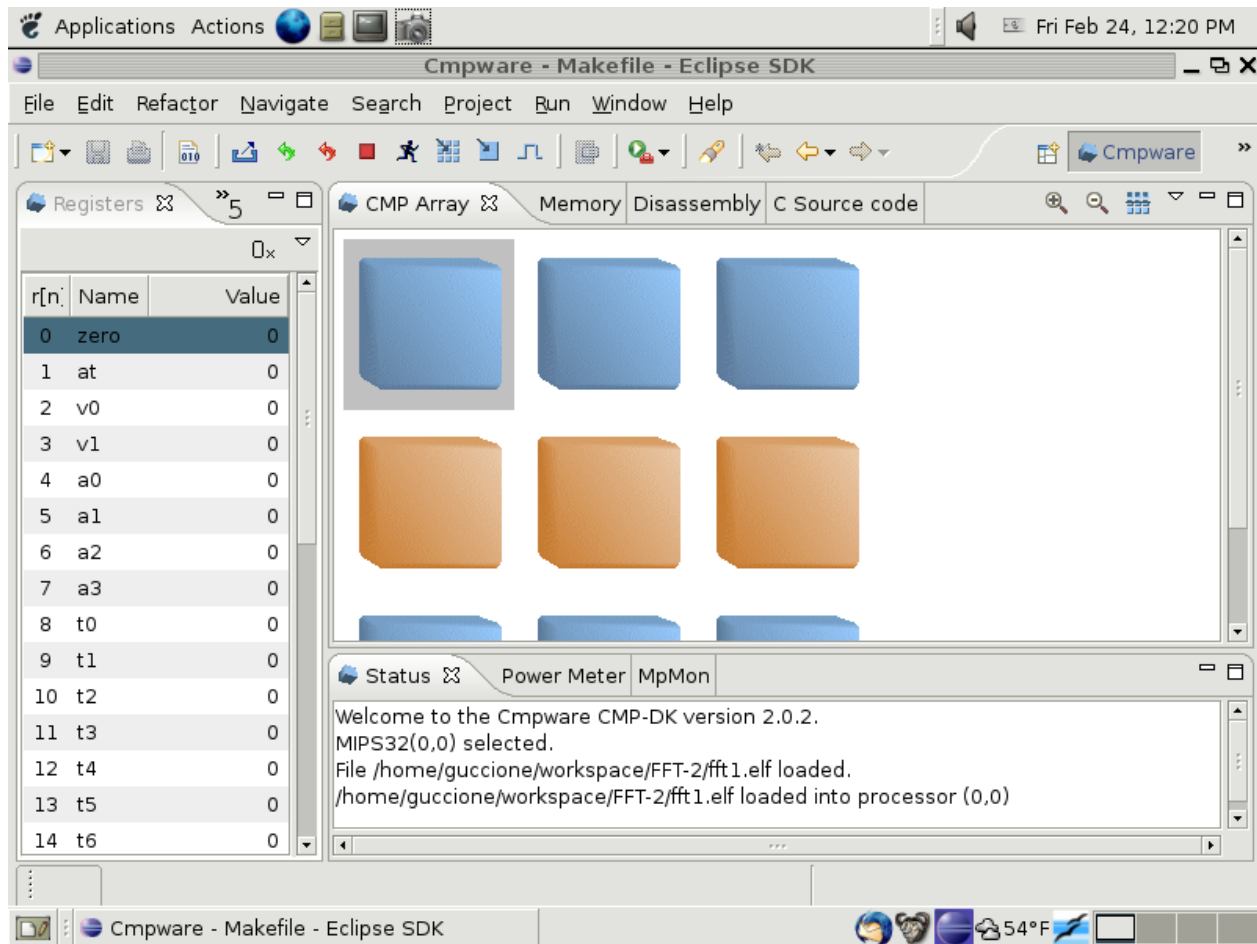


Figure 4: *fft1.elf* loaded into node (0,0).

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 FFT results to a pre-specified memory location. The compiled code for a *MIPS32* processor is supplied in the *FFT* directory



as *fft1.elf*.

Running the Single Processor FFT Application

To execute this uniprocessor FFT 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.

