CHALMERS

High Level Description of an ASIC Implementing CPU Support and I/O Control

EDVIN CATOVIC

Computer Science and Engineering Program

DANIEL HEDBERG

Electrical Engineering Program

Master's Thesis

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Göteborg 2002

Abstract

This report details a simulator tool implemented by the authors.

In software projects where the hardware is unavailable, a simulator can be a helpful tool when debugging the developed software.

The simulated hardware is a CPU board with I/O control and CPU support functions for use in space applications.

The simulator tool consists of a cycle-true processor emulator and a high level description of its environment. The high level description is implemented as a loadable module attached to the third-party developed processor emulator.

The report treats aspects on how the requirements on modularity, performance and usability of the high level hardware description were met. The trade-offs made to meet the requirements favour functional behaviour consistency rather than cycle-true timing.

The developed tool has proved to be useful when debugging application software providing better observability of the simulated system than the target hardware.

Contents

1 Introduction

In this chapter we will introduce you to why and how this Master's thesis was carried out. We will also present you with some facts about the company Saab Ericsson Space that issued the project.

1.1 Background

In November 2001 we came in contact with the company Saab Ericsson Space. It is a company that supplies space equipment to companies and organisations such as ESA. They describe themselves as:

Saab Ericsson Space is an independent space equipment supplier. We specialize in digital and microwave technologies and mechanics. The main applications are navigation, telecommunications, observation and launchers.

The company has its headquarters in Gothenburg, Sweden and its Division for Mechanical Systems in Linköping, Sweden. It also has subsidiaries in Austria, Austrian Aerospace in Vienna, and in the USA, Saab Ericsson Space Inc., with offices in Los Angeles, CA.

Saab Ericsson Space presented us a Master's thesis with the aim of developing a simulator for a CPU card with processor support functions and I/O controller units implemented in a high-complex ASIC[1].

The project was carried out during the spring of 2002 at the company headquarters in Gothenburg.

1. Abbreviation for **A**pplication **S**pecific **I**ntegrated **C**ircuit

1.2 Purpose

The software group at Saab Ericsson Space was involved in a project where development of the software and the target hardware run in parallel. The possibility to debug the software in the early stages of the project was limited due to the lack of the target hardware. The project, specially the software development part, would benefit from an alternative debugging environment.

The purpose of the Master thesis was to develop the simulator for the CPU card. The simulator would provide an alternative debugging environment to the real target hardware as well as other debugging features such as better observability than with the real target hardware.

1.3 Method

Saab Ericsson Space already licensed a simulation tool called TSIM developed by the Gaisler Research[1] which simulates the processor and memory mounted on the target hardware.

To develop the tool requested we used the built-in support in TSIM to perform simulation of the communication hardware and its environment. The communication hardware is an ASIC developed at Saab Ericsson Space under the name COCOS[2].

1.4 Scope

Due to the complexity of the communication hardware this Master's thesis would hardly suffice to implement the entire simulator. Therefore the extent of this thesis was left open-ended. There was however expressed a minimum requirement on finding a suitable overall design including logging and stimuli facilities. An implementation of a specific unit for serial communication was also of great interest.

If time remained, additional units were to be implemented.

1. http://www.gaisler.com
2. Abbreviation for **C**omputer **C**ore **S**upport

2 Prerequisites

In this chapter the simulated system is described with its main parts, the ERC32 processor and the COCOS ASIC. The TSIM simulator that is used to simulate the system is also described. Since the COCOS ASIC and the CPU board intellectual property is confidential to Saab Ericsson Space AB the details of this chapter are limited.

2.1 System Overview

The simulated system is a CPU board based on the ERC32 processor and the COCOS ASIC. The block diagram of the board is shown in Figure 2.1.

The main parts of the simulated system are the ERC32 processor and the COCOS ASIC. In the simulated system TSIM provides simulation of the ERC32 processor while the simulation of the COCOS ASIC is done in a loadable I/O module attached to TSIM. RAM and ROM are simulated by TSIM.

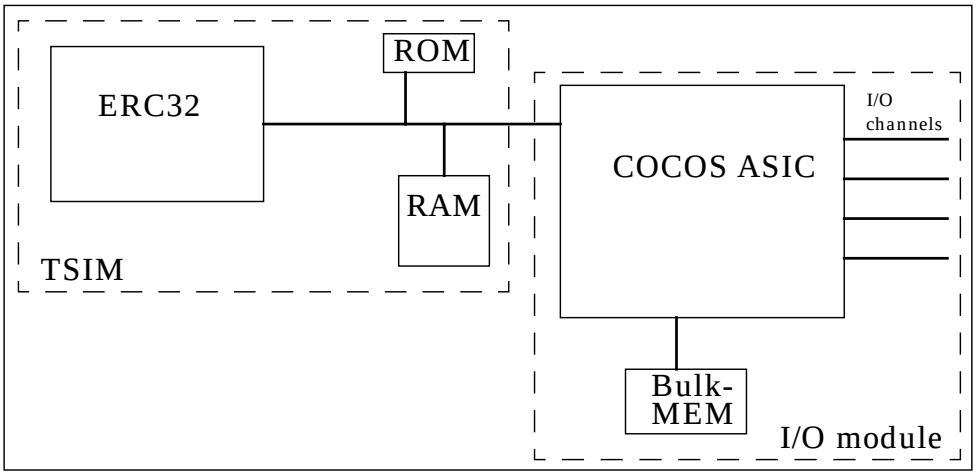

Figure 2.1: System overview

2.2 ERC32

The ERC32 is a 32-bit embedded RISC processor implementing the SPARC V7 architecture specification [2]. It has been developed with the support of ESA (European Space Agency) for space applications using a radiation tolerant CMOS process.

The board is based on the single chip implementation of the ERC32 processor called TSC695E. Memory controller and FPU are included on-chip, as well as peripherals such as: watchdog, timers, DMA arbiter, EDAC and parity generator and checker, interrupt controller and UART making the chip suitable for embedded space applications. The processor runs at clock frequency of 20 MHz.

Since the ERC32 implements the SPARC V7 specification it has characteristics of a RISC processor including simple instruction set, constant instruction length, simple load/store data accesses to memory and pipelined design.

A cross-compilation system, LECCS[1], including GNU tools such as gcc and gdb is available from Gaisler Research as well as the TSIM simulator.

