

# FIR: 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](mailto: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](http://www.cmpware.com/Apps/CmpwareApps_2_0.zip)

## The FIR Filter Application

The *Finite Impulse Response (FIR)* filter extends the ideas in the previous example applications. In particular this offers a more 'real world' example of a popular Digital Signal Processing (DSP) algorithm. Some further ideas in parallelizing applications, including parameterization, will also be examined.

The FIR filter is a commonly used DSP filter typical in audio and other applications. The FIR filter has a fairly simple structure that makes it highly parallelizable and suitable for a hardware implementation. This structure also makes it fairly simple to parallelize in a single chip multiprocessor.



The *FIR* code is all in the compressed ZIP archive under the *FIR* 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 FIR Application

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, is more complex and will use real input data to produce a real output.

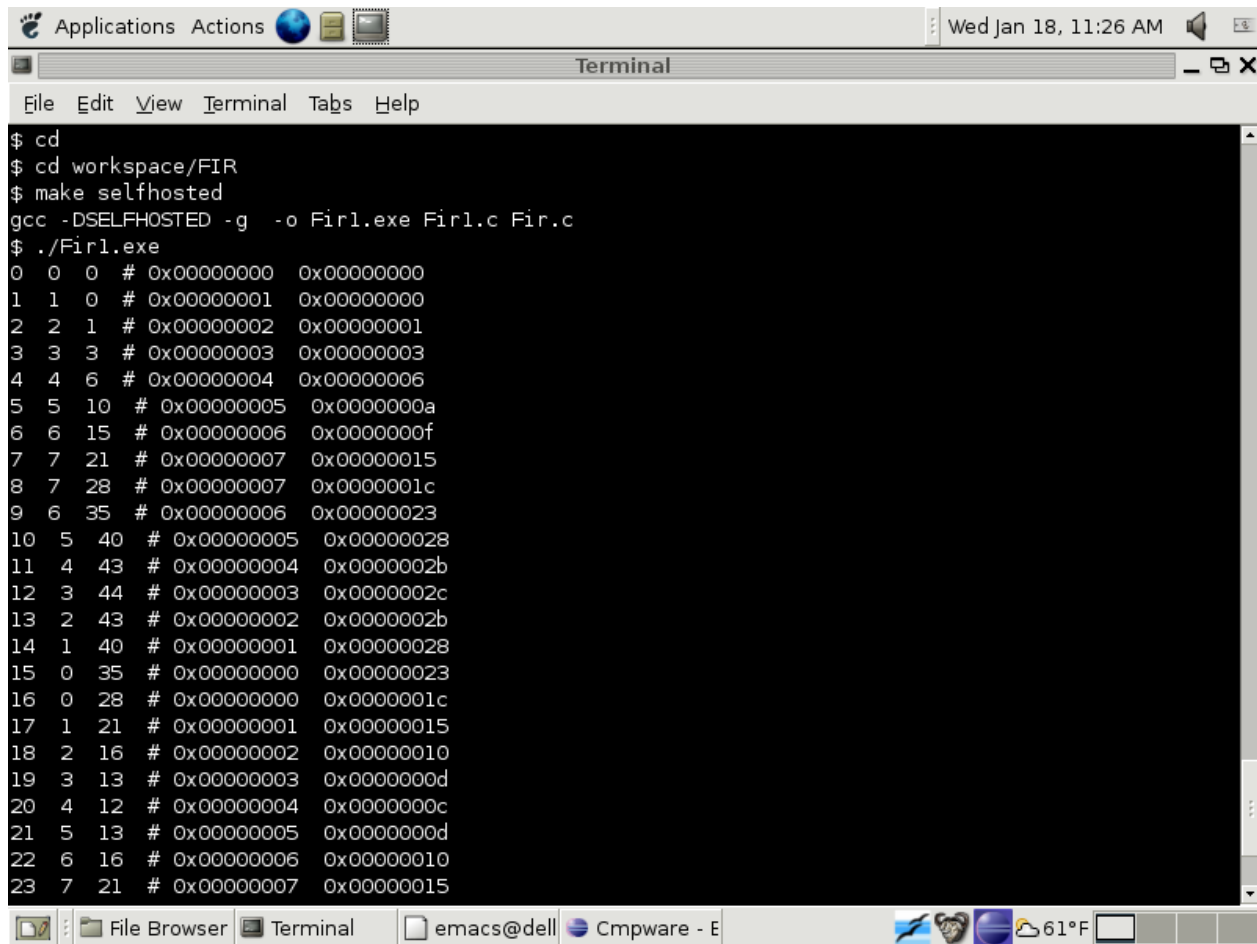
In order to simplify the development process and to create a 'benchmark' for future comparisons, a single processor version of the FIR filter is first developed. This approach has several benefits. First, developing a single processor version of any algorithm is typically much easier than building a 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 FIR filter 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 A* at the end of this document. 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.



Also included in the source code for the FIR filter are files to graphically plot the output from this execution. If the results of the *FIR1.exe* file are piped to a file named *fir.dat*, the *gnuplot* plotting application may be used to graph the output of the FIR filter to verify that the functionality is correct. These files are supplied in the FIR demonstration directory, including a saved bitmap of the plot.



The screenshot shows a terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Tabs, Help) and a system tray at the bottom. The terminal output is as follows:

```
$ cd
$ cd workspace/FIR
$ make selfhosted
gcc -DSELFHOSTED -g -o Fir1.exe Fir1.c Fir.c
$ ./Fir1.exe
0 0 0 # 0x00000000 0x00000000
1 1 0 # 0x00000001 0x00000000
2 2 1 # 0x00000002 0x00000001
3 3 3 # 0x00000003 0x00000003
4 4 6 # 0x00000004 0x00000006
5 5 10 # 0x00000005 0x0000000a
6 6 15 # 0x00000006 0x0000000f
7 7 21 # 0x00000007 0x00000015
8 7 28 # 0x00000007 0x0000001c
9 6 35 # 0x00000006 0x00000023
10 5 40 # 0x00000005 0x00000028
11 4 43 # 0x00000004 0x0000002b
12 3 44 # 0x00000003 0x0000002c
13 2 43 # 0x00000002 0x0000002b
14 1 40 # 0x00000001 0x00000028
15 0 35 # 0x00000000 0x00000023
16 0 28 # 0x00000000 0x0000001c
17 1 21 # 0x00000001 0x00000015
18 2 16 # 0x00000002 0x00000010
19 3 13 # 0x00000003 0x0000000d
20 4 12 # 0x00000004 0x0000000c
21 5 13 # 0x00000005 0x0000000d
22 6 16 # 0x00000006 0x00000010
23 7 21 # 0x00000007 0x00000015
```

Figure 1: The self hosted FIR application.

