University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

A Software Development Kit for CeDIMES

Juan B. del Cuvillo
Robert Klosiewicz
Yingping Zhang

CAPSL Technical Note 10 – Revised
March 15, 2005

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

CeDIMES is an FPGA-based multiprocessor emulation system under development at CAPSL [9, 10]. Its first goal is to accomplish the hardware emulation of a single-chip Embedded Cyclops system. Such an emulator would be the first hardware implementation of the BlueGene/C cellular architecture proposed by IBM [2]. Concurrently with the hardware implementation, system software (compiler, assembler, linker, RTS, loader, debugger, etc.) is being developed so that end users can write, compile and execute applications that take advantage of the new platform. To overlap as much as possible hardware and system software development, a functional simulator, the CeDIMES System Simulator, has been implemented and used for the initial testing of system software tools. A Run-Time System that supports a Pthread-like programming model has been developed and tested with several multithreaded applications such as Conway's Game of Life [6] and a more sophisticated Mandelbrot-set generator [8].

Contents

List of Figures

List of Tables

1 Introduction

In this technical note, we present a work-in-progress report on the system software support for a cellular multiprocessor-on-a-chip architecture. This work is part of a more comprehensive research program focused on programming methods and software infrastructure for cellular architectures actively pursued at CAPSL [5]. Our next discussion will focus on the realization of an IBM Blue Gene cellular architecture known as Embedded Cyclops or CyclopsE [2, 1]. More specifically, we will refer to it as CeDIMES, the first hardware implementation of CyclopsE, materialized by means of the DIMES platform (Delaware Iterative Multiprocessor Emulation System) — a general purpose FPGA-based emulation system under development at CAPSL.

Blue Gene/Cyclops is an architecture that follows the cellular concept proposed by IBM [2]. Each chip contains a number of processors and each processor consists of multiple thread units and a single floating point unit. Each processor is a simple (in-order issue) 32-bit processor operating at a moderate clock rate. The memory hierarchy of the Blue Gene architecture is deep, having registers, caches, on-chip and off-chip memories. More specifically, within a chip, there are multiple memory banks and all the processors in a chip have access to them. The memory banks together form an on-chip shared memory system for the processors. Lastly, there is an off-chip memory associated with each chip. The off-chip memory of all chips form a distributed memory/address space. The program execution model supported is shared memory programming within a chip and message passing across chips.

The document is organized as follows. First we give an overview of the CeDIMES emulator, then proceed to describe a functional simulator of such a cellular platform. As part of the system software, a compiler suite targeted for Embedded Cyclops and a run-time system produced for this system are explained in sections 4 and 5 respectively. Since the user console will be at a host machine, specific software components such as a boot loader, section 7, and a host-RTS, section 8, are presented as well.

2 CeDIMES Emulator

This section provides a brief introduction to the Embedded Cyclops emulator system currently under development at CAPSL. More precisely, it presents the main features implemented by such an emulator. Should the reader be interested in further details, these can be found in the DIMES and DIMES/P documentation [9, 10].

CeDIMES is an FPGA-based board, which is plugged into the PCI bus of a host machine, that emulates a single-chip Embedded Cyclops system. However, the size of the FPGA used for this hardware implementation imposes several constraints on the Cyclops system that can be realized. For instance:

- It has limited hardware: eight processors with four thread units each.

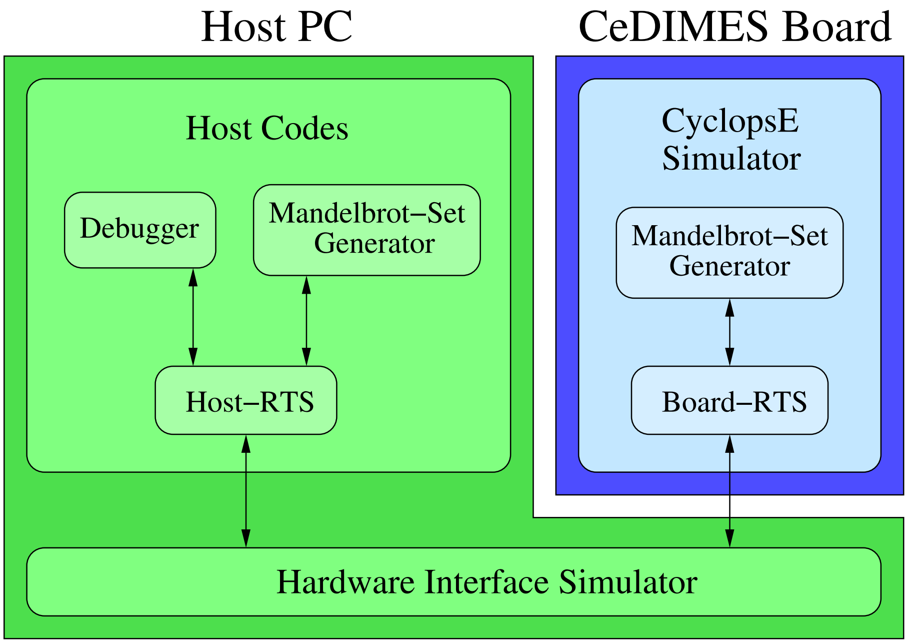

Figure 1: CeDIMES System Simulator Block Diagram

- It has limited memory: 16kB scratch pad (i.e. local per processor data memory) and 512kB global memory.