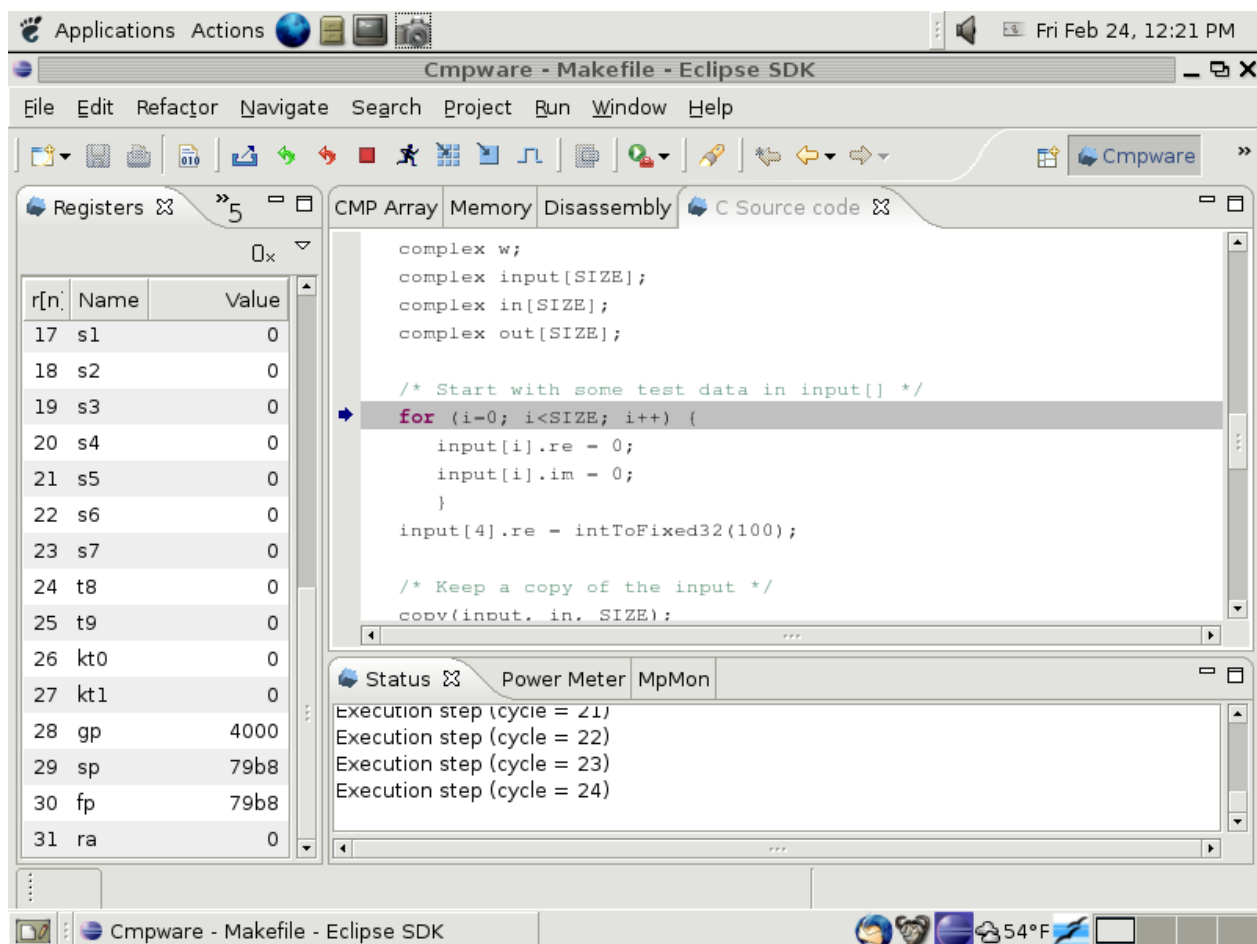


Figure 5: The single processor *fft1.elf* executing on node (0,0).

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.

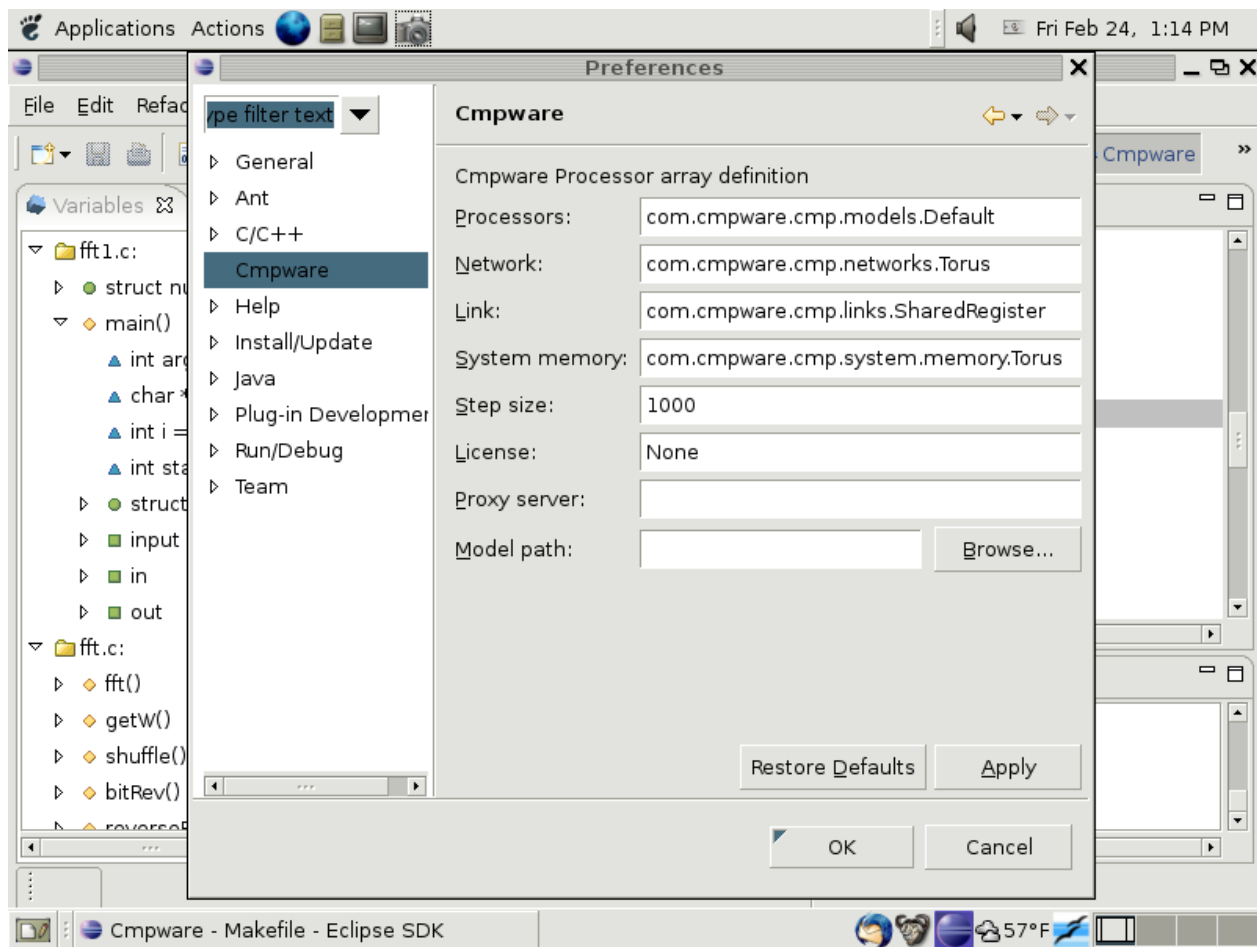


Figure 6: The *Cmpware CMP-DK* Preferences page.


Use the **Load** button () to bring up a file selection dialog. Using this file selection dialog, select the *fft1.elf* file from the list of files for the *FFT* 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 *fft1.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 24 cycles, as indicated by the **Status** window.