Figure 2 shows the plot generated with the command:

```
$ gnuplot -persist fir.p
```

This plot shows that the input data is a triangle or sawtooth wave. The FIR filtered output is a smoothed version of this data, as expected. Note that the final result is not



scaled and that the FIR parameters are all '1'. This can be changed to experiment with the filter implementation. These values are set in arrays at the beginning of the *Fir1.c* source code file in *Appendix A*.

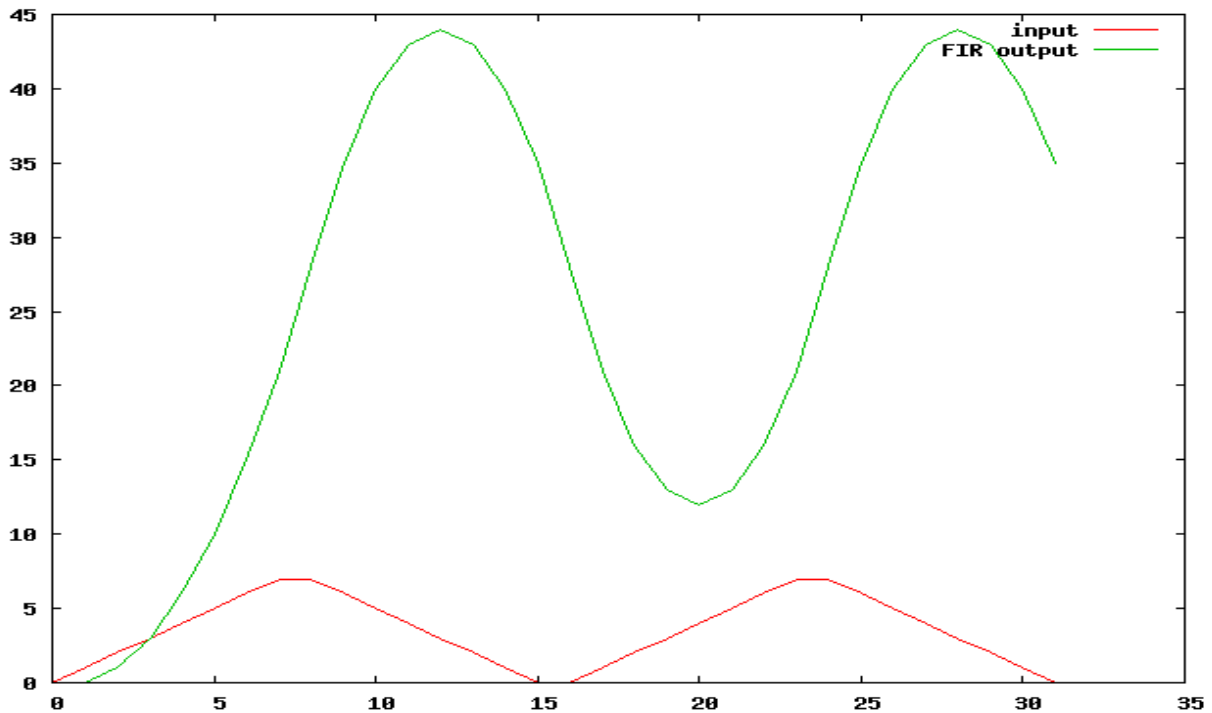


Figure 2: The self-hosted FIR filter input and output.

Once the FIR 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 operated 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 FIR results to a pre-specified memory location. The compiled code for a *MIPS32* processor is supplied in the *FIR* directory as *Fir1.elf*.



## Running the Single Processor FIR Application

To execute this uniprocessor FIR 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.