Moreover, it has limited support for the CyclopsE instruction set. For instance, SYNC and ICI instructions are not supported, and SLEEP can not be executed after a branch instruction.

Memory and instruction set support limitations are not due to the iterative emulator approach but to resource constraints and hardware bugs in the design, respectively. In fact, the hardware emulator has proven to be the useful verification platform it was intended to be. First, initial testing performed on the emulator revealed a few bugs in the chip design. Some have been fixed by the chip architect, others still remain. Fortunately, the remaining bugs only affect specific parts of the design, e.g. the ICI instruction, so that the programmer can work around these issues quite easily. Secondly, the system software, which was mainly developed on the simulator described in section 3, could be validated on a hardware platform. Hence, increasing the confidence of the software development team with regard to the quality of the software produced.

3 CeDIMES Simulator

A functional simulator of the CeDIMES system has been developed. It is implemented as two processes, to run the host machine code and an Embedded Cyclops simulator independently of each other, see Figure 1.

One of the simulator's processes (left box in Figure 1) runs exclusively the host's code, i.e., the software that will run on the host once the hardware CeDIMES platform becomes available. The host-side software consists of: an application that provides the console to its peer process running on the board; a host run-time system intended to provide an interface layer for application developers to communicate with, control and run programs on the CeDIMES board; an object file loader to load and start the execution of programs in the CeDIMES board; and a hardware interface simulator, which behaves like the host-PCI interface available in the CeDIMES emulator. Concurrently, the second process mimics the behavior of the CeDIMES board (right box in Figure 1). This is achieved by means of a functional simulator of an Embedded Cyclops single-chip system, which runs the unmodified user's application linked to the Cyclops run-time system. Such a simulator was provided by IBM, extended at CAPSL with multithreading capabilities, and tailored according to the hardware specification of the CeDIMES emulator.

4 Embedded Cyclops Compiler Suite

We use the GNU C compiler infrastructure version 2.95.2 retargeted to the CyclopsE instruction set. The initial work of retargeting the compiler suite was carried out by IBM, with Cygwin as the development environment. At CAPSL, the compiler tool set was then ported onto the Linux operating system. For further information on how to download and install the compiler suite, refer to appendix A.

5 Cyclops Run-Time System

In this section we discuss the implementation of a run-time system for the CeDIMES cellular architecture. Our RTS consists of a set of libraries that provide the required functionality such as bootstrapping, elapsed time measurement, thread creation and termination, mutex handling, and cross-platform memory allocation. The latter also requires host RTS participation, which is the topic of section 8. However, this section will focus on the RTS that runs on the CeDIMES board.

5.1 Cyclops Thread Library

We start by discussing an implementation of our Program eXecution Model (PXM) on the CeDIMES cellular architecture. The first implementation of our PXM is influenced by the popular Pthread model and uses a direct mapping of software threads into hardware thread units. Upon initialization, a software thread is given control over a well-determined region of the on-chip memory, which is allocated to every physical thread unit at boot time. This enables fast thread creation and reuse. As in the Pthread model, a waiting thread (waiting on an external event/synchronization) goes to sleep; such a thread is woken up by another thread through a hardware interrupt/signal.

Under our PXM, thread execution is non-preemptive. In other words, after a software thread is assigned to a hardware thread unit by the RTS, it will run in the hardware thread unit until completion. Furthermore, the RTS will not swap out a sleeping thread and reassign the idle hardware resources to another software thread.

Since the user is responsible for creating, synchronizing and terminating threads, a minimal set of functionality for writing parallel programs for our cellular platform has been identified and provided to the user as a library. We call it the Cyclops thread or Cthread library.

`cthread_create()` – Run user provided function in next available thread unit. If no thread unit is available, it returns an error condition. One parameter can be passed to the thread function.

`cthread_exit()` – Caller thread terminates its execution.

`cthread_join()` – Caller waits for a thread to terminate.

`cthread_self()` – Returns own thread ID (11bits).

`cthread_sleep()` – Caller thread is suspended for a given number of clock cycles or indefinitely.

`cthread_resume()` – Awake a suspended thread.

The 16kB of local per processor available memory are evenly split among the four thread units, see Figure 2 (left). Thread units have the stack placed in the assigned 4kB region of scratch-pad memory. At the end of the run-time stack area towards which the stack grows, the RTS puts each thread's thread-unit-unique area where General Purpose Registers (GPRs) are saved when an interrupt occurs. The compiler tool set and run-time system software have also agreed that a register (GPR2), which points to a thread-unit-unique area, can be used both to check for stack overflow and to address the thread-unit-unique area, see Figure 2 (center). At the beginning of the thread-unit-unique area, four memory words have been reserved for a special purpose, see Figure 2 (right). The first is used as a lock to guarantee atomic transactions on the remaining three words. The thread unit status, enabled/disabled, and the software thread status, running/asleep, are stored in the second word. Words three and four hold the function address and optional argument specified by the user when a new thread is created. Upon completion the thread overwrites the optional parameter with the exit code.