2.3 COCOS Overview

In this Master Thesis project we had access to documents describing the COCOS ASIC and its modules at different levels (User manuals, Functional descriptions and so on) as well as the VHDL[2] code for the circuit. In this subchapter we briefly describe the COCOS ASIC with its modules.

The COCOS ASIC is a general CPU support device and I/O controller that can interface with several processors including ERC32. The modules integrated on the COCOS ASIC provide a number of CPU support functions such as watchdog and interrupt controller as well as interfaces to different bus types such as MIL-STD-1553, PCI and SpaceWire. The COCOS ASIC has an interface (MEM I/F in Figure 2.2) to optional local buffer memory (bulk memory). The block diagram of the COCOS ASIC is shown in Figure 2.2.

The COCOS ASIC allows the processor to set up different transfers on I/O channels connected to the COCOS, such as reading sensor measurements or receiving data from a payload instrument, and let the COCOS handle the actual transfer. The processor can proceed with another tasks while the COCOS complete the transfer.

1. Abbrevation for **L**EON/**E**RC32 **C**ross **C**ompilation **S**ystem
2. Abbreviation for **V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage

All of the COCOS modules registers as well as the bulk memory and PCI address space can be mapped into the ERC32 I/O memory area providing the ERC32 applications with easy access to modules configuration registers and bulk memory.

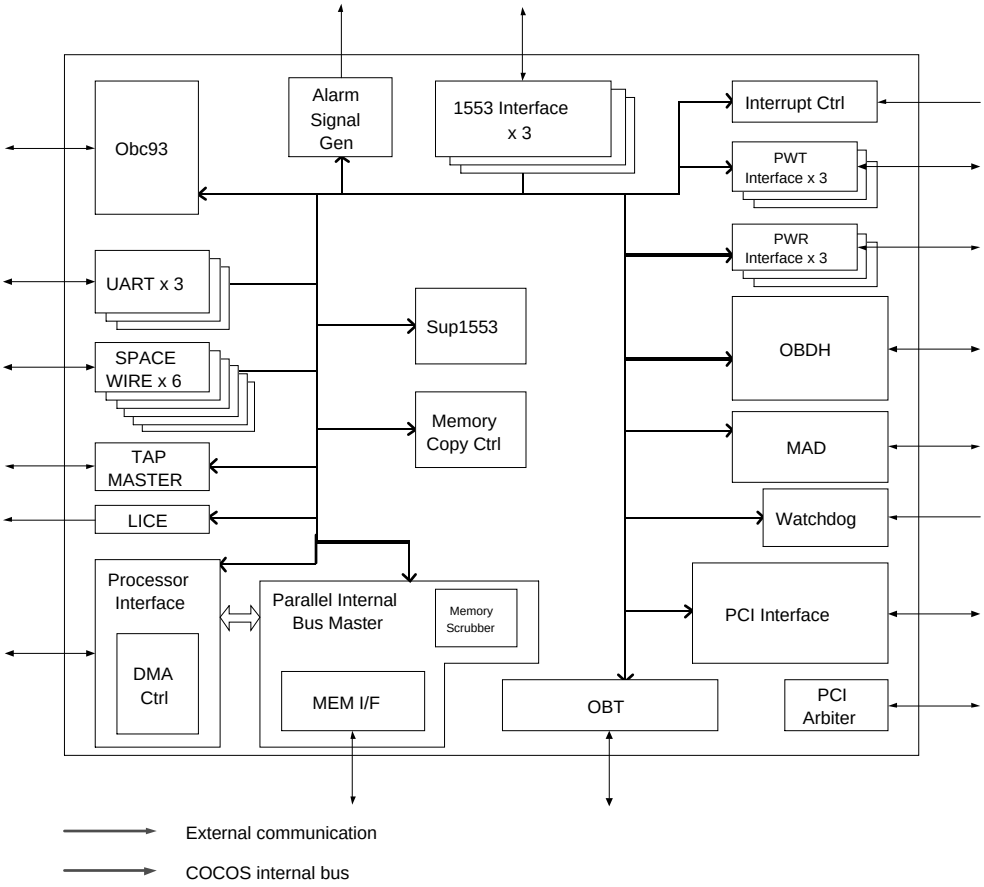

Figure 2.2: COCOS ASIC block diagram

The following modules are integrated on the COCOS ASIC:

- Processor Interface
  The COCOS processor interface handles the communication between the COCOS ASIC and the ERC32 processor.
- Parallel Internal Bus Master
  The Parallel Internal Bus Master handles the internal bus of the COCOS ASIC and has an interface to optional external bulk memory.
- Interrupt Controller
  The Interrupt Controller is used to increase the number of available interrupts to the processor.
- Alarm Signal Generator
  The alarm signal generator generates maskable alarm signals for internal and external errors.

- MIL-STD-1553B Interface
  The MIL-STD-1553B Interface supports MIL-STD-1553B bus standard for serial communication on a multiplexed bus. Three 1553 interfaces are included on the COCOS ASIC.
- Packet Wire Interface (PWR and PWT)
  The Packet Wire Interface supports high speed, one direction point-to-point serial communication. Three Packet Wire receive (PWR) interfaces and three Packet Wire transmit (PWT) interfaces are included on the COCOS ASIC.
- OBDH
  The OBDH module implements an interface to the OBDH (On-Board Data Handling) bus with OBDH bus handler.
- MAD
  No information available.
- Watchdog
  The watchdog supervises the CPU and the software. It has to be refreshed within a given time interval, otherwise it will expire and trigger an alarm.
- PCI Interface
  The PCI interface is able to function either as a PCI bus master or as a PCI bus target. It can also act as system controller on the bus, arbitrating external requests.
- On Board Time (OBT)
  The OBT module provides reference time in a system. The OBT module can synchronize to external reference time or provide master reference time to other units.
- LICE
  The LICE module provides interface to LICE probe used for debugging purposes.
- Tap Master
  Test Access Point (TAP) Master provides interface to devices with Test Access Point according to JTAG standard.
- SpaceWire
  The SpaceWire module is used to send and receive data packets over SpaceWire link.
- UART
  The UART supports the V24 standard. Three UARTs are included on the COCOS ASIC.
- 1553 Support
  The 1553 Support module provides support functions for common 1553 operations.