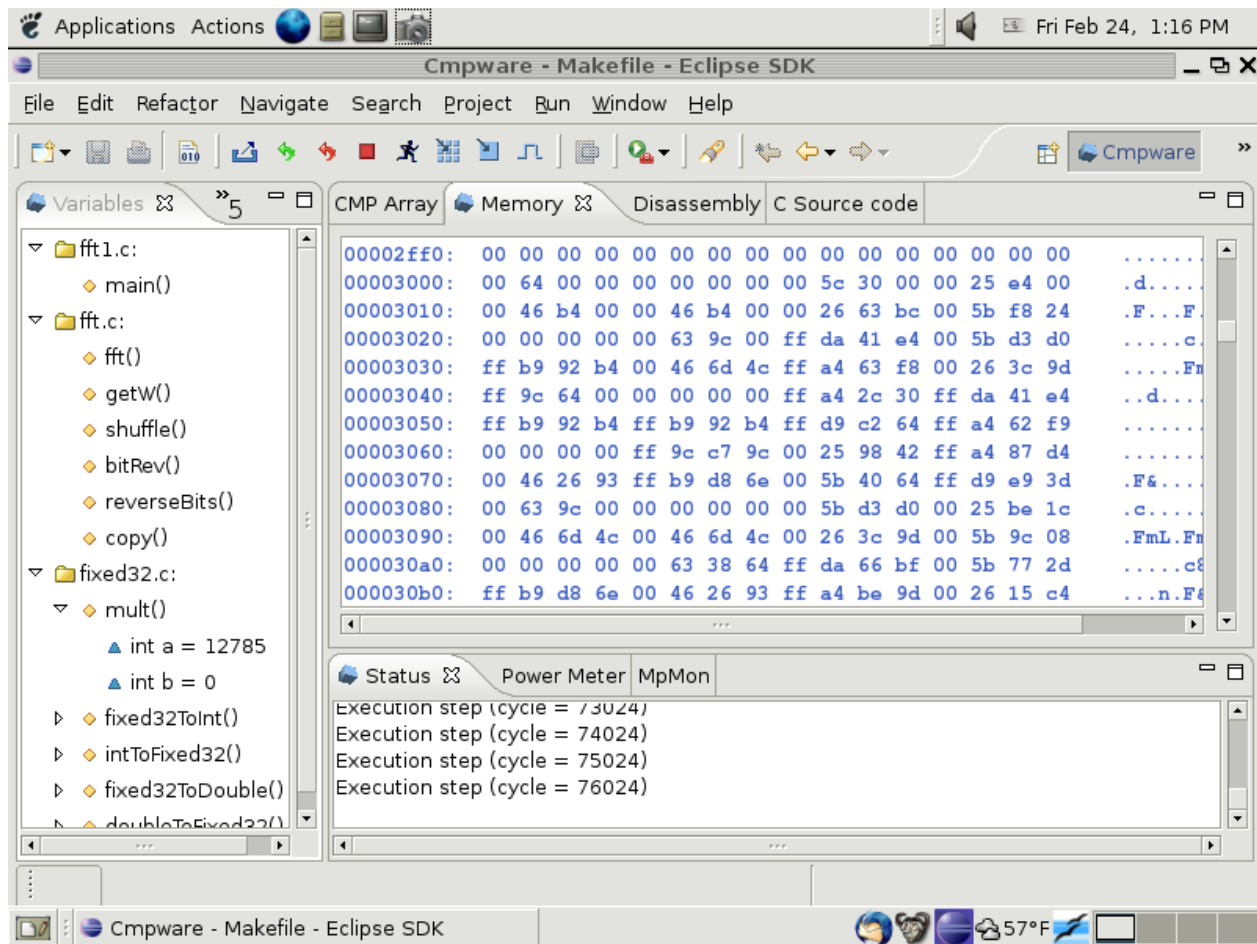




Figure 7: The single processor FFT results in local memory.

For larger applications such as the FFT 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 allows 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.

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, a block of data identical to those printed in the self hosted version can be seen being written to the memory. Note that this was the primary reason for printing the hexadecimal values in the self-hosted version. it makes it easier to compare to results to the ones displayed in the **Memory** view.

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 FFT filter is indeed functioning correctly on the *MIPS32* model and generating the correct results.

Parallelizing the FFT 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.