Besides thread management, the Cyclops thread library provides a mutex object that allows mutual-exclusion via acquiring and releasing locks. The following functionality is also part of the Cthread library:

`cthread_mutex_init()` – Initialize a mutex object.

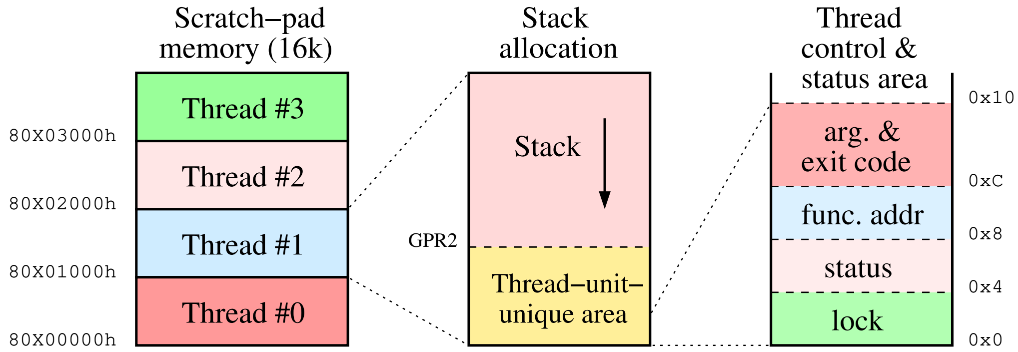

Figure 2: Scratch-pad Memory Organization

`cthread_mutex_destroy()` – Destroy a mutex object.

`cthread_mutex_lock()` – Lock a mutex. If the mutex is already locked, the calling thread will block until the mutex becomes available.

`cthread_mutex_trylock()` – This function is identical to `cthread_mutex_lock` except that if the mutex is currently locked, the call returns immediately.

`cthread_mutex_unlock()` – Unlock a mutex, if it was previously locked by the calling thread.

A mutex has two pieces of state: a "locked" bit and a queue. When the mutex is not locked, the queue is empty. Otherwise, the queue contains 0 or more thread IDs representing asleep threads waiting to acquire the lock. When the mutex is unlocked while the queue is not empty, the first queue entry is removed and its thread awaken, implying it now has the lock. In the current version, the queues are allocated in global memory within the *.data* section. For this reason, there is a limited number of mutexes and the queues' sizes are defined by a constant in the RTS. A queue can hold up to thirty-two thread identifiers, allowing additional thread units to do a busy wait.

5.1.1 CeDIMES Memory Model

This section provides the programmer the basic information regarding the memory model actually supported by the CeDIMES system. It is *not* an exhaustive reference of Cyclops memory model, as the programmer will have explicit knowledge by reading specific documentation on the subject [3, 4, 5].

The facts the programmer needs to be aware of, in order to write data-race free parallel programs are the following:

- The compiler does not take care of the memory model. Furthermore, the compiler is not even aware of any parallel execution constructs, which are explicitly handled by the programmer.

- The hardware design of the CeDIMES network switch is such that global memory follows the Sequential Consistency (SC) memory model [11].
- The scratch-pad memory does not follow the SC model. Therefore, the programmer must use the SYNC instruction. However, the SYNC instruction has not yet been implemented in CeDIMES.
- The stack is placed by the RTS in scratch-pad memory. Since accesses to this memory area cannot be synchronized among threads, mutexes and any other data objects the programmer might implement for similar purposes must be placed in global memory, in other words, cannot be declared as local variables.

5.2 Cyclops Clock Library

Each chip has one Timer register and each thread unit has a Timer Alarm special purpose register. They are both controlled by the RTS to provide two time functions with different degrees of resolution. `cclock_ns()` reads the chip Timer SPR that is counted up at a rate of one count per processor cycle. Assuming a cycle time of 2 ns, the Timer value wraps approximately every 8.59 seconds. `cclock_us()` returns the number of microseconds since the board was reset. It is counted up continuously based on the Timer register, wrapping from the value $(2^{32} - 1)$ to 0 every 70 minutes approximately.

5.3 Cyclops Memory Library

The RTS also implements an atom table that is used by the host to identify data structures within the DIMES-based memory. Because of this property, it may be used to communicate with the host. However, it is not meant for memory allocation in general. For that purpose, the programmer should use `malloc()` and `free()`.

The atom table consists of a number or tag, size and pointer triplets. Each time an object is requested to be allocated by calling `cmem_malloc()` and the tag parameter can not be found in the atom table, the board RTS creates a new entry in the atom table. Such an entry contains the tag and size parameters passed by the function call as well as a pointer to the object allocated in memory.