- Memory Copy Controller
  The Memory Copy Controller supports memory copy operations between different locations in the memory and between different memory banks accessible by the COCOS.

All COCOS modules internal registers are memory mapped and can be accessed in the ERC32 I/O mapped memory area. Two central modules in the COCOS are the Processor Interface, which handles communication between the ERC32 and the COCOS and the Parallel Internal Bus Master, which handles internal data accesses in the COCOS.

2.3.1 Processor Interface

The Processor interface (CPU_IF) handles the communication between the CPU and the COCOS. The Processor interface handles data accesses from the CPU to the COCOS and performs DMA accesses from the COCOS to the CPU local memory. Interrupt routing from the COCOS modules to the CPU is also handled by the processor interface.

The Processor interface maps the external requests received from the ERC32 to COCOS internal memory map. The mapping function is configurable and the mapping information is contained in the internal registers of Processor interface module.

2.3.2 Parallel Internal Bus Master

The Parallel Internal Bus Master (PIM) arbitrates the internal bus of the COCOS ASIC. All internal read/write request are handled by the PIM.

The internal memory map of the COCOS ASIC is configurable and determined by setting up internal registers of the PIM.

The PIM receives requests from the COCOS modules and arbitrates the modules DMA channels allowing one module at time to access a register or a memory location.

The module also implements interface to optional Bulk memory used as local buffer memory during I/O transfers.

2.3.3 M1553 Modules

The MIL-STD-1553B standard and the M1553 module are described in this section.

MIL-STD-1553B

MIL-STD-1553B is U.S. Department of Defence standard for multiplexed serial data bus communication mainly used in aeroplane and space systems. The communication takes place on a single transmission line using time division multiplexing. Communica-

tion between different units takes place at different moments in time. Optional redundant transmission lines can be used in a 1553 system to make it more fault tolerant. A typical 1553 bus architecture is shown in Figure 2.3.

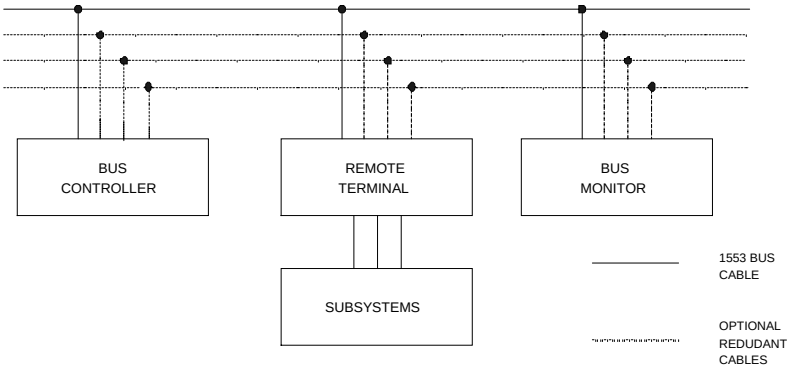

Figure 2.3: A typical 1553 bus architecture

Three types of terminals take part in communication on a 1553 bus:

- *Bus Controller (BC)* initiates all data transfers on 1553 bus by issuing commands to other units on the bus. There is only one Bus Controller on a 1553 bus.
- *Remote Terminal (RT)* responds to commands from the Bus Controller. Remote Terminals are typically devices gathering information from sensors.
- *Bus Monitor (BM)* records the information on the bus and does not actively take part in the communication.

All communication on the 1553 bus is initiated by the Bus Controller that sends a receive or transmit command optionally followed by data words to one or more Remote Terminals. A response from a Remote Terminal consists of a status word followed by optional data words.

In Figure 2.4 a typical 1553 message format is shown. BC sends a transmit command to a RT requesting a number of data words from the RT. As a response to the transmit command issued by the BC, the RT sends a status word followed by the requested data words. In this case the data flow is directed from RT to BC since the actual data is transferred from RT to BC.

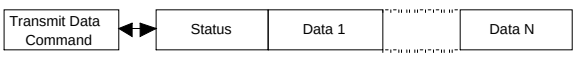

Figure 2.4: A typical 1553 message format

The format of a command word, status word and data word is shown in Figure 2.5. The command word consists of 16 bits divided into four fields:

- *Remote Terminal Address* is used to identify a Remote Terminal to which a command is transmitted. Each RT in a system is assigned a unique RT address.
- *T/R* bit indicates if a command is a transmit or receive command.
- *Subaddress/Mode* field identifies a subsystem or a subfunction in a Remote Terminal. Subaddresses 0 and 31 are reserved for special Mode Code commands.
- *Data Word Count / Mode Code* field contains a number of data words associated with a command or in case of Mode Code command this field contains a Mode Code instructing a RT to perform special operation such as reset or synchronize.

Status word contains the address of the Remote Terminal sending the status word and bits used to convey RT status information to BC, indicate an error in transfer or request service. All responses from a RT begin with a status word.

Data word is 16 bits long and contains the actual information that is transferred from the RTs to the BC. The MIL-STD-1553B standard does not describe the contents of a data word.

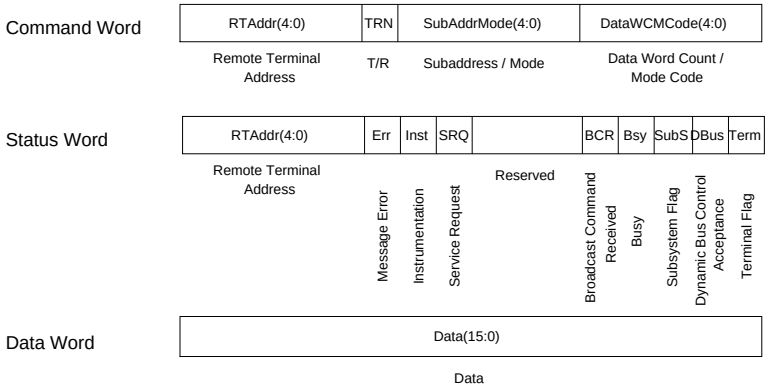

Figure 2.5: Command, Status and Data word

Different message types involving a BC and a single RT are shown in Figure 2.6. A BC to RT message type indicates the transfer where the data flow is from a BC to a RT. The BC issues a receive command followed by data words to a RT. The RT receives the command words and the data words and responses with a status word. RT to BC message type is described previously. Mode Code messages are also shown in Figure 2.6 and can be of either transmit or receive type with optional data word associated with the command or the response.