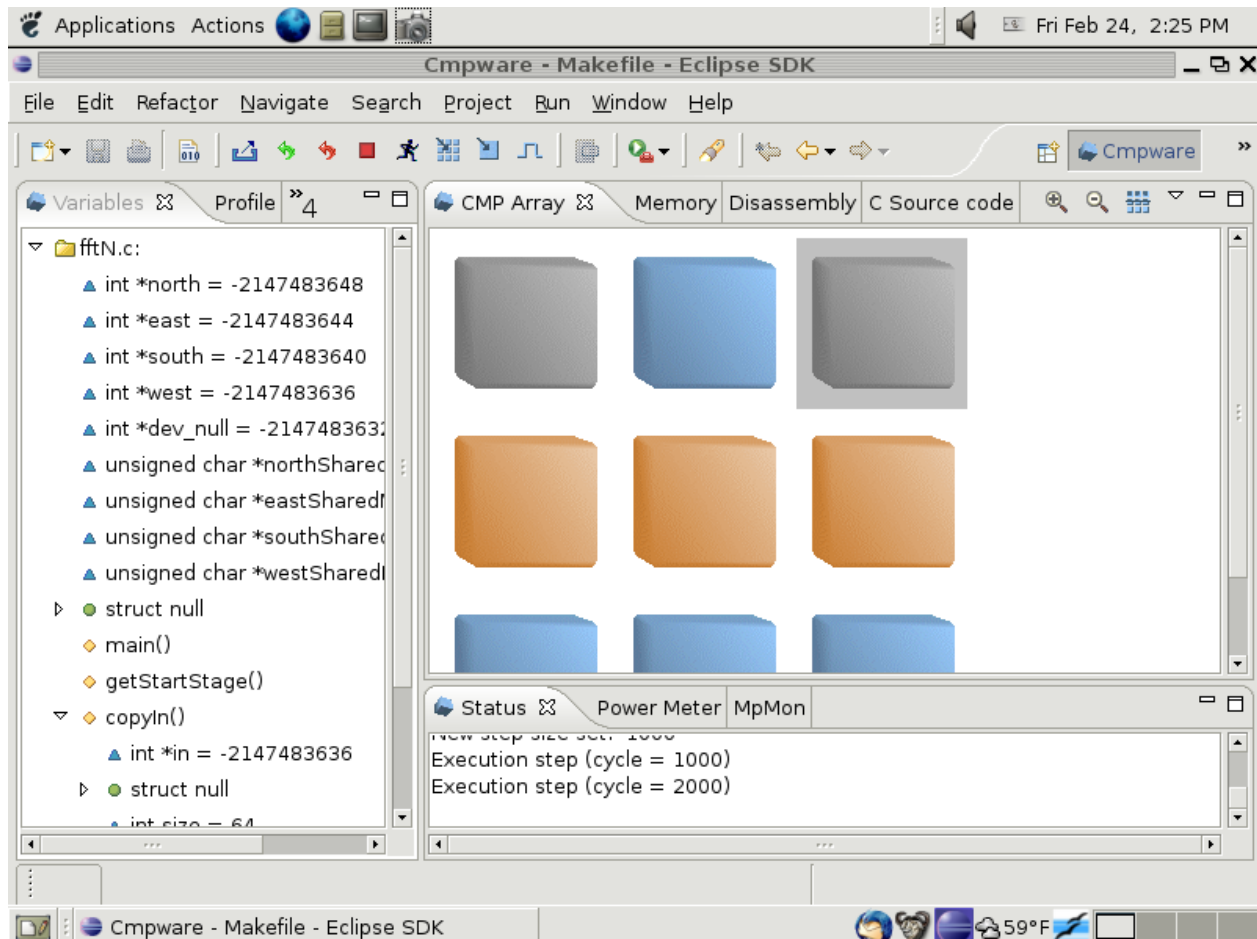


Figure 8: The FFT running on two nodes.

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.


While the previous FFT example made use of shared memory to send data from one processor to another, this implementation uses the Shared Register links. The links provide built-in synchronization when sharing data between processors, where shared memory does not. Using links permits the implicit synchronization of the processors that simplify not only the coding but the debug. While it is expected that there will be some additional overhead because of this approach, the benchmarks are given later in this document.

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 H* 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*.

As in the other examples, these registers exist in the processor memory map and 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 *Appendices* demonstrate how communication across processors is performed by a simple assignment to or from address pointers.

Because of this pointer access to the communication channels, it becomes fairly simple to parallelize this code from the *fft1.c* serial version. The regular serial functions in *fft.c* and *fixed32.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.

The main loop in the *fftN.c* source code in *Appendix G* is very similar to the single processor code. All that has changed is that partial result inputs now come from a pointer to a Shared Register and are sent to another pointer to a Shared Register. In addition, some temporary storage is used for these values. There are some other features of this code, but these will be discussed after the demonstration of the code executing.

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 *fft0.elf*. The next two processors, (0,1) and (0,2) are loaded with the *fftN.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.

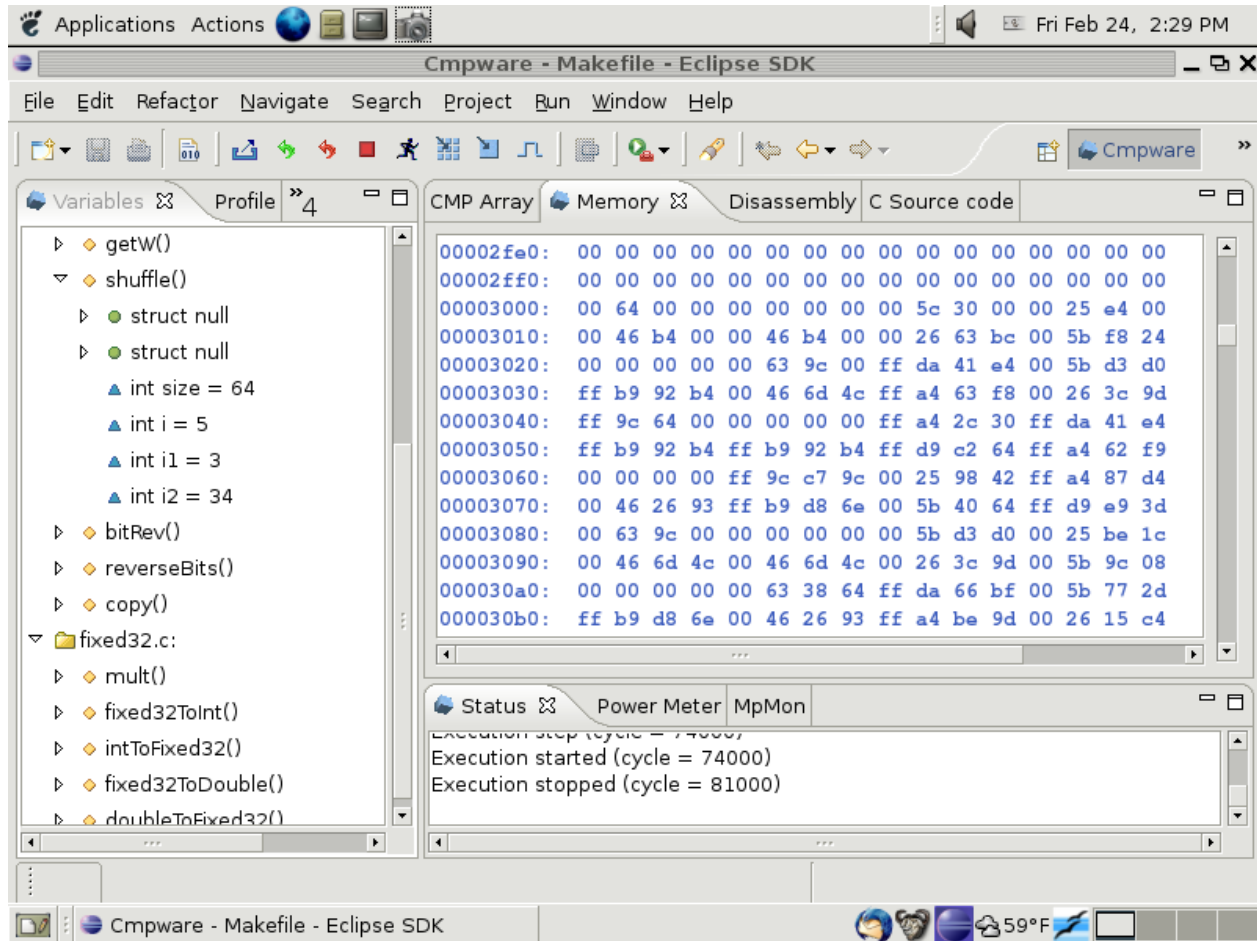


Figure 9: The two node FFT implementation results.