To access such an object from the host, the program calls the functions `read_atom_mem()` or `write_atom_mem()` to either read or write data into the object. These operations are handled by the host RTS. First, it makes a copy of the atom table managed by the RTS running on the board. Once the memory address of the target object is known, functions provided by the hardware interface are called to read from or write into the object. Obviously this process is carried out by the RTS without user program intervention, hence avoiding the use of magic numbers, nor needing to re-compile user code if the RTS memory allocation scheme is changed.

A point worth noting is that there are two atom tables enabled in the RTS: an application atom table accessible to the user through the `cmem_malloc()` function, and a system atom table only seen by the host/Cyclops RTS. The latter contains atoms such as buffers used by the CeDIMES monitor, and the application atom table itself. Since the host needs to know the location of the system atom table in advance, we have decided to store its address in memory location 0x00. When bootstrapping, the host RTS shall know how to find the system atom table and carry out all required initializations.

The RTS relies on the standard C library functions such as `malloc()` and `free()` to manage the heap. Since a thread's stack is placed in the scratch pad memory, all global memory not used by the application is managed as a heap. Therefore, the heap can grow up to the 512kB global memory upper limit.

5.3.1 Atom Tables

Atom tables are a technique used by operating systems to refer to data allocations that are shareable between processes. The OS maintains a map of "identifier – data". These identifiers have to be unique and somehow known to the process, either by hard-coding the identifier within the program at compile-time, or the process queries some external store for the identifier at runtime. The data field usually refers to a handler that provides access to a system object.

Similarly, the goal of the CeDIMES RTS atom table is to allow the host to identify data structures within the board memory, which in turn, makes communication between board and host possible. Any addresses generated within the DIMES-based part are specific to DIMES, and meaningless on the host. A direct dereference of such a memory-pointer value would be meaningless on the host, just as it would be if pointers were passed between two different processes, because those memory references refer to different memory spaces. The atom table provides the host-based software a way of identifying objects that have a built-in type within DIMES, without having to fix up memory addresses and offsets within the host-based software that references those memory locations within the DIMES board itself.

The atom table can only handle built-in type objects. Therefore, linked lists will not work. For instance, a singly-linked list has a pointer to the next element. That pointer is an address. How should that DIMES-based address be translated by the host? It can't at all. This is the reason for the atom table, which uses identifiers to reference objects. However, the programmer chooses those identifiers at compile-time. Therefore, there is a static number of identifiers within any running code (they cannot be generated at run-time) so dynamic allocation of a linked list that may be used in an identical way by both the host and DIMES is impossible. Only address fix-ups performed via the atom table are meaningful, and those entries within the atom table are allocated at compile-time, not run-time.

In summary, the programmer cannot do dynamic memory allocation *and* directly share that data with the host using the atom table as it is currently implemented.

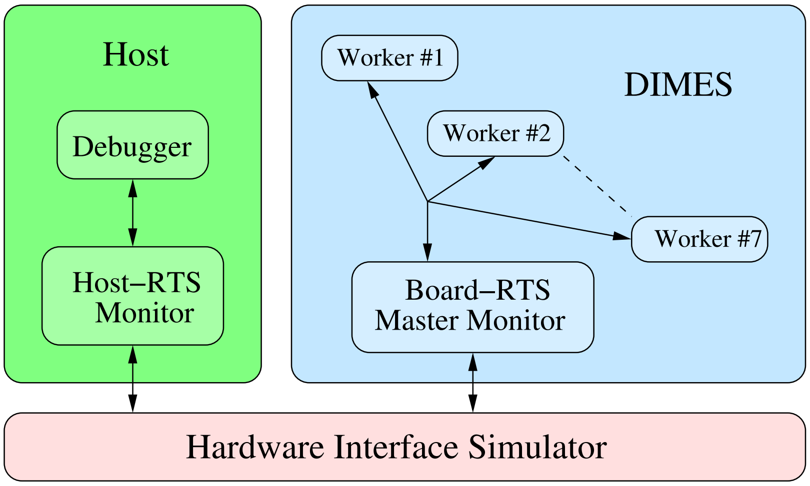

Figure 3: Monitor Software Architecture

5.4 Cyclops Monitor Library

To debug a program in the CeDIMES platform, the user needs access to the debugger, as described in section 9, and to enable a monitor in the CeDIMES RTS. Resource constraints lead us to the decision of having the monitor disabled by default. Hence, it is the user responsibility to enable the monitor and recompile the RTS. Such a procedure is described in detail in appendix C.

Once the monitor is enabled, the RTS hands control of execution over to the monitor program instead of the user's program. The monitor runs exclusively in thread unit 0, while remaining thread units are available for launching user's code, see Figure 3.

A fundamental principle of operation for the debugger is not to lose control of the board. That requires that messages sent from the host must always be replied by the board in order to guarantee the status of the board. Hence, the program being debugged is always well known by the user. Since external interrupts are not implemented in CeDIMES, the only method to ensure such a responsive monitor is to have it running all the time, i.e. exclusively in one thread unit.