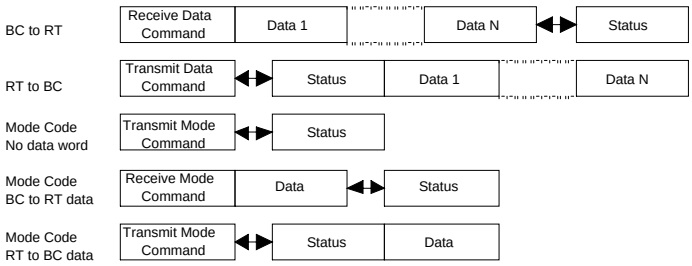

Figure 2.6: Different 1553 message types involving a BC and a single RT

Two Remote terminals can also exchange information. This type of 1553 transfer is initiated by the BC as well as all other data transfers on a 1553 bus. The BC sends two commands, a receive command to the RT that is to receive data and a transmit command to the RT that is to send the data. After the two commands are issued the actual data is transferred. The sending RT sends a status word followed by data words. The receiving RT receives this information and sends a status word as shown in Figure 2.7.

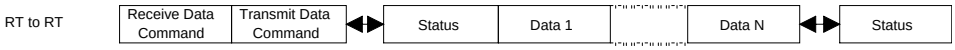

Figure 2.7: RT to RT message format

Broadcast messages are similar to the previously described message type except that all Remote Terminals recieve the command and optional data words. A broadcast message is recognized by Remote Terminal Address field being set to 31. Different types of broadcast messages are shown in Figure 2.8.

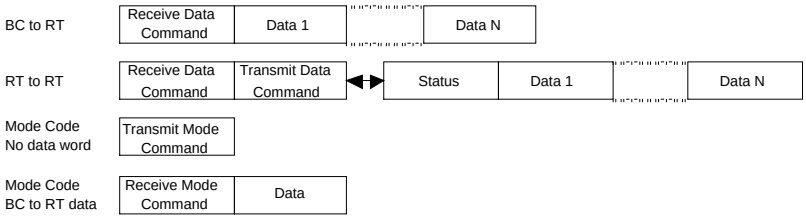

Figure 2.8: Broadcast message formats

M1553 module

The COCOS ASIC implement the module three times to enable communication on multiple M1553 busses. Each M1553 module can be configured to function as either Bus Controller or Remote Terminal, at the same time as they also functions as Bus Monitor.

When configured as Bus Controller the M1553 module is used to schedule the output of commands on the 1553 bus, handle the responses from RTs and handle different types of errors on the 1553 bus. A set of memory mapped registers and special data structures are used to determine the communication pattern on the 1553 bus.

Send List is a data structure used to handle the output of commands and handling of responses from RTs. For each message on the 1553 bus there is an element in a Send List containing information on when the command should be sent, a pointer to a memory location where the command block is stored and a pointer to a memory location where the response block is written. The command block contains 1553 command and optional data words to be sent on the 1553 bus while the response block contains the status word and optional data words received from a RT as well as status information about the message transfer. A Send List is shown in Figure 2.9.

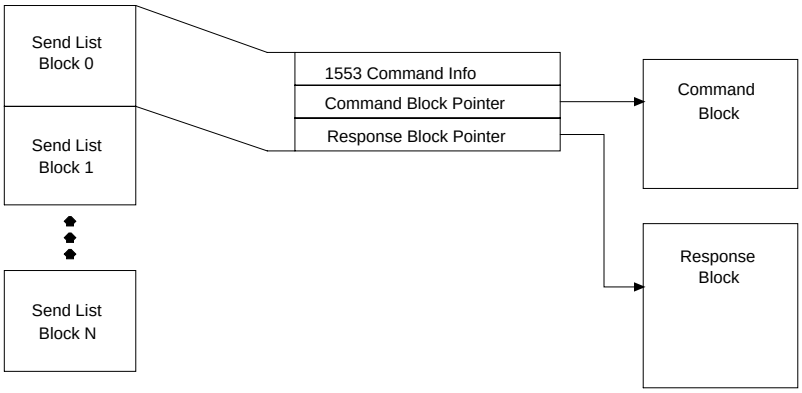

Figure 2.9: Send List

When enabled, the Bus Controller sends commands on the 1553 bus and stores the responses from RTs according to the information in the Send List. No interaction from the processor is required during normal operation. BC can be configured to raise interrupt to the processor on a number of events such as end of Send List, different kinds of errors on 1553 bus and so on.

In this Master Thesis we focused on Bus Controller functionality of the M1553 Module. Functionality of the M1553 module when configured as Remote Terminal or Bus Monitor was not part of our simulation.

2.3.4 Alarm Signal Generator

The Alarm Signal Generator (ASG) generates external signals when other modules get into a state where an alarm shall be asserted/deasserted. When the ASGs interrupt register is unmasked an interrupt is asserted upon alarm assertions.

2.3.5 Watchdog

The purpose of the watchdog (WD) is to supervise the CPU and the software or an external critical application. The watchdog requires a refresh watchdog command periodically within a given time window, otherwise the watchdog will expire. When the watchdog expires an interrupt will be issued, and the Alarm Signal Generator will assert the WDAlarm signal.

The refresh watchdog command can be given by either writing a specific register or by asserting the external WdRefresh signal.

A separate clock, WdClk drives the watchdog, to make sure that it expires even if the system clock is lost.

2.4 TSIM

TSIM is a generic SPARC architecture simulator capable of emulating ERC32 based computer systems. TSIM provides accurate and cycle-true emulation of the ERC32 processor.

2.4.1 TSIM Core

TSIM is capable of simulating ERC32 processor with its on-chip peripherals and a configurable amount of external RAM and/or ROM. TSIM provides a number of functions to simulate ERC32 applications including commands to examine and update processor memory, insert breakpoints and trace execution of an application.

TSIM can also be attached to gdb acting a remote gdb[1] target. Applications are loaded and debugged through gdb or a gdb front-end product such as ddd[2] providing graphical user interface.

2.4.2 TSIM I/O Modules

Loadable modules are used by TSIM to simulate I/O devices. A loadable I/O module is dynamically linked by TSIM when it is started and handles the simulation of the I/O device.