Unlike the uniprocessor versions of this software, the synchronous communication of the FFT 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 FFT calculation.

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.

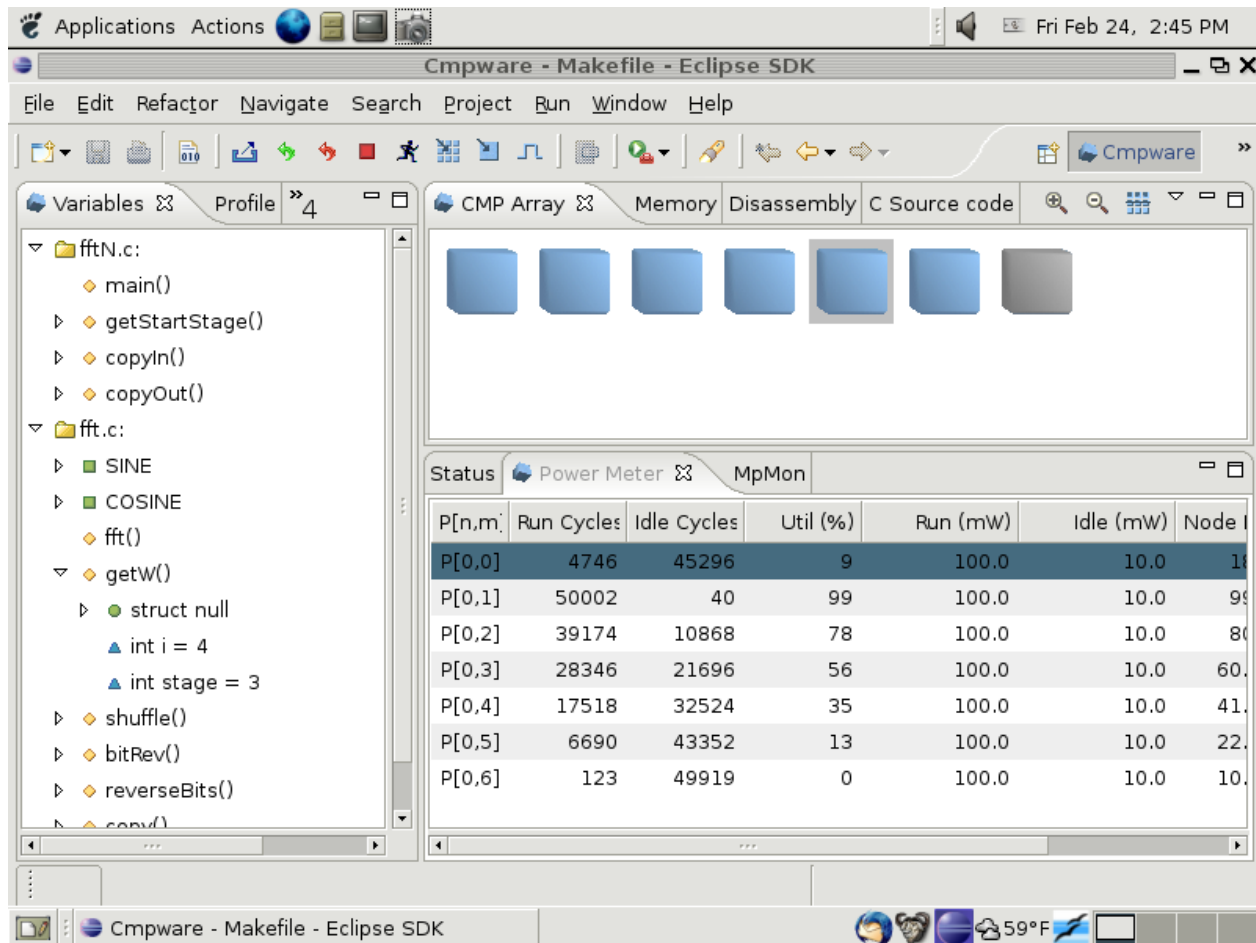


Figure 10: The six node FFT executing.

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. This can be compared to the results in either the self hosted or the single processor version of the FFT.

Figure 10 shows an six processor version of the FFT 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 six 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 numbers for all processors 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. In this example, a 64 point FFT is computed. The number of stages is determined by this value. The number of 'stages' of the FFT algorithm is the base two logarithm of the number of points in the FFT. In this case, the number of stages is six. Larger FFTs can use more processors, but only growing logarithmically with this approach. Additionally, it may be possible to exploit more parallelism within a stage. This is left as an exercise for the reader.

```
/* The FFT main loop */
for (stage=firstStage; stage<=lastStage; stage++) {

    /* On the first stage shuffle in from shared memory */
    /* else shuffle from local temp to local temp */
    if (stage == firstStage) {
        wait = *in; // Wait for data in shared memory
        shuffle(inmem, t2, fftSize);
        *in = 1; // Finished with shared memory
    } else
        shuffle(t1, t2, fftSize);
}
```

Figure 11: FFT data transfer using shared memory.

And as with the previous examples, it is also possible to calculate more than one stage per processor. So in this example, from one to six processors can be effectively employed. It is also interesting to point out that this allocation happens completely at run time and that no changes to the original *fftN.c* code have to be made. All that is required is that the parameter giving the number of processors in *fft0.c* be changed to the appropriate value.

Shared Memory Versus Shared Registers in the FFT Application

As discussed in the previous FFT example, using shared memory in a multiprocessor requires special care. Data cannot simply be written to and read from shared memory at will. There must be some form of 'handshaking' or synchronization between processors when data is transferred. With Shared Registers, however, only one word is transferred at a time and synchronization is built in to the Shared Register hardware.