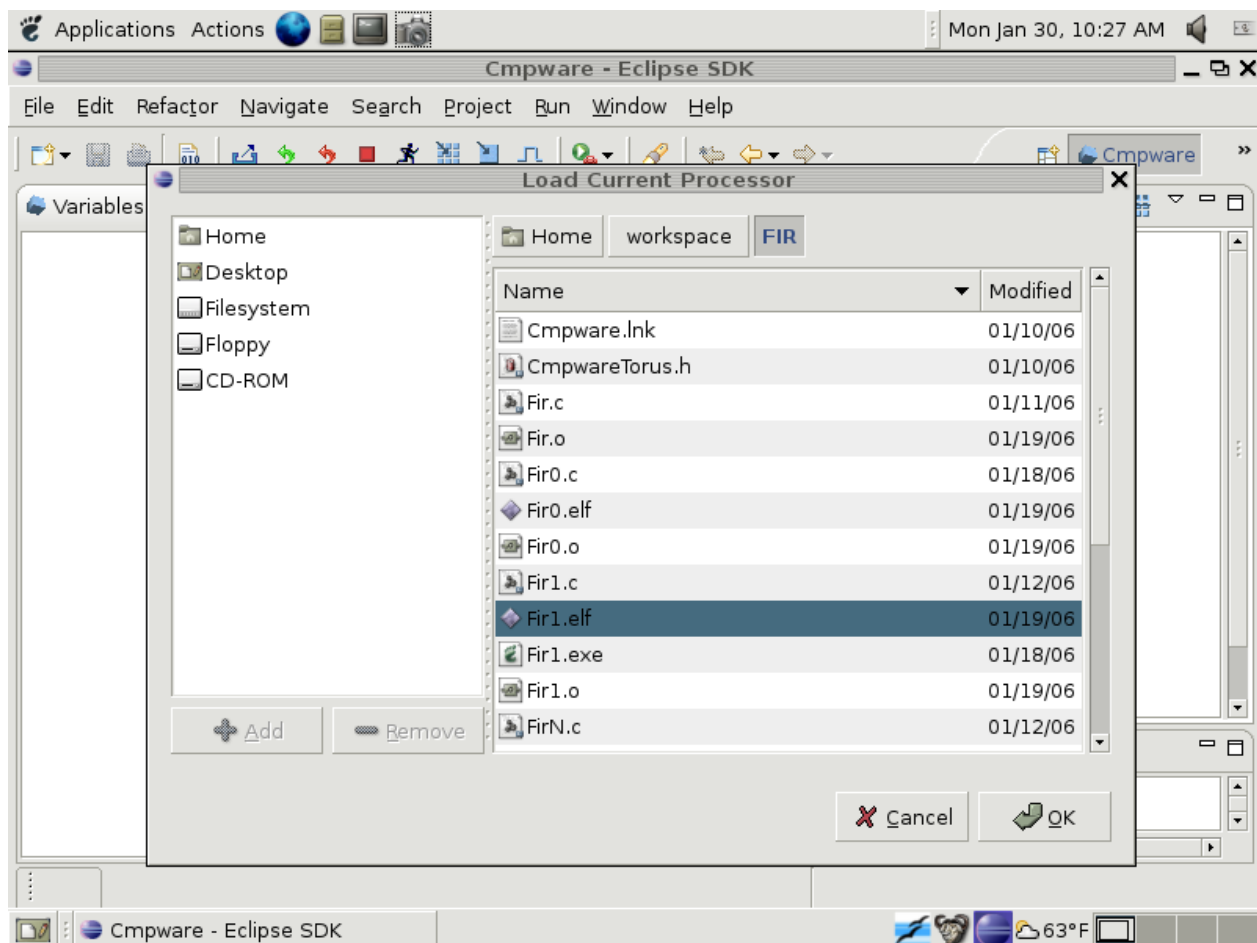


Figure 3: Loading *Fir1.elf* into node (0,0).

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.

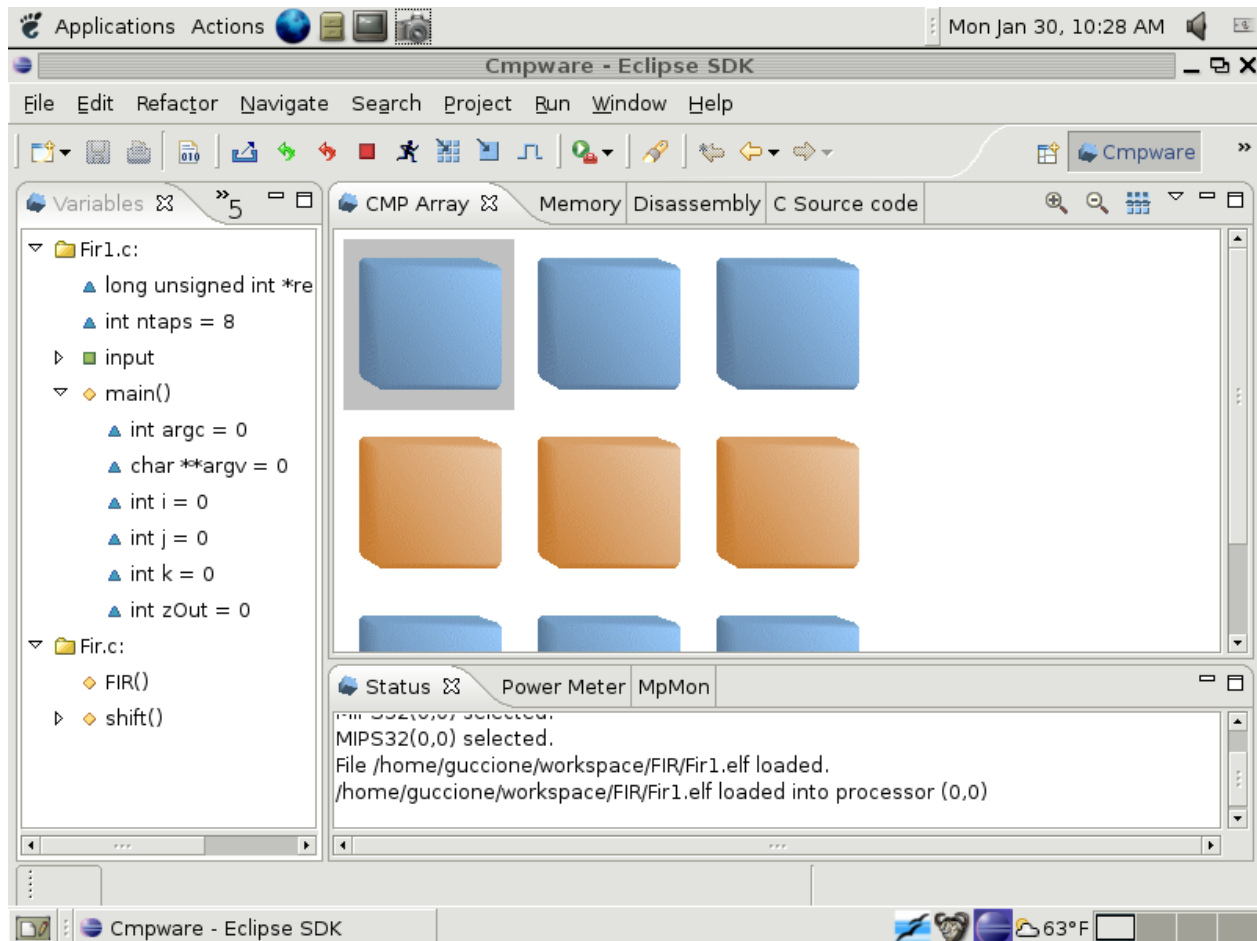


Figure 4: *Fir1.elf* loaded into node (0,0).