The I/O module contains code and data structures used for the simulation of the I/O device. TSIM does not impose any restrictions on how the I/O module should be implemented except that it shall export a number of functions to simulate the interaction between the processor and the I/O device. These functions define the interface exported by the I/O module and include functions, which TSIM calls to initiate the I/O module or perform an access to the I/O memory area. When TSIM calls these functions the I/O module performs necessary actions to perform the simulation of the I/O device such as updating its internal data structures and its internal state.

1. gdb - GNU project debugger, http://www.gnu.org/software/gdb/
2. ddd - Data Display Debugger, http://www.gnu.org/software/ddd/

The interaction between the I/O module and TSIM primarily takes place when TSIM executes a load or store instruction. In real system the processor requests read/write access to the I/O device on the external bus connecting the processor with external devices. The external devices, on the other hand, can request access to the processor local memory and assert interrupt signals. TSIM provides interface to simulate the interaction between the processor and the I/O device. This interface consists of structures containing pointers to functions exported by either the I/O module or TSIM. Each function simulates an interaction between the processor and the I/O module. Typical interactions are accesses to I/O memory areas, asserting interrupt signals and requesting service from the TSIM simulation engine.

TSIM provides an interface, which the I/O module can use to get services from the TSIM internal simulation engine. The I/O module can get current simulation time or schedule an event to be executed in the future.

Interface to TSIM

The interface between TSIM and the I/O module is shown in Figure 2.10. The interface consists of the following structures:

- `iosystem` consists of functions that simulate the I/O device. The I/O module exports these functions.
- `ioif` consists of functions providing the interface to the simulated processor. This interface allows the simulated I/O device to assert interrupt signals and perform DMA to the processor local memory.
- `simif` provides access to the TSIM internal simulation engine.

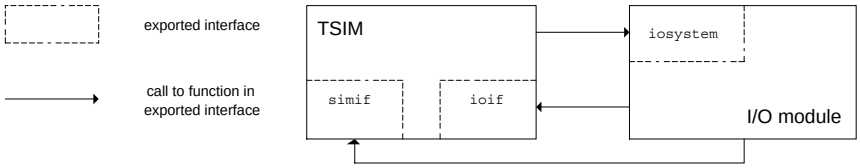

Figure 2.10: Interface between TSIM and the I/O module

TSIM and the I/O module interact through the described interface on events such as:

- Special events, such as simulator start-up, simulator exit, processor reset or when user gives a command to the I/O module at TSIM command line.
- Read/write accesses to ERC32s I/O area. When the processor executes a load/store instruction addressed to the I/O area the control is transferred to the I/O module.
- DMA access from the I/O device to the processor local memory or assertion of interrupt signal to the processor.

- Events in the event queue. The I/O module has access to the TSIM event queue, which can be used by the module to schedule the events in the future by putting them in the event queue. When the event expires the control is transferred to the I/O module allowing it to take care of the event.

The `iosystem` structure

The `iosystem` structure exported by the I/O module is shown in Figure 2.11. It contains pointers to functions called by TSIM on simulator start-up, exit and reset (`io_init()`, `io_exit()` and `io_reset()`) allowing the I/O module to perform appropriate actions, such as initializing internal data structures, on these events.

The functions `io_read()` and `io_write()` are called when the processor performs an access to the I/O mapped memory area. The `addr` parameter contains the address for the I/O access and in case of read access (`io_read()`) the data is supplied in `*data`. In case of write access (`io_write()`) the data is returned in `*data`. Number of waitstates for the access is returned in `*ws`.

The function `get_io_ptr()` is used by TSIM to get the direct access to the simulated memory in the I/O module.

The function `command()` decodes the command given on the TSIM command line but not recognized by TSIM. This feature gives user the possibility to implement additional commands.

The `SIGIO` signal can be handled by the I/O module using the `sigio()` function (not used in our simulation).

The functions `save()` and `restore()` are called when the save or restore commands are given to TSIM giving the module possibility to save or restore its internal state and data structures. In this way a complete simulation state of both TSIM and the I/O module can be saved and restored.

```
struck io_subsystem {
void   (*io_init)();             /* called once on start-up */
void   (*io_exit)();             /* called once on exit */
void   (*io_reset)();            /* called on processor reset */
void   (*io_restart)();          /* called on simulator restart */
int    (*io_read)(unsigned int addr, int *data, int *ws);
                                 /* called on read access in I/O area */
int    (*io_write)(unsigned int addr, int *data, int *ws, int size);
                                 /* called on write access in I/O area */
char   *(*get_io_ptr)(unsigned int addr, int size);
                                 /* direct access to sim. memory */
int    (*command)(char *cmd);    /* I/O specific commands */
void   (*sigio)();               /* called when SIGIO occurs */
void   (*save)(char *fname);     /* save state */
void   (*restore)(char *fname); /* restore state */
};
struct io_subsystem *iosystem;
```

Figure 2.11: Structure provided by the simulated I/O device

The `ioif` structure

The `ioif` structure exported by TSIM is shown in Figure 2.12. It contains pointers to functions used to assert interrupt signals (`set_irq()`) and to perform direct read or write access to processor local memory (`dma_read()` and `dma_write()`).

```
struct io_interface {
void   (*set_irq)(uint32 irq, uint32 level);
                                  /* generate external interrupt */
int    (*dma_read)(uint32 addr, uint32 *data, uint32 num);
                                  /* read access to CPU local memory */
int    (*dma_write)(uint32 addr, uint32 *data, uint32 num);
                                  /* write access to CPU local memory */
};
extern struct io_interface ioif;
```

Figure 2.12: Structure provided by TSIM

The `simif` structure

The `simif` structure exported by TSIM providing the access to the TSIM simulation engine is shown in Figure 2.13.

TSIM startup options can be accessed in `sim_options`.

The function `simtime()` gives current simulation time.

The function `event()` inserts a function (`cfunc()`) to be executed at current simulation time + `offset` in the TSIM internal event queue. In this way the future events are scheduled.

Processor interrupt level can be monitored using `irl`.

A function (event) in the event queue is removed from the queue by calling `stop_event()`.

Calling `sys_reset()` performs system reset.