While this eliminates the extra synchronization code using the Shared Register ports in the shared memory implementation, some additional work must be done to copy data from the Shared Register links to local memory where it can be used by the application.

```
/* Get the data from the port */
copyIn(in, t1, fftSize);

/* The FFT main loop */
for (stage=firstStage; stage<=lastStage; stage++) {

    shuffle(t1, t2, fftSize);

    ...
}
```

Figure 12: FFT data transfer using Shared Registers.

Figure 11 shows the code used by the shared memory FFT implementation. Note that the first stage is treated specially, in that data is shuffled in from the `inmem` shared memory, while the other stages use data in local memory `t1`. Also note the use of the `in` port to synchronize access to the `inmem` shared memory.

Similarly, Figure 12 shows the Shared Register implementation. Note that there is a one-time call to `copyIn()` to copy data from the Shared Register in to the local storage `t1`. After this, the `shuffle()` operates on the local data in both `t1` and `t2`. Also note that the synchronization is built in to the Shared Registers and explicit calls to the Shared Register ports bracketing the shared memory access are no longer required. Similar changes occur in the final stage of the computation where the result is passed on to the next processor.

While the code is somewhat simplified, it is expected that the extra copying of data may result in some performance penalty. Figure 13 shows a graph of both the shared memory and the Shared Register implementations of the FFT and their speedups as compared to the single processor case. Indeed, the shared memory implementation is somewhat faster than the Shared Register implementation, but only by a small factor. In fact, the difference in this application tends to be less than 10%. While this is significant, the performance gains must be weighed against other system parameters including reliability and development time. Also, shared memory will tend to be a more expensive hardware resource to implement.

In the case of the FFT, the difference between the Shared Register and shared memory implementations does not necessarily offer a clear choice. This may not be



the case, however with other algorithms. Some algorithms which are communication intensive may favor shared memory over Shared Registers. However, this is likely to be highly implementation dependent. The granularity and patterns of the communications between processors will determine the difference in performance in different communication styles.

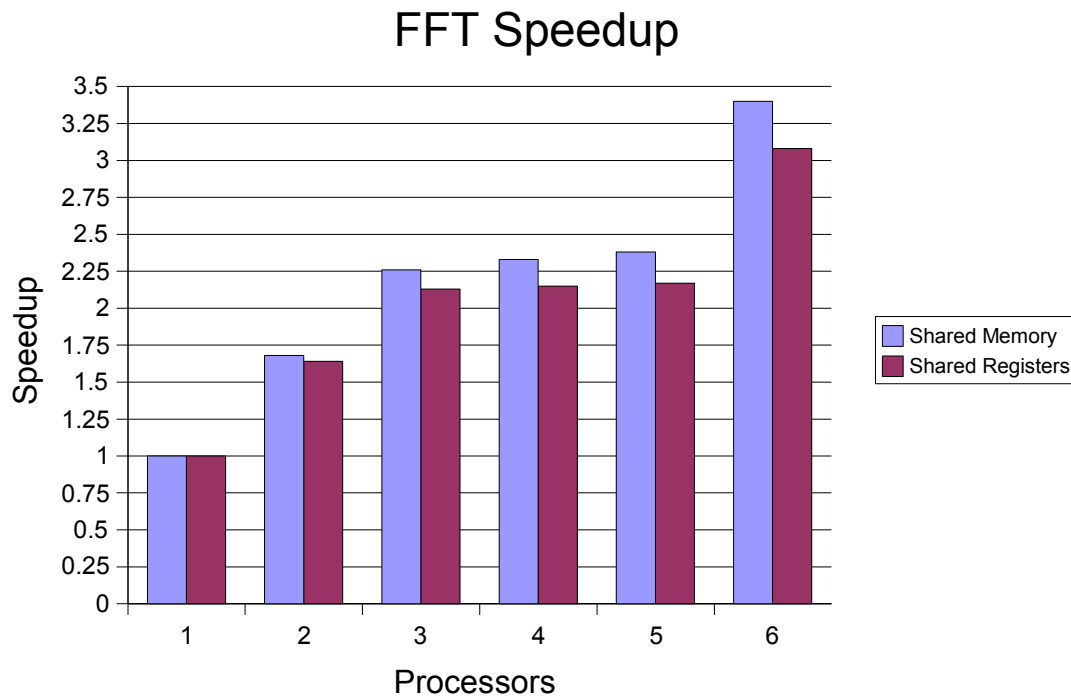


Figure 13: FFT speedup for shared memory versus Shared Register communication.

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: fft.h Source Code

```
#ifndef _FFT_H_
#define _FFT_H_

/*
** This defines the code used to implement both the uniprocessor
** and multiprocessor Fast Fourier Transform (FFT). Note that
** all code in this unit is serial.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#include "fixed32.h"

/* A complex number */
typedef struct {fixed32 re; fixed32 im;} complex;

/*
** This function performs the FFT 'butterfly' kernel. It takes
** in a 'twiddle factor' w and two complex inputs a and b and
** produces two complex results, c and d.
*/

void fft(complex w, complex a, complex b, complex *c, complex *d);

/*
** This method gets the 'twiddle' factor W. This currently
** assumes that the size of the SINE and COSINE tables are the
** same size as the FFT, otherwise a scale factor for the
** index should be used. This also masks off the last
** <stage> bits to get the W index. This works well with
** the algorithm organized as stages of shuffled outputs.
** Note that 0 <= i < (FFT_SIZE/2).
*/

void getW(complex *w, int i, int stage);

/*
** This functions performs a perfect shuffle of the
** array of complex numbers in in[] putting the result
** in out[]. These arrays should be disjoint. The
** result of this function is undefined if out[] and
** in[] overlap.
*/
```



```
void shuffle(complex in[], complex out[], int size);

/*
** This functions re-orders an array in bit-reverse
** order, as required by the final processing of the
** FFT. These arrays should be disjoint. The
** result of this function is undefined if out[] and
** in[] overlap.
*/

void bitRev(complex in[], complex out[], int size, int bits);

/*
** This function reverses the least significant <size> bits
** in an integer a.
*/

int reverseBits(int a, int size);

/*
** This function just copies one array to another. This
** is used primarily for debug.
*/

void copy(complex in[], complex out[], int size);

#endif /* _FFT_H_ */
```