Under these conditions, a simple protocol is implemented to establish a communication channel between host and board. According to our protocol, the monitor running in thread 0, master monitor, waits until a message from the host is received. The monitor then checks its correctness. If needed, it relays to the remaining threads where the slave monitor should be temporally running after the user's code has been trapped into supervisor mode (either because the user sent a stop command or because execution reached a breakpoint). Once the requested operation is performed, either by the master or slave monitor, the master monitor replies to the host with the result. Slave monitors only reply directly to the host when reporting events such as thread creation or destruction, or whenever execution is stopped at a breakpoint or by the user.

The first version of the monitor ran the user program, single-stepping instructions whenever a breakpoint was enabled. Because the I-Cache Invalidate instruction (ICI) is not implemented, there is no guarantee the modified program (that with a breakpoint set) will be properly executed, as would happen if the instruction where the breakpoint is set is already loaded into I-cache. As a consequence, the execution of a program in this mode was 100 times slower than without breakpoints. More recently, the monitor has been upgraded with a routine that flushes the I-cache after a breakpoint is set. This piece of code is nothing but a series of branch instructions aligned in such a way that for each instruction an I-cache miss will occur. To ensure the instruction where the breakpoint is set gets evicted from cache, only eight instructions need to be executed; as long as they are 4kB apart. In the end, the code size of the "flush" routine is 32kB (not surprisingly the same size as the I-cache), which is not an issue any more now that the emulator has 512kB of global memory.

6 Hardware Interface

To enable the communication between host and board software components within the simulator, functions to be performed by the hardware host-PCI interface are also provided by the simulator.

The hardware host-PCI interface provides two basic operations: reading from CeDIMES memory into the host machine memory and writing into the board memory. It also provides procedures to reset both the FPGA board and the emulated thread units. All this functionality is also supplied by the simulator hardware interface, to the extent required by the simulator, so that host and board sides can communicate under the simulated environment as if they were running in the hardware platform.

`hardware_reset()` – FPGA reset. Resets all internal registers, initializes I-cache directory and the branch history table.

`dimes_reset()` – Execution reset. Sets the instruction pointer in all thread units to 0x20.

`ll_read_mem()` – Read data from CeDIMES memory into the host machine.

`ll_write_mem()` – Write data from the host machine into CeDIMES memory.

The latest version of the CeDIMES simulator runs the host and Embedded Cyclops codes in separate processes. Copying data from one process' address space into the other can be accomplished by several means. Since the amount of memory that needs to be shared is only 512kB, we use shared memory between processes to implement the hardware interface. As a consequence, the simulation of the host-board interface is possible with simple memory copy functions.

The hardware host-PCI interface imposes a restriction in the use of the read and write memory operations when operating with the actual board: memory locations (both at the host and board) have to be word aligned (32-bit). Byte transfers were possible but at the expense of a much reduced host-board bandwidth. Hence, the option was disregarded for this first implementation.

7 CeDIMES Boot Loader

In order to load and start the execution of a program in the CeDIMES board, a boot loader is also provided. This tool is based on the hardware interface common to hardware (emulator) and software (simulator) implementations of the CeDIMES system. Therefore it works with no modification in both platforms.

The general procedure followed by the boot loader is described below:

- Reset the FPGA board (system reset).
- Load an ELF32 object file onto the board. This file contains the user's code linked with the run-time system that will start running on the board at the end of the reset cycle.
- Initialize the run-time system on the board. For instance, variables that manage the heap are set with the memory address and size of the memory pool used as heap.
- Set the program counter of all thread units to 0x20 and start the execution.

Such a process is mandatory for programs that are to be run in both the host and board sides and make use of the host-RTS [1]. Projects built by the rules 'board' and 'deb' in appendix A include the full boot loader.

For launching programs that are intended to be run only on the CeDIMES board and demand no service of the host-RTS, a stand-alone boot loader is also available. This tool basically loads and starts the execution of an ELF32 object file on the board, and then waits for the oncoming exit code from the board. In appendix A, we explain how to build such a tool.

8 Host Run-Time System

The CeDIMES host side RTS or host-RTS is intended to provide an interface layer for application developers to communicate with, control and run programs on the CeDIMES board. It provides functions for simple loading and execution of programs, as well as supporting more advanced debugging features. It is designed in such a way as to be extensible, so functionality can be added as needed without interfering with already existing features.

[1] Remember that the Cyclops memory library requires of the host-RTS participation.

This section provides a high level overview of the included functions and their functionality. It is *not* a technical reference, as the programmer will have explicit knowledge of the required parameters for each function. These functions serve as a method to hide the low level communication of the host/board RTS.

`load()` – Load a file onto the board.

`run()` – Initial start of the program.

`continue_exec()` – Continues execution from a halted state.

`exit_pid()` – Exit a process.

`read_registers()` – Read either the general or special purpose registers.

`write_registers()` – Write either the general or special purpose registers.

`set_bp()` – Set a breakpoint.

`del_bp()` – Delete a breakpoint.

`single_step()` – Single step.

`query_board()` – Query the board for an event. This is called to query the board for events such as a thread being created, destroyed or stopped at a breakpoint.

`read_atom_mem()` – Determines whether the program running on the board has registered an object in the atom table with a specific tag, and in that case, copies the object contents into the host memory.