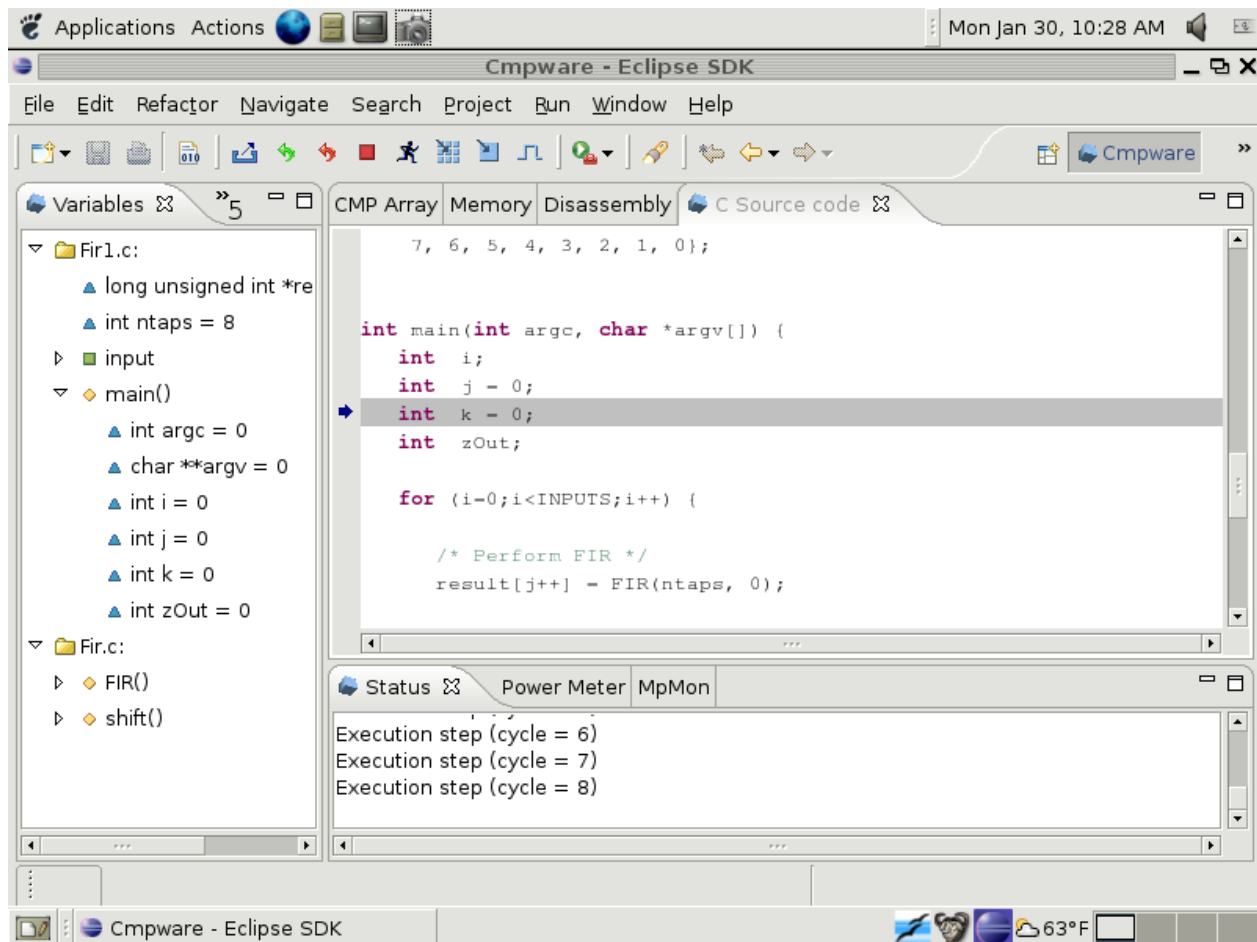

Use the **Load** button (  ) to bring up a file selection dialog. Using this file selection dialog, select the *Fir1.elf* file from the list of files for the *FIR* demonstration as shown in Figure 3. A message in the **Status** window at the bottom of the IDE should indicate that the file was successfully loaded into the MIPS32 processor at location (0,0) as in



Figure 4.

At this point, the executable file *FIR1.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 8 cycles, as indicated by the **Status** window.

Figure 5: *Fir1.elf* executing on node (0,0).

For larger applications such as the FIR 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.