The function `sim_stop()` stops the simulation.

```
struct sim_interface {
struct sim_options *options;     /* TSIM command-line options */
uint64 (*simtime)();             /* current simulator time */
void   (*event)(void (*cfunc)(), uint32 arg, uint64 offset);
                                 /* insert an event in the event queue */
void   (*stop_event)(void (*cfunc)());
                                 /* remove event from the event queue */
uint32 *irl;                     /* interrupt request level */
void   (*sys_reset)();           /* reset processor */
void   (*sim_stop)();            /* stop simulation */
};
extern struct sim_interface simif;
```

Figure 2.13: Structure providing the access to the TSIM simulation engine

The interface described above defines the interaction between the I/O module and TSIM. It imposes however no restrictions on other interfaces that can be implemented in the I/O module such as I/O module specific user interface that does not utilize the provided interface for optional command decoding through the TSIM user in-

terface (`command()` function in `iosystem` interface). The module can use system calls to the operating system on the host platform to request services such as file handling or opening a terminal window for input/output from the user.

3 Requirements

These are the requirements we concluded to base our design on. They are in line with our detailed studies and customer feedback.

3.1 User feed-back

As software developers we had a golden starting point. Our future users of the application were situated at the same office. This was something we definitely had use for when concluding the requirements.

When a draft for the design was completed in the beginning of March after approximately five weeks work, we held a presentation for the user group. During this meeting we discussed the future implementation. We had earlier sent out a specification to all participants. During this session we got very positive response on the design we had chosen. This made us confident that the draft was worth developing further.

When the implementation had come far enough we also provided the software for evaluation among the employees and got feedback this way during the process.

3.2 Performance

The only speed performance criteria we could conclude from speaking to the users was that the simulation should be fast. We later established that a suitable performance was close-to real time.

Another aspect we had to consider was how detailed our simulation should be. This has much effect on the speed performance. We discovered that normal use of the COCOS would not claim high requirements on the timing. For instance it is not common to create a certain data structure. Then enable it to be transmitted over a se-

rial link to a remote system and as the next instruction make changes to the same structure. Therefore we could decide that timing was not too critical. It was however, necessary to be able to provoke timing errors in some way.

3.3 Modularity

Since our Master's thesis hardly would suffice for constructing the entire simulator, it was very important that the design was modular. This would enable future expansion of the simulators functionality.

One user expressed misgivings about the future performance, as the simulator would expand in complexity. The modularity would, if the prophesy came true, give the option of disabling certain parts of the simulator to enhance performance.

3.4 Usability

During the presentation to the department we had little feedback on this issue. We were given rather free hands under the premise to make it user friendly. Although, the number one priority was to fulfil the scope.

Some ideas came up, though. They wanted logging capabilities of the different external buses and an easy dump command of COCOS registers. They also wanted warnings if their programs did unexpected processing. This could be reading of a write only register, which always returns the well defined value 0x0 regardless of its contents, or putting the COCOS in an undefined state. This would yield valuable information not available when running on the target hardware.

4 Implementation

In this chapter we present the implementation details of the COCOS I/O module.

4.1 Simulator Overview

The I/O module simulating the COCOS ASIC is implemented in C programming language and compiled as a loadable object module. It is attached to the TSIM by dynamically linking the I/O module with the TSIM at the startup of the TSIM.

The user interacts with the simulator (TSIM + I/O module) using the standard TSIM command line user interface and by writing COCOS I/O module specific stimuli files. The output from the simulator is presented in the TSIM command window or written to a log file.

The module interacts with the TSIM through the interface described in [2.4.2] simulating the interaction between the ERC32 processor and an external I/O device.

An overview of the simulator is shown in Figure 4.1.

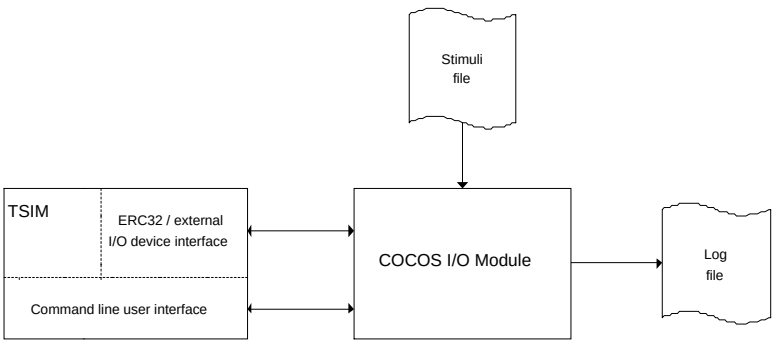

Figure 4.1: Simulator overview

4.2 COCOS I/O Module

4.2.1 Design Choice

The goal of this Master's thesis was to make simulation of the COCOS ASIC meeting the requirements for modularity, performance and usability as described in chapter 3: "Requirements". To meet the modularity requirements in section [3.3], we identified the ways in which COCOS modules interact with each other and defined a suitable software interface as abstraction for this interaction. We found that the appropriate submodule boundaries for the implementation of the I/O module where similar to those of the VHDL design. Each COCOS module is implemented as a separate software module communicating with other modules through an interface simulating the data accesses on the COCOS internal bus.

Since we had concluded that there was no need for a true emulation of the target hardware (see section [3.2]), but rather an adequate simulation we decided for a functional high-level representation. This has many advantages. An idea of cycle-true emulation was therefore abandoned. Not only is it faster to develop a high-level implementation but it also has a greater chance to fulfil the requirement of a close-to-real time performance (see section [3.2]) on the host system.