`write_atom_mem()` – Similar to the previous function, but writes the contents of the object allocated on the board instead.

The host RTS and CeDIMES RTS communicate using a simple command and reply format, both utilizing the same underlying packet structure. All of the functions initialize packets and send them using calls to lower level read and write functions. Packets are sent to a defined CeDIMES receive buffer. After sending, the host begins querying the CeDIMES transmit buffer, in order to obtain a status reply from the board. After the host receives the reply packet, it clears the CeDIMES transmit buffer. It is then free to issue another command, or in the case of a read/write register, read/write the register buffer. These details are exactly what the host RTS is intended to hide.

9 CeDIMES Debugger

The former section regarding a command line debugger for the Embedded Cyclops has been removed. A new debugger with a graphical interface and more powerful means of controlling the execution of the user program has been developed by one of the authors, and therefore supersedes the old one. Complete documentation about this debugger can be found in Klosiewicz's thesis [7].

References

[1] George Almási et al. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. In *MEmory access DEcoupled Architectures: Proceedings of the 2nd MEDEA Workshop*, pages 10–16, Charlottesville, Virginia, September 22–25, 2002.

[2] Calin Cascaval et al. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 311–321, Boston, Massachusetts, February 02–06, 2002.

[3] Monty Denneau. Embedded Cyclops principles of operation — Part I. Technical report, IBM Watson Research Center, Yorktown Heights, New York, March 2002.

[4] Monty Denneau. Embedded Cyclops principles of operation — Part II. Memory organization and SPRs. Technical report, IBM Watson Research Center, Yorktown Heights, New York, May 2002.

[5] Guang Gao, Juan del Cuvillo, Ziang Hu, Robert Klosiewicz, Clement Leung, Jason McGuiness, Hirofumi Sakane, and Yingping Zhang. Programming method and software infrastructure for cellular architecture. CAPSL Technical Memo 48, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, July 2003.

[6] Martin Gardner. The fantastic combinations of John Conway's new solitaire game life. *Scientific American*, 223:120–123, October 1970.

[7] Robert S. Klosiewicz Jr. A parallel debugger for the cyclops architecture. Master's thesis, University of Delaware, Newark, Delaware, July 2004.

[8] Jason M. McGuiness. A DIMES demonstration application: Mandelbrot-set generation using a work-stealing algorithm. CAPSL Technical Note 11, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, May 2003.

[9] Hirofumi Sakane, Levent Yakay, and Vishal Karna. DIMES/P hardware technical manual. CAPSL technical note, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, September 2003. In preparation.

[10] Hirofumi Sakane, Levent Yakay, Vishal Karna, Clement Leung, and Guang R. Gao. DIMES: A rapid prototyping platform for multiprocessor design. CAPSL technical memo, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, September 2003. In preparation.

[11] Yuan Zhang, Weirong Zhu, Fei Chen, Ziang Hu, and Guang Gao. Sequential consistency revisited: The sufficient conditions and method to reason consistency model of a multiprocessor-on-a-chip architecture. In *the IASTED International Conference on Parallel and Distributed Computing and Networks*, pages 451–457, Innsbruck, Austria, February 15–17, 2005.

A CeDIMES Software Development Kit Installation

This section will guide you through the installation of the CeDIMES Software Development Kit.

A.1 Requirements

The SDK for CeDIMES is only available at the CAPSL CVS repository. In order to checkout the source files, you must have a CAPSL account.

Environment variables may be defined and exported (and included in your ~/.bashrc file preferably) to set the CVS repository where the SDK files are.

To access from a CAPSL local machine:

```
export CVSROOT=/m/europa/cvsroot
```
(1)

When access from a remote machine is required:

```
export CVS_RSH=ssh
export CVSROOT=:ext:username@capsl.udel.edu:/m/europa/cvsroot
```
(2)

where username is the actual user's name.

To install the Software Development Kit, you need a C compiler with the C libraries installed. The SDK has been tested with GCC 2.95.4/glibc 2.2.

A.2 Installation

At this moment, source files, executables and temporary files are kept under a common directory tree. To create all the required directories type:

```
mkdir -p ~/cyclops/src/
mkdir -p ~/cyclops/local/
mkdir -p ~/cyclops/build/
```
(3)

The installation script expects the environment variable `CYCLOPS_HOME` to point at the root of the directory tree. You may want to define and export it (and include it in your ~/.bashrc file preferably) before proceeding with the installation.

```
CYCLOPS_HOME=${HOME}/cyclops/
```
(4)

The next step is to check out the SDK from the CVS server. To do so enter the following commands on the command line:

```
cd ~/cyclops/src/
cvs checkout -r cyclops32_version_2 gcc-2.95.2
cvs checkout -r cyclops32_version_2 binutils-2.11.2
cvs checkout -r cyclops32_version_2_patches newlib-1.10.0
cvs checkout cysim32
cvs checkout -P testsuites
cvs checkout -P dimes
```
(5)

Then you need to compile all these files and install the executables and libraries. Let us start with the compiler tool set. To do so, please use the install script available at file:///usa/jcuvillo/cyclops/public/dimes/doc/install.sh [2]. Before starting with the installation add the `~/cyclops/local/bin/` directory to your `$PATH` variable. CyclopsE executables will be put in this directory.