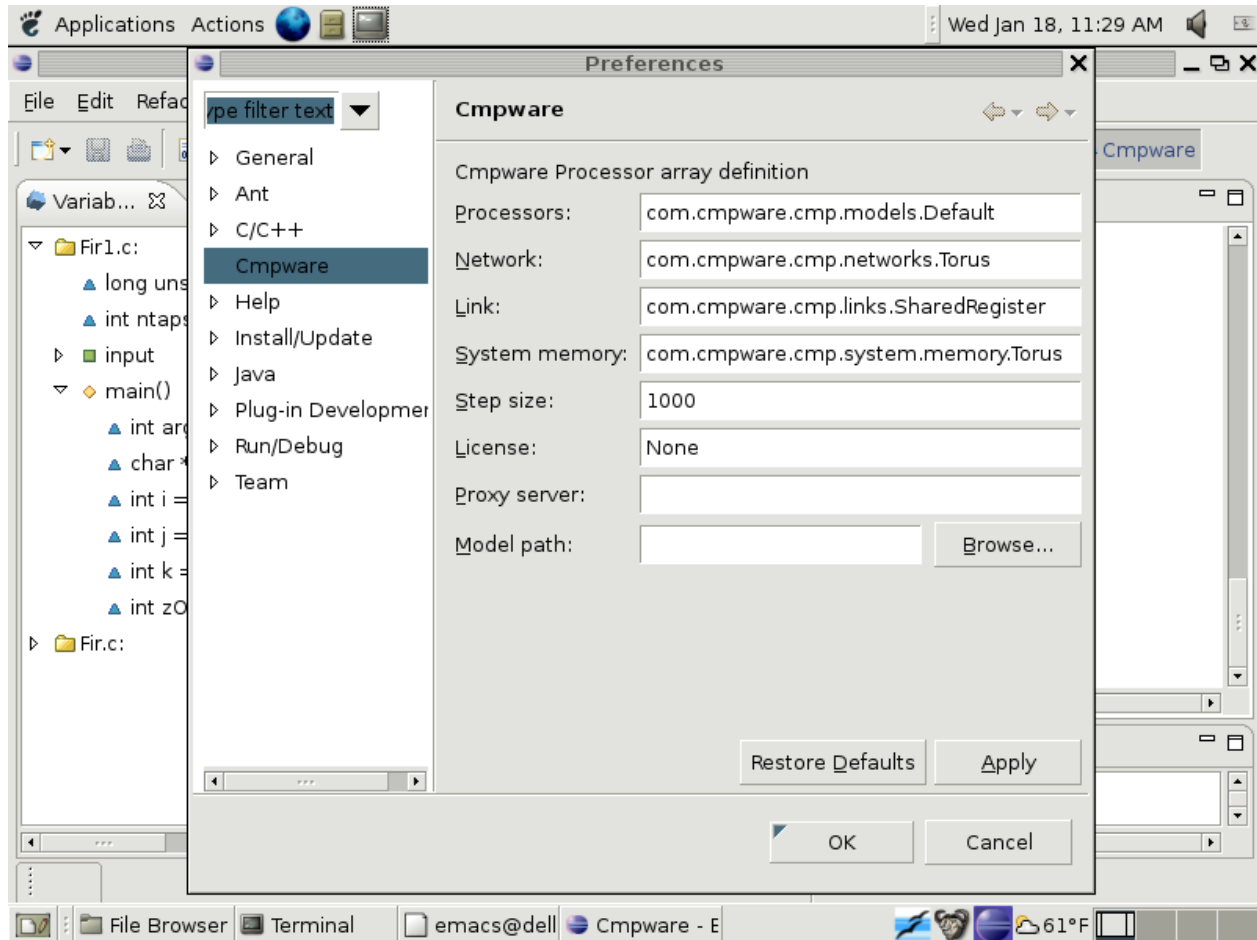




Figure 6: The Cmpware CMP-DK Preferences page.

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 00003000 as in Figure 7. This is the address in the source code where the results will be placed. Stepping the simulation, four byte integer values identical to those plotted from the self hosted version can be seen being written to the memory.

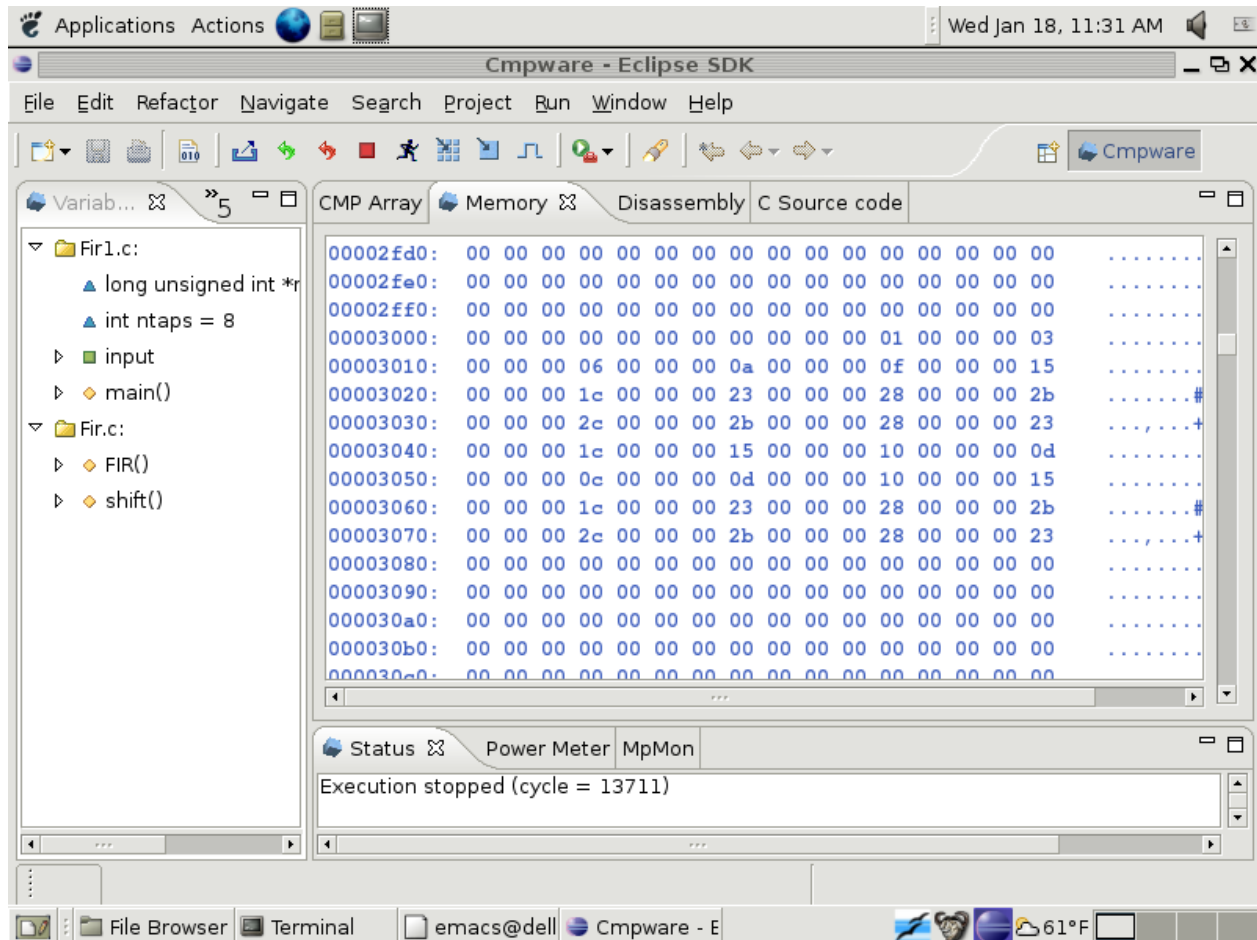


Figure 7: The FIR results in local memory.

At this point, the execution can be single stepped until new values fail to appear in the memory display. More practically, a breakpoint should 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 00003000 in the **Memory** display window, it is clear that the FIR filter is indeed functioning correctly on the *MIPS32* model and generating the correct results.

## Parallelizing the FIR 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 *Fir.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 *FirN.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.