Appendix B: fft.c Source Code

```
/*
** See fft.h for comments.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#include "fft.h"

/* The number of entries in the sin(x) and cos(x) table */
#define TABLE_ENTRIES 64

/* The sin(x) table (see end of file) */
extern fixed32 SINE[TABLE_ENTRIES];

/* The cos(x) table (see end of file) */
extern fixed32 COSINE[TABLE_ENTRIES];

void fft(complex w, complex a, complex b, complex *c, complex *d) {

    c->re = a.re + b.re;
    c->im = a.im + b.im;

    d->re = mult(w.re, (a.re - b.re)) - mult(w.im, (a.im - b.im));
    d->im = mult(w.re, (a.im - b.im)) + mult(w.im, (a.re - b.re));

} /* end fft() */

void getW(complex *w, int i, int stage) {
    w->re = COSINE[((i >> stage) << stage)];
    w->im = SINE[((i >> stage) << stage)];
} /* end getW() */

void shuffle(complex in[], complex out[], int size) {
    int i;
    int i1 = 0;
    int i2 = size / 2;

    for (i=0; i<size; i++)
        if ((i & 0x01) == 0) {
            out[i].re = in[i1].re;
            out[i].im = in[i1++].im;
        } else {
```



```

        out[i].re = in[i2].re;
        out[i].im = in[i2++].im;
    }

} /* end shuffle() */

void bitRev(complex in[], complex out[], int size, int bits) {
    int i;
    int rev_i;

    for (i=0; i<size; i++) {
        rev_i = reverseBits(i, bits);
        out[rev_i].re = in[i].re;
        out[rev_i].im = in[i].im;
    }

} /* end bitRev() */

int reverseBits(int a, int size) {
    int i;
    int bit;
    int result = 0;

    for (i=0; i<size; i++) {
        bit = (a >> i) & 0x01;
        result = (result << 1) | bit;
    }

    //printf("Bit reversal: 0x%x 0x%x\n", a, result);

    return (result);

} /* end reverseBits() */

void copy(complex in[], complex out[], int size) {
    int i;

    for (i=0; i<size; i++) {
        out[i].re = in[i].re;
        out[i].im = in[i].im;
    }

} /* end copy() */

/* A table of sin(x) from 0 to 2*PI or sin(2*PI*n/N) */
fixed32 SINE[TABLE_ENTRIES] = {
0, 6423, 12785, 19023, 25079, 30892, 36409, 41574,
46340, 50659, 54490, 57796, 60546, 62713, 64275, 65219,

```



```
65535, 65219, 64275, 62713, 60546, 57796, 54490, 50659,  
46340, 41574, 36409, 30892, 25079, 19023, 12785, 6423,  
0, -6423, -12785, -19023, -25079, -30892, -36409, -41574,  
-46340, -50659, -54490, -57796, -60546, -62713, -64275, -65219,  
-65535, -65219, -64275, -62713, -60546, -57796, -54490, -50659,  
-46340, -41574, -36409, -30892, -25079, -19023, -12785, -6423,  
};
```

```
/* A table of sin(x) from 0 to 2*PI or sin(2*PI*n/N) */  
fixed32 COSINE[TABLE_ENTRIES] = {  
65535, 65219, 64275, 62713, 60546, 57796, 54490, 50659,  
46340, 41574, 36409, 30892, 25079, 19023, 12785, 6423,  
0, -6423, -12785, -19023, -25079, -30892, -36409, -41574,  
-46340, -50659, -54490, -57796, -60546, -62713, -64275, -65219,  
-65535, -65219, -64275, -62713, -60546, -57796, -54490, -50659,  
-46340, -41574, -36409, -30892, -25079, -19023, -12785, -6423,  
0, 6423, 12785, 19023, 25079, 30892, 36409, 41574,  
46340, 50659, 54490, 57796, 60546, 62713, 64275, 65219,  
};
```



Appendix C: fixed32.h Source Code

```
#ifndef _FIXED32_H_
#define _FIXED32_H_

/*
** This defines a 32-bit fixed point implementation with 16 bits
** of integral and 16 bits of fractional value.
**
** Copyright (c) 2005 Cmpware, Inc. All rights reserved.
**
*/

/* A 32 bit fixed point number with 16 bit integer, 16 bit fractional parts */
typedef int fixed32;

/*
** This multiplies two fixed32 numbers and returns a
** fixed32.
*/

fixed32 mult(fixed32 a, fixed32 b);

/*
** This converts a fixed32 value to an integer. The fractional
** portion is truncated.
*/

int fixed32ToInt(fixed32 f);

/*
** This converts an integer to a fixed32.
*/

fixed32 intToFixed32(int i);

/*
** This converts a fixed32 to a double precision float.
*/

double fixed32ToDouble(fixed32 f);

/*
** This converts a floating point double to a fixed32.
*/
```



```
*/  
fixed32 doubleToFixed32(double d);  
  
#endif /* _FIXED32_H_*/
```



Appendix D: fixed32.c Source Code

```
/*
** This defines a 32-bit fixed point implementation with 16 bits
** of integral and 16 bits of fractional value.
**
** Copyright (c) 2005 Cmpware, Inc. All rights reserved.
**
*/

#include "fixed32.h"

fixed32 mult(fixed32 a, fixed32 b) {
    return ((a >> 8) * (b >> 8));
} /* end mult() */

int fixed32ToInt(fixed32 f) {
    return (f >> 16);
} /* end fixed32ToInt() */

fixed32 intToFixed32(int i) {
    return (i << 16);
} /* end intToFixed32() */

double fixed32ToDouble(fixed32 f) {
    return ( ((double) f) / (double) ((1 << 16)-1) );
} /* end fixed32ToDouble() */

fixed32 doubleToFixed32(double d) {
    return ( (fixed32) (d * ((double) ((1 << 16)-1))) );
} /* end doubleToFixed32() */
```