After the script does its job, the reader is encouraged to test the compiler tool set and CyclopsE simulator, and then to proceed with the installation of the SDK. The procedure to perform this test is described in appendix B.3.

Continuing with the installation, we must first ensure the latest release of the crt0 file as well as the run-time system supporting libraries are installed in the directory where the linker looks for by default. Type:

```
cd ~/cyclops/src/dimes/mon/                                                    (6)
make clean
make crt0
```

Finally, to build CeDIMES specific tools, type:

```
cd ~/cyclops/src/dimes/                                                        (7)
./configure --target=cyclopsE-linux-elf      \
            --prefix=$CYCLOPS_HOME/local     \
            --srcdir=$CYCLOPS_HOME/src/dimes
make clean
make <option>
```

where `<option>` could be any of the following:

run To build the stand-alone boot loader described in section 7. This tool can only be built on a machine that has the CeDIMES board and software libraries installed.

board To build the CeDIMES system boot loader. The user has to modify the `USRS` Makefile variable, which specifies user's object code to be linked to the boot loader. This program can only be built on a machine that has the CeDIMES board and software libraries installed.

sim To build the CeDIMES system simulator.

deb To build CeDIMES debugger and run it in the simulator. However, building this option alone will fail. The debugger is a stand alone application that need to be built separately and then integrated into the CeDIMES simulator (that is what `make deb` does). The details of this process are explained in the Appendix B of Klosiewicz's thesis [7].

Once the installation of the CeDIMES SDK is completed the reader is advised to run a simple set of tests in order to verify that the system is working. Appendix B.4 explains how to accomplish this.

[2]Only accessible within CAPSL main network

B CeDIMES Software Development Kit Usage

B.1 Compiler Tool Set Usage

To compile, use `cyclopsE-linux-elf-gcc`. Do not forget to add `-lc` or `-lm` if the standard C or math library is needed, or `-lcrts` when the run-time system libraries are required, or any necessary include paths.

If the assembler is needed, use `cyclopsE-linux-elf-as` with the `-memb` option to generate code for the Embedded Cyclops.

B.2 CeDIMES SDK Usage

To run any of the CeDIMES specific tools simply call the program followed by the name of the elf32 object file.

cyclopsE-linux-elf-run The stand-alone boot loader can be used to load and start running programs in the CeDIMES board when no RTS support is required on the host-side. Programs such as those contained in our test suite, which require no host-board communication except for loading the object file into the board and returning an exit code at the end of the execution, can be launched with this tool.

dimes-board When the user requires peer processes to be launched in both the board and host, `dimes-board` is the tool to use.

dimes-sim CeDIMES system simulator provides the same features as the `dimes-board` tool. The only difference is that the elf32 object file is run in a functional simulator instead of the real CeDIMES board.

B.3 Testing the Compiler Tool Set and CyclopsE Simulator

As soon as the compiler tool set and CyclopsE simulator are set up, the reader should confirm all tools are working properly. For this purpose there is a test suite consisting of 140 programs (89 C, 2 C++, and 49 Fortran) and a script, `test-sim.sh` that compiles and runs every single program in the simulator. At the end, it prints out a report with the number of successes and compilation and simulation failures. To start with the test just type on the command line:

```
cd ~/cyclops/src/testsuites/                                    (8)
./test-sim.sh
```

At the time of writing, two programs fail: `it_memacc.c` and `20001201.f`. You might get some other failures but this is no reason for concern. A problem with the installation usually causes a whole bunch of programs to fail. Should that be the case, the reader should go back to appendix A and start the process again.

B.4 Testing the CeDIMES SDK Installation

After the latest version of the RTS libraries are installed, and the stand-alone boot loader is built (`make run`), a similar test to that performed above can be carried out. First, we use the simulator as target platform:

```
cd ~/cyclops/src/testsuites/                                        (9)
./test-sim.sh -lcrts
```

Note the extra parameter provided in this case. It tells the script to link the new RTS library (adding a -lcrts switch). Not providing such a parameter will cause compilation failures for all the programs.

The CeDIMES board can also be used to run the test programs. To do so type:

```
cd ~/cyclops/src/testsuites/                                        (10)
./test-board.sh
```

Once both tests are completed, the reader will see similar reports as before. However, the results are not the same. For instance, there are some programs that would have hung up if they were executed on the board. To prevent the second script from stalling, these "bad" programs are skipped, hence the last number reported. Additional failures or problems are also due to the RTS requiring more memory or the unavailability of services such as file I/O. Nonetheless, an occasional run fail should not discourage the reader, since in the end it also validates the installation.