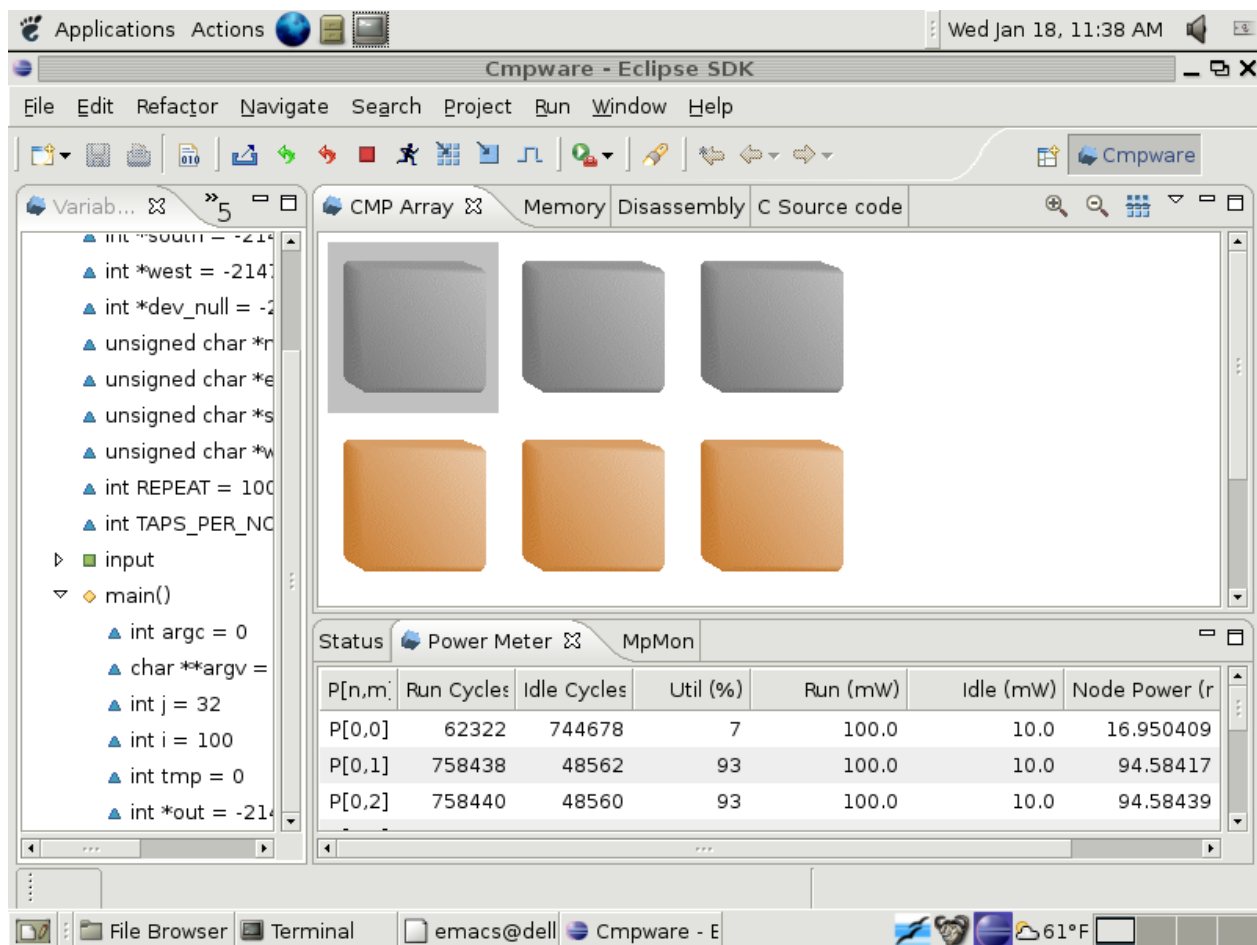



Figure 8: The two node FIR filter completes execution.



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 *Fir0.elf*. The next two processors, (0,1) and (0,2) are loaded with the *FirN.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.