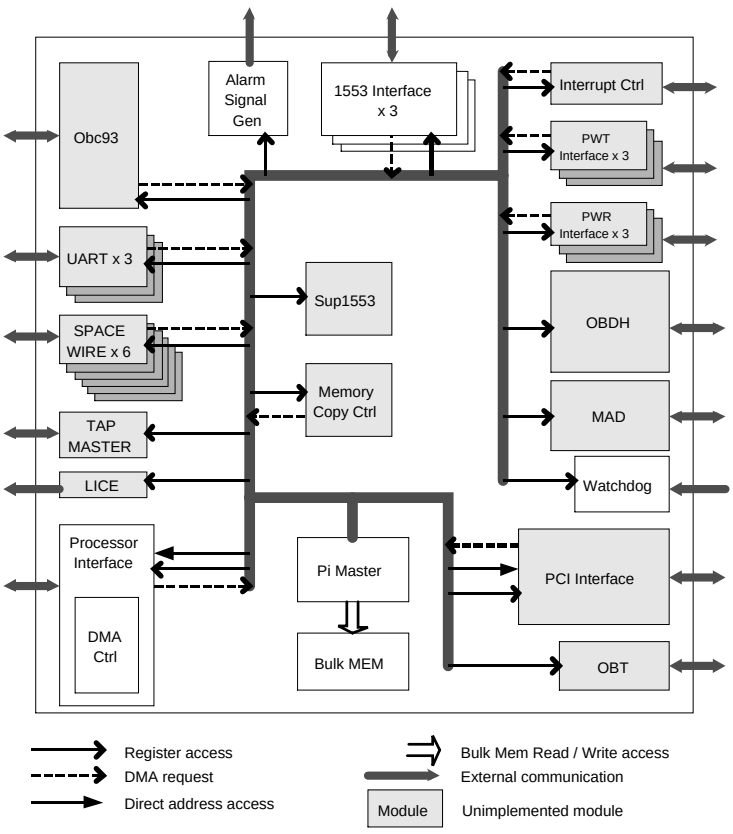

Figure 4.2: Software modules diagram

4.2.2 Modular Structure of the COCOS I/O Module

Two modules, Processor Interface (CPU_IF) and Parallel Internal Master (PIM), play a central role since they are involved in all accesses to the COCOS. Figure 4.2 shows the model we used for communication on the COCOS internal bus. PIM is involved in all internal data accesses and can access a module by performing Register access or direct address access. A module can make a DMA request to PIM to access other modules address space.

Figure 4.3 shows the modular structure of the COCOS I/O module and the interface between the submodules. The interface to the TSIM is also shown. Each module is represented as a box with a list of functions it exports. Calls to functions exported by other modules are marked by arrows pointing from the calling module to the module that exports the function.

External functions are marked by arrows pointing from the calling module to the module that exports the functions.

The core of the COCOS I/O module are CPU_IF and PIM submodules that all submodules interface.

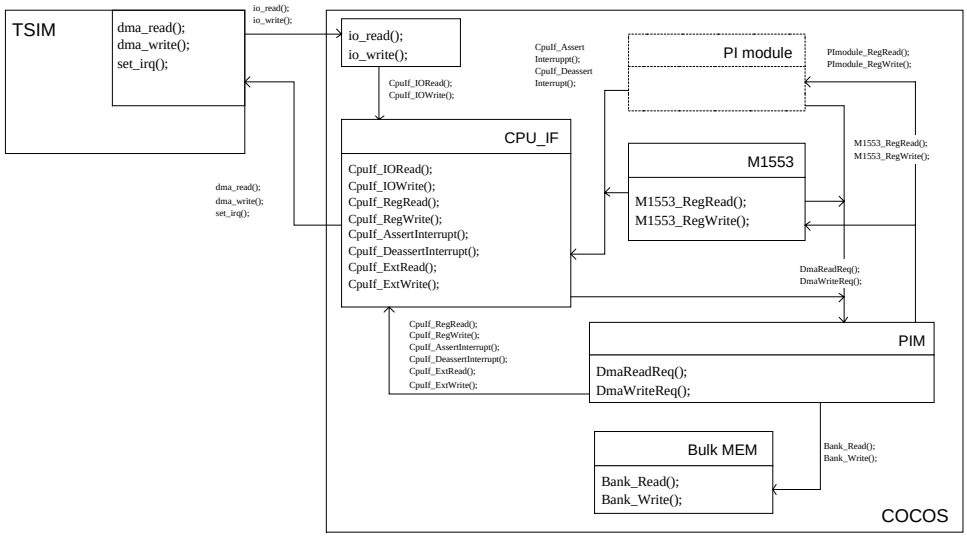

Figure 4.3: Modular structure of the COCOS I/O module

The Processor Interface module handles accesses form the CPU which TSIM simulates by calling functions `io_write()` and `io_read()`. These two functions are exported by the COCOS module as indicated in Figure 4.3. Both functions are called with ERC32 address to be accessed and in case of `io_write()` with data to be written. Return values are the number of waitstates for the access and in case of `io_read()` the data is returned. The Processor Interface calls functions `DmaReadReq()` and `DmaWriteReq()` exported by Parallel Internal Master module to perform the requested access. The Processor Interface also handles the assertion (and deasser-

tion) of interrupt signals to the ERC32 as well as DMA to the processor local memory. The Processor Interface software module exports the functions `CpuIf_AssertInterrupt()`, `CpuIf_DeassertInterrupt()`, `CpuIf_ExtRead()` and `CpuIf_ExtWrite()` for this tasks.

The Parallel Internal Bus Master (PIM) module arbitrates the internal bus of the COCOS ASIC. The PIM software module exports two functions for these tasks, `DmaReadReq()` and `DmaWriteReq()` allowing other COCOS modules to perform internal accesses to other modules or the bulk memory.

Internal registers of the implemented modules are implemented as a part of the corresponding software module. In order to allow the Parallel Internal Bus Master to access these registers all software modules export functions for this purpose (`<module name>_RegRead()` and `<module name>_RegWrite()`). Modules that implement direct address access must also export functions providing access to the modules local address space (`<module name>_ExtRead()` and `<module name>_ExtWrite()`).

4.2.3 Module Specific User Interface

A part of I/O module is an interface used to log messages from the module. Purpose of this interface is to receive messages from different submodules in the I/O module and to present them to the user. Using the I/O module specific command on the TSIM command line the user can decide how the messages are presented, e.g. setting a low debugging level for the I/O module will filter most of the messages coming from the I/O module. The interface exports the following function:

```
void CLog(Module_T Module, CLog_T LogStatus, int LogLevel, char *str, ...);
```

Each message consists of a string and is tagged with module name, logging status (info, warning or error) and logging priority. The tags are used by the function to decide whether the message should be presented to the user or not.

Since TSIM does not impose any restrictions on how the I/O module should be implemented except for the interface between the module and TSIM, the module can make use of OS services to implement module specific user interfaces by directly making system calls to the OS (e.g. the M1553 module reads a stimuli file describing the traffic coming from other 1553 terminals on the simulated 1553 bus).

4.3 COCOS I/O Submodules

In this chapter all implemented COCOS modules are described.

4.3.1 General Implementation of a Submodule

The simulation of the COCOS I/O module and thus the submodules are event-driven. The internal simulation engine of TSIM is used to schedule events.