B.4.1 Testing the CeDIMES Run-Time System

The multithreading capabilities of the CeDIMES run-time system are not tested by default. Yet, a set of simple programs aimed to test specific functionality such as thread creation and termination, mutex handling, and elapsed time measurement, is also ready to be used. The procedure to kick off the testing script is similar to before.

```
cd ~/cyclops/src/testsuites/                                        (11)
./test-board.sh rts
```

In this case thirteen programs are compiled and executed on the board, of which, `mutex_long1` and `mutex_long2` take 10 and 8 minutes, respectively to complete. Obviously, no error should occur.

Finally, to verify the correct operation of the atom table a specific CeDIMES tool has to be built, as well as the program to be run on the board. All this can be done by typing:

```
cd ~/cyclops/src/dimes/                                             (12)
make clean
make board
cyclopsE-linux-elf-gcc -o atom_table mon/examples/atom_table.c -lcrts
./dimes-board atom_table
```

Intentionally, we made the source code for these programs easy to read and find (they are all available in `./dimes/mon/examples`), so programmers can learn how to write their own programs with the CeDIMES SDK.

B.5 The Game of Life Demo

The purpose of this section is two-fold. First, it tests the functionality of the CeDIMES SDK. Secondly, it provides a more complete example on how to develop applications for the CeDIMES platform.

However, it is out of the scope of this section to provide details about this particular implementation. Instead, we encourage the programmer to read the code to understand how applications can be written for the CeDIMES platform.

Briefly, the demonstration is a Cthread parallel implementation of the popular Conway's Game of Life simulation game [6]. Game of Life is a mathematical game that resembles a real-life process. It simulates the birth, survival and death of organisms based on a simple set of rules. However, the rules are chosen such that the behavior of the populations is unpredictable. If needed, the reader could find lots of resources entitled to this popular program on the Internet.

The procedure to install and run the Game of Life program is explained below. It assumes the CeDIMES SDK and the Qt3 libraries have been installed.

```
cd ~/cyclops/src/                                                    (13)
cvs checkout gol
cd ~/cyclops/src/gol/
./configure --target=cyclopsE-linux-elf          \
            --prefix=$CYCLOPS_HOME/local         \
            --srcdir=$CYCLOPS_HOME/src/gol       \
            --dimessrcdir=$CYCLOPS_HOME/src/dimes
make clean; make qt; make all
./gol gol-board.bin
```

C Monitor-Enabled Cyclops Run-Time System

To debug applications using CeDIMES SDK, first you have to enable the monitor functionality of the RTS, which is disabled by default, and build the corresponding library.

Once the SDK is installed the procedure to get monitor support is quite simple:

cd into the `./dimes/mon` directory and on the command line type:

```
make clean                                                    (14)
make mon
```

to get the RTS and monitor libraries compiled and installed properly for you. Hereafter, you should append the library flags `-lcrts` and `-lcmon` to all your linker commands when building a program with debug support. Be aware that this procedure has to be undone whenever monitor support is not needed any more.

D Cyclops Thread Library Reference Information

cthread_ret_code cthread_create(cthread_t *thread, void *(*start_routine)(void *), void *arg)

Runs the user provided function in next available thread unit. If no thread unit is available, it returns an error condition. One parameter can be passed to the thread function.

cthread_ret_code cthread_exit(void *status)

Caller thread terminates its execution and the exit status specified by *retval* is made available to any successful join with the terminating thread.

cthread_ret_code cthread_join(cthread_t thread, void **status)

Caller waits for the target thread to terminate. If it returns successfully, the value passed to **cthread_exit()** by the terminating thread will be placed in the location referenced by *thread_return*.

cthread_ret_code cthread_self(void)

Returns own thread ID (11bits).

cthread_ret_code cthread_sleep(const cycles_t cycles)

Caller thread is suspended for a given number of clock cycles, maximum 255, or indefinitely if *cycles* is 0.

cthread_ret_code cthread_resume(const cthread_t thread)

Awakes a suspended thread.

cthread_ret_code cthread_sync(void)

Waits for all accesses to memory, that were initiated by the calling thread, to be completed in memory. All this function does is a SYNC instruction.

cthread_ret_code cthread_mutex_init(cthread_mutex_t *mutex)

Initializes a mutex object.

cthread_ret_code cthread_mutex_destroy(cthread_mutex_t *mutex)

Destroys a mutex object.

cthread_ret_code cthread_mutex_lock(cthread_mutex_t *mutex)

Locks a mutex. If the mutex is already locked, the calling thread will block until the mutex becomes available.

cthread_ret_code cthread_mutex_trylock(cthread_mutex_t *mutex)

This function is identical to **cthread_mutex_lock()** except that if the mutex is currently locked, the call returns immediately.

cthread_ret_code cthread_mutex_unlock(cthread_mutex_t *mutex)

Unlocks a mutex if it was previously locked by the calling thread.

cclock_t cclock_t cclock_n(void)

Returns clock ticks (1 tick = 2ns).

cclock_t cclock_t cclock_u(void)

Returns the number of microseconds since the board was reset.

void *cmem_malloc (const tag_t tag, const size_t size)

Allocates dynamic memory and assigns it the tag *tag*.

void cmem_free (const tag_t tag)

Frees dynamic memory that matches *tag*.