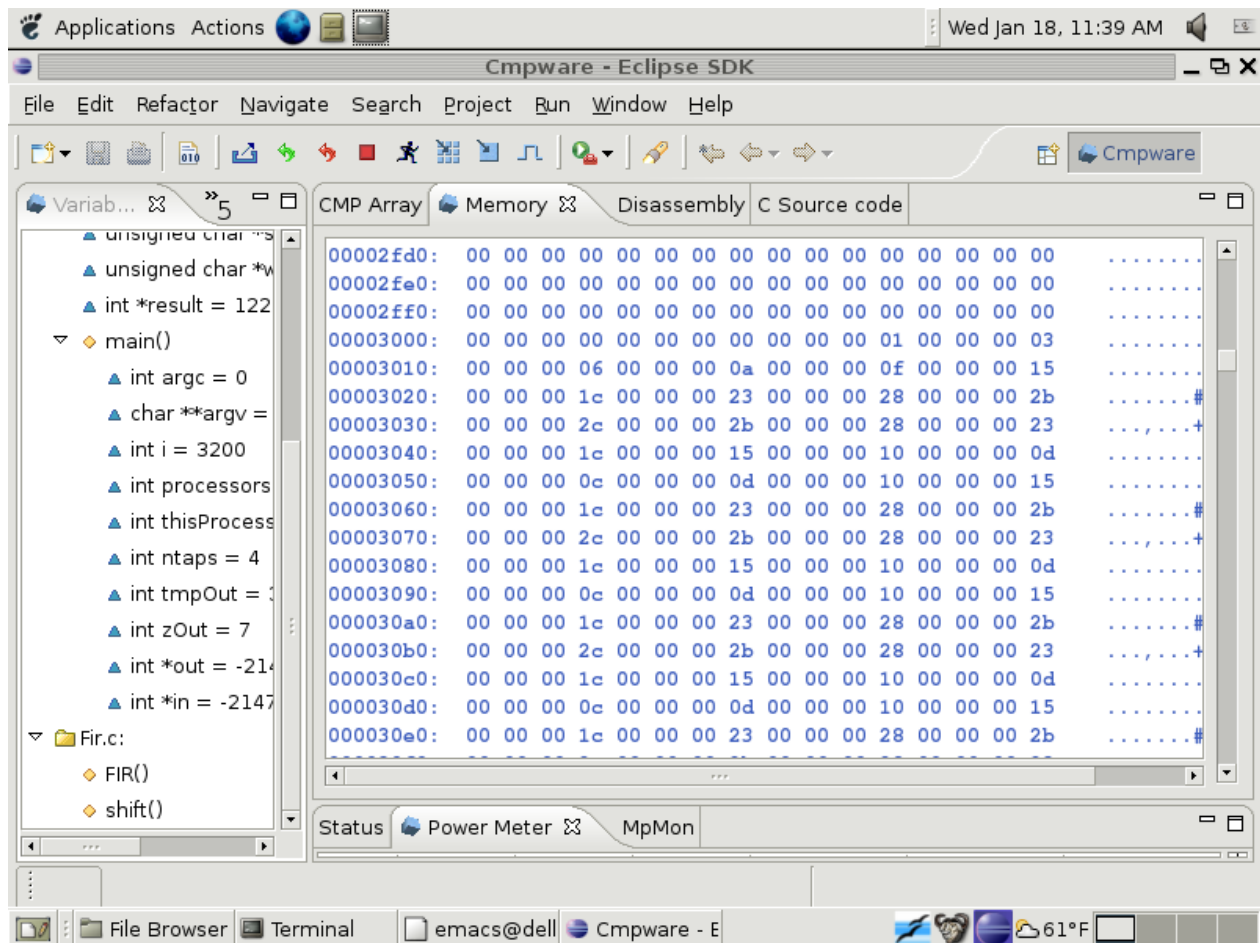


Figure 9: The two node FIR filer results.

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 FIR 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.

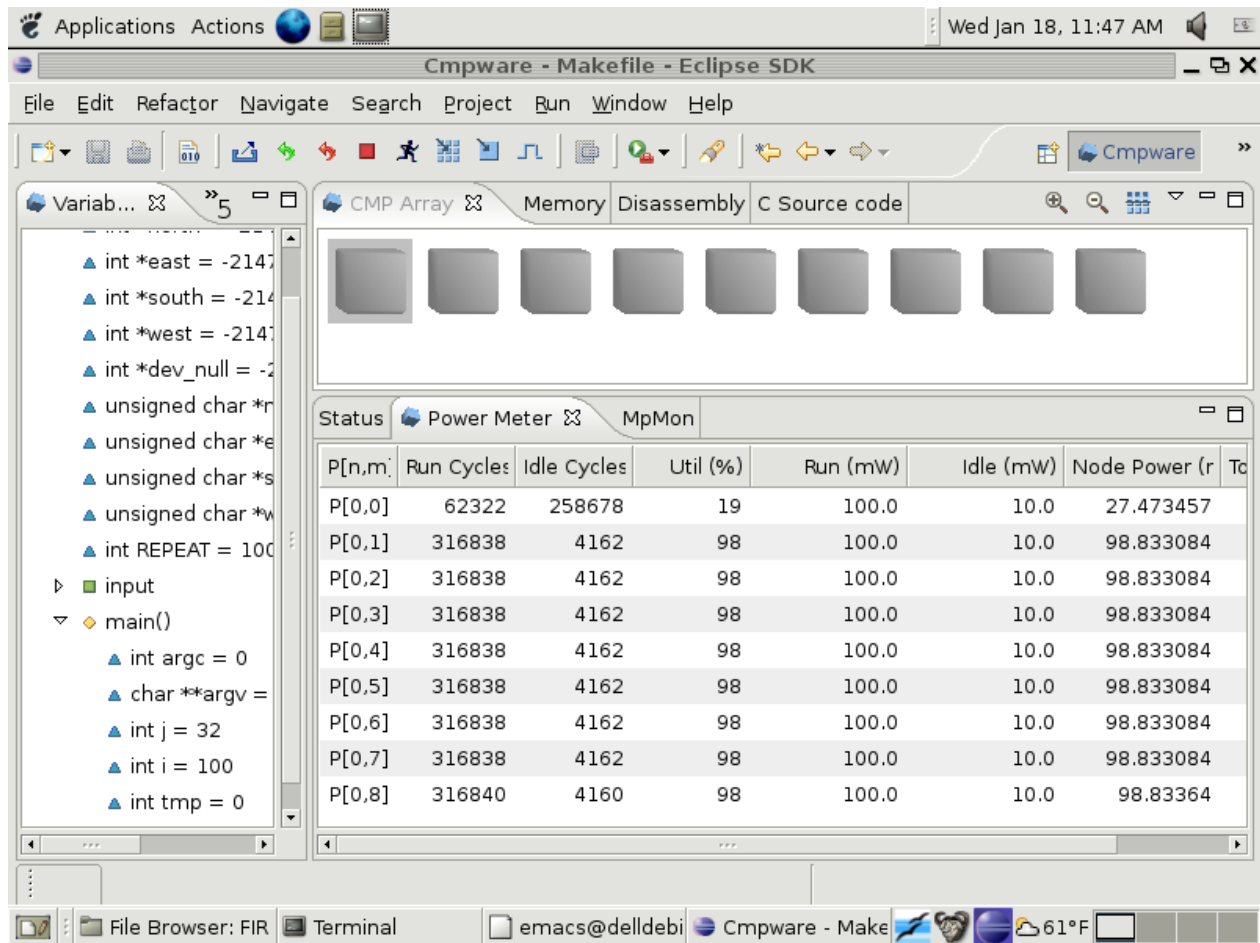


Figure 10: The eight node FIR filter completes execution.

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. Also of interest is the **Power Meter** view. It indicates that the FIR filter executed nearly twice as fast on two processors as on one. Figure 8 shows that the two processing nodes remained



busy 93% of the time. The first processor at (0,0) is busy only 7% of the time because it is simply sending test data to the first processing node, and doing no processing itself.

Figure 10 shows an eight processor version of the FIR filter. 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 eight processors, the utilization remains high at 98% and a large amount of processing in parallel occurs. 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 FIR Application

The code used to perform the basic computation in the FIR filter is exactly the same in the uniprocessor and multiprocessor implementations. This code can be found in the *Fir.c* file and in Appendix B at the end of this document. This code is basic, serial version of the functions `FIR()` and `shift()` as they may be found in a textbook. In the uniprocessor version of the code, *Fir1.c*, these two functions are used inside of the main loop to perform the FIR function. Similarly, these same functions are used inside of the main loop of the multiprocessor FIR code in the *FirN.c* source code file. However, the multiprocessor code in *FirN.c* contains some additional functionality.

```
/* Get the parameters */
processors = *in;          // The number of processing nodes
thisProcessor = *in;      // This processor number
ntaps = *in;              // The number of taps per processor

/* 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 = ntaps;             // Number of taps per node
```

Figure 11: The parameterization code.