Appendix E: fft1.c Source Code

```
/*
** This program is used to implement the Fast Fourier
** Transform (FFT) using integer / fixed point arithmetic
** on a single node or on a self-hosted system. This is
** used to be sure the algorithm is solid and is used
** in a later parallelized version.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

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

#include "fft.h"

/* The size of the FFT */
#define SIZE 64

/* The number of FFT stages (must be log2(SIZE)) */
#define STAGES 6

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

int main(int argc, char *argv[]) {
    int i;
    int stage;
    complex w;
    complex input[SIZE];
    complex in[SIZE];
    complex out[SIZE];

    /* Start with some test data in input[] */
    for (i=0; i<SIZE; i++) {
        input[i].re = 0;
        input[i].im = 0;
    }
}
```



```
input[4].re = intToFixed32(100);

/* Keep a copy of the input */
copy(input, in, SIZE);

/* The main loop */
for (stage=0; stage<STAGES; stage++) {

    shuffle(in, out, SIZE);

    for (i=0; i<SIZE; i=i+2) {
        getW(&w, (i/2), stage);
        fft(w, out[i], out[i+1], &(in[i]), &(in[i+1]));
    } /* end for(i) */

} /* end for(stage) */

/* The final 'bit reversal' */
bitRev(in, result, SIZE, STAGES);

#ifdef SELFHOSTED

/* Print out the result (if we have a stdout) */
/* (in decimal for plotting, and hex to compare */
/* to the embeded / simulated results) */
for (i=0; i<SIZE; i++)
    printf("%d  %d  %d  0x%08x  0x%08x\n", i,
           fixed32ToInt(input[i].re), fixed32ToInt(result[i].re),
           result[i].re,  result[i].im);

#endif

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



Appendix F: fft0.c Source Code

```
/*
** This program sends data to the FFT filter. It is a
** simple loop which also repeats for benchmarking.
**
** Copyright (c) 2004, 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#include "CmpwareTorus.h"
#include "fft.h"

/* The number of processors */
/* (Can also be set with the -DPROCESSORS=n flag in gcc) */
#define PROCESSORS 2

/* The size of the FFT */
#define SIZE 64

/* The number of FFT stages (must be log2(SIZE))*/
#define STAGES 6

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

int main(int argc, char *argv[]) {
    int i;
    int j;
    int wait;
    Port out = east; // Set up the output port

    /* Send out parameters */
    *out = PROCESSORS; // Number of processors
    *out = 0; // The first processor number
    *out = SIZE; // FFT size
    *out = STAGES; // FFT Stages (log2(SIZE))

    /* Put data in shared memory */
    for (i=0; i<REPEAT; i++) {

        /* Put data in shared memory */
        for (j=0; j<SIZE; j++) {

            /* Real part of input (impulse at 4) */
            if (j == 4)
                *out = intToFixed32(100);
            else

```



```
        *out = 0;
        /* Imaginary part of input */
        *out = 0;

    } /* end for(j) */

} /* end for(i) */

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

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



Appendix G: fftN.c Source Code

```
/*
** This program is used to implement the Fast Fourier
** Transform (FFT) using integer / fixed point arithmetic.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

/* Define SELFHOSTED to run on standard host */
/* undefine to run on simulator / hardware */
// #define SELFHOSTED

#include "CmpwareTorus.h"
#include "fft.h"
#include "fixed32.h"

/* The largest possible FFT */
#define MAX_FFT 1024

/* A place to put the result */
static complex *result = (complex *) 0x3000;

/* Function prototypes */
int getStartStage(int thisProcessor, int processors, int stages);
void copyIn(Port in, complex out[], int size);
void copyOut(complex in[], Port out, int size);

int main(int argc, char *argv[]) {
    int i = 0;
    int processors;
    int thisProcessor;
    int fftSize;
    int fftStages;
    int firstStage = 0; // The first stage of the FFT done on this node
    int lastStage = 5; // The last stage of the FFT done on this node
    int stage;
    complex t1[MAX_FFT];
    complex t2[MAX_FFT];
    complex w;

    Port out = east;
    Port in = west;

    /* Get the parameters */
    processors = *in; // The number of processors
```




```

thisProcessor = *in;    // This processor number
fftSize = *in;        // The size of the FFT
fftStages = *in;      // The number of FFT stages (log2(fftSize))

/* If this is the last node, the output port is dev_null */
if (thisProcessor >= (processors-1))
    out = dev_null;

/* Send the parameters to the next node */
*out = processors;      // Number of processors
*out = (thisProcessor+1); // Number of next processor
*out = fftSize;        // The size of the FFT
*out = fftStages;     // The number of FFT stages (log2(fftSize))

/* Which stages are being done on this processor? */
firstStage = getStartStage(thisProcessor, processors, fftStages);
lastStage = getStartStage(thisProcessor+1, processors, fftStages) - 1;

/* Loop as long as data is available */
for (;;) {

    /* Get the data from the port */
    copyIn(in, t1, fftSize);

    /* The FFT main loop */
    for (stage=firstStage; stage<=lastStage; stage++) {

        shuffle(t1, t2, fftSize);

        for (i=0; i<fftSize; i=i+2) {
            getW(&w, (i/2), stage);
            fft(w, t2[i], t2[i+1], &(t1[i]), &(t1[i+1]));
        } /* end for(i) */

    } /* end for(stage) */

    /* The final 'bit reversal' in last stage */
    if (lastStage == (fftStages-1))
        bitRev(t1, result, fftSize, fftStages);
    else
        copyOut(t1, out, fftSize);

    i = 0; // place for a breakpoint

} /* end for(;;) */

} /* end main() */

/*
** This method is used to spread stages across processors
** evenly, even when the number of processors is not an

```



```
** even multiple of the number of stages. This method gives
** the number of the first stage to be executed on a processor.
**
** @param thisProcessor The processor number, starting
**         at zero.
**
** @param processors The number of processors.
**
** @param rounds The number of stages.
**
** @return The number of first stage to be performed on
**         this processor.
**
*/

int getStartStage(int thisProcessor, int processors, int stages) {
    return ((thisProcessor * stages) / processors);
} /* end getStartStage() */

/**
** This method copies data in from a port and stores
** it in a local array.
**/

void copyIn(Port in, complex out[], int size) {
    int i;

    for (i=0; i<size; i++) {
        out[i].re = *in;
        out[i].im = *in;
    }

} /* end copyIn() */

/**
** This method copies data out from a local array to a
** port.
**/

void copyOut(complex in[], Port out, int size) {
    int i;

    for (i=0; i<size; i++) {
        *out = in[i].re;
        *out = in[i].im;
    }

} /* end copyOut() */
```



Appendix H: 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 memory 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_ */
```