Figure 11 shows three 'parameters' being read in to the processor from the input port `*in` and then being modified and sent to the output port `*out`. What this does is establish three global parameters used to describe the FIR filter. 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 `ntaps`, the number of taps per processor used in the filter. These parameters are read in and sent to the next node, with the `thisProcessor` parameter being incremented.

In this case, the number of taps per node is fixed, and there is little real use for the `processors` and `thisProcessor` parameters, although they are useful for debug. It is also possible to further increase the flexibility by making the number of taps dependent on the number of processors. This could dynamically vary the number of taps calculated per processor.

This 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 *FirN.elf* executable from the *FirN.c* source code file can be used to build an FIR filter of any size, given the current requirement of exactly two taps per processor. This not only help reuse 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.

Finally, there is a single line of code between the parameters being read and written. This checks to see if the current processor is indeed the last processor in the multiprocessor operation. 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 `out` 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 in-then structures to take into account the final node in the sequence. Also note that it is more likely that this filter 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 FIR filter.





## 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](mailto:info@cmpware.com)*



## Appendix A: FIR1.c Source Code

```
/*
** This implements the single processor version of the
** FIR filter. 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 FIR
** and is used primarily to test the algorithm.
**
** Copyright (c) 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

#ifndef 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

/* The number of data inputs */
#define INPUTS 32

/* The function prototypes */
int FIR(int ntaps, int inSum);
int shift(int ntaps, int inSum);

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

/* Number of filter taps */
const int ntaps = 8;

/* A simple sawtooth input */
static int input[INPUTS] =
    {0, 1, 2, 3, 4, 5, 6, 7,
     7, 6, 5, 4, 3, 2, 1, 0,
     0, 1, 2, 3, 4, 5, 6, 7,
     7, 6, 5, 4, 3, 2, 1, 0};

int main(int argc, char *argv[]) {
    int i;
```



```
int j = 0;
int k = 0;
int zOut;

for (i=0;i<INPUTS;i++) {

    /* Perform FIR */
    result[j++] = FIR(ntaps, 0);

    /* Shift the delay line */
    zOut = shift(ntaps, input[k++]);

} /* end for() */

#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<INPUTS; i++)
    printf("%d %d %d # 0x%08x 0x%08x\n", i,
           input[i], result[i], input[i], result[i]);

#endif

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



## Appendix B: FIR.c Source Code

```
/*
**
** This implements the routines used in the FIR filter. Note that
** These routines are all completely serial.
**
** Copyright (c) 2004, 2005, 2006 Cmpware, Inc. All rights reserved.
**
*/

/* The filter coefficients */
static int h[] =
    {1, 1, 1, 1,
     1, 1, 1, 1};

/* The delay line */
static int z[] =
    {0, 0, 0, 0,
     0, 0, 0, 0};

/*
** This method computes the FIR.
**
** @param ntaps The number of taps in the
**             FIR filter.
**
** @param sum The partial sum into the FIR.
**
** @return This method returns the FIR result.
**
*/

int FIR(int ntaps, int sum) {
    int i;

    for (i=0; i<ntaps; i++)
        sum += h[i] * z[i];

    return (sum);
} /* end FIR() */

/*
** This method shifts the delay line z[].
**
** @param ntaps The number of elements in
```



```
**          the delay line z[].
**
** @param zIn  The shifted in value
**
** @return This method returns the value shifted
**          out fo the delay line z[].
**
**/

int shift(int ntaps, int zIn) {
    int i;
    int zOut;

    /* Save the last value (being shifted out) */
    zOut = z[ntaps-1];

    /* Shift the delay line */
    for (i=ntaps-2; i>=0; i--)
        z[i+1] = z[i];

    /* Add in the new shifted in value */
    z[0] = zIn;

    return (zOut);
} /* end shift() */
```



## Appendix C: FIR0.c Source Code

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

#include "CmpwareTorus.h"

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

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

/* The number of data inputs */
#define INPUTS 32

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

/* The number of taps per processor */
// const int TAPS_PER_NODE = 4;

/* A simple sawtooth input */
static int input[INPUTS] =
    {0, 1, 2, 3, 4, 5, 6, 7,
     7, 6, 5, 4, 3, 2, 1, 0,
     0, 1, 2, 3, 4, 5, 6, 7,
     7, 6, 5, 4, 3, 2, 1, 0};

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

    /* Send out parameters */
    *out = PROCESSORS; // Number of processors
    *out = 0; // Node number
    *out = TAPS_PER_NODE; // Taps per node

    /* Send data out */
}
```



```
for (i=0; i<REPEAT; i++)
  for (j=0; j<INPUTS; j++) {
    /* Partial sum */
    *out = 0;
    /* Next input */
    *out = input[j];
  }

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

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



## Appendix D: FIRN.c Source Code

```
/*
**
** This implements an FIR filter which may be run on one
** or more processors, depending on the input parameters.
**
** Copyright (c) 2004 Cmpware, Inc. All rights reserved.
**
*/

#include "CmpwareTorus.h"

/* The function prototypes */
int FIR(int ntaps, int inSum);
int shift(int ntaps, int inSum);

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

int main(int argc, char *argv[]) {
    int i = 0;
    int processors;
    int thisProcessor;
    int ntaps;
    int tmpOut;
    int zOut;
    Port out = east;
    Port in = west;

    /* Get the parameters */
    processors = *in; // The number of processing nodes
    thisProcessor = *in; // This processor number
    ntaps = *in; // The number of taps per processor

    /* 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 = ntaps; // Number of taps per node

    for (;;) {

        /* Perform FIR */

```





```
tmpOut = FIR(ntaps, *in);

/* Shift the delay line */
zOut = shift(ntaps, *in);

/* Save tmpOut in memory (for debug) */
result[i++] = tmpOut;

/* Send outputs to next node */
*out = tmpOut; // Partial FIR sum
*out = zOut;   // Shifted out value

} /* 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_ */
```