Initially events are generated by TSIM when it interacts with the I/O module. On an event a submodule takes appropriate action, e.g. updating its own data structures or generating output to the user, or/and generates new events which can be either internal or external. An external event is handled by other submodules while the internal event is handled by the submodule itself.

An external event is typically writing to the other modules internal registers. Since the entire interaction between the modules is simulated through the software interface this kind of events are generated by calling the appropriate function in the external submodule's exported interface. E.g. if TSIM performs a write operation to the M1553 submodule internal register to enable it, a series of events is generated: the first event is generated by calling the `io_write()` function which is part of the I/O modules interface to the TSIM, then `CpuIf_IOWrite()` in CPU_IF is called followed by call to `DmaWriteReq()` in the PIM submodule and finally PIM calls `M1553_RegWrite()` to update M1553 submodules internal register. The register update generates internal events that are handled by the M1553 submodule itself.

In order to meet the modularity requirements and handle externally generated events a submodule has to implement all of the functions that are part of the software interface described in [4.2]. The functions forming the modules interface become entry points to the submodule simulation code as they are entered (executed) each time an external event occurs.

Internally generated events are handled by the module itself and are scheduled using the TSIM internal simulation engine (`event()` function, see section [2.4.2]). Our implementation uses functions representing state machines to schedule internal events. The pseudo code of such function is shown in Figure 4.4.

```
static void Sim()
{
   static State_T state;

   switch (state)
   {
   case Disabled:
      ...                          /* Perform simulation of current state */
      state = Enabled;             /* Next state */
      simif.event(Sim, 0, Delay);/* Put new event in the event queue */
      break;
   case Enabled:
      ...                          /* Perform simulation of current state */
      state = SendData;
      simif.event(Sim, 0, Delay);
      break;
   case ReceiveData:
      ....

   case SendData:
      ...

      ...
}
```

Figure 4.4: Function used to represent a state machine

Communication with other submodules in the I/O module is handled through the interfaces to PIM and the CPU_IF (see Figure 4.3). To request data external data access, in the COCOS local address space or ERC32 local memory) request the submodule makes request to PIM by calling `DmaReadReq()` / `DmaWriteReq()`. An interrupt is assert/deassert by making call to `CpuIf_AssertInterrupt()` / `CpuIf_DeassertInterrupt()`. The structure of a submodule is shown in Figure 4.5.

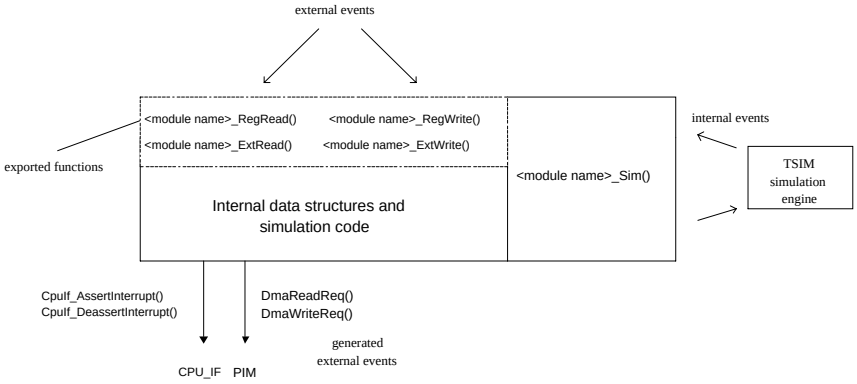

Figure 4.5: Structure of a submodule

4.3.2 COCOS Core Modules

The modules that are defined as parts of the COCOS core are described in this section.

Parallel Internal Bus Master

All data accesses in the COCOS are arbitrated by the PIM. Configuration of the PIM's internal registers determines the internal memory map of the COCOS. When an access is requested PIM maps the request to the appropriate module according to the COCOS internal memory map and takes some of following actions:

- Reads/writes a COCOS modules internal register. This is done by calling modules `<module name>_RegRead()` and `<module name>_RegWrite()` functions.
- Performs a read/write access to the Bulk Memory. The Bulk Memory software module provides functions `Bank_Read()` and `Bank_Write()` for this purpose.
- Performs a read/write access to the ERC32 local memory. The Processor Interface software module provides functions `CpuIf_ExtRead()` and `CpuIf_ExtWrite()` for this purpose.

The arbitration algorithm that the PIM uses to resolve conflicts on the bus is not simulated; instead all modules are given default access time for accessing the bus. This value is the average case number of clock cycle for getting control over the bus. The arbitration time can be changed by the user by giving a command on the TSIM command line and in that way allow the user to simulate the affect of other arbitration times such as worst case arbitration time.

Processor Interface

Main parts of the Processor Interface software module are:

- Memory mapping. The Processor Interface module is called by TSIM from `io_read()` and `io_write()` functions in the `iosystem` structure (see section [2.4.2]) to perform access in ERC32s I/O memory area. The CPU_IF module maps this accesses to accesses in the COCOS local memory map. Information on how ERC32 address space is mapped on the COCOS address space is configurable and contained in CPU_IF internal registers. In the software module this mapping is implemented by matching requests from the ERC32 against information in CPU_IF configuration registers.

- Interrupt forwarding. This module exports the functions `CpuIf_AssertInterrupt()` and `CpuIf_DeassertInterrupt()` which are called by a COCOS module to assert/deassert an internal interrupt signals. A set of registers including interrupt mask register and pending interrupt register are implemented. Internal interrupts are routed to TSIM, if not masked in mask configuration registers, by calling the functions TSIM provides for interrupt handling (see the `ioif` structure in section [2.4.2]).
- DMA to ERC32 local memory. A request for an access in the ERC32s local memory comes from the PIM and is handled by the CPU_IF by calling functions exported by TSIM (see the `ioif` structure in section [2.4.2]).

Bulk Memory

We have decided to include the Bulk memory in the COCOS core. The reason is its own interface to PIM and therefore does not hold the specific properties of the miscellaneous modules presented below.

It may be written to in bytes (8 bits), half words (16 bits), words (32 bits) or double words (64 bits) but is only readable in words. Both reading and writing can though be done through block request to read/write multiple of the above types.

4.3.3 Miscellaneous Modules

These are the modules that are defined as submodules to the COCOS core. These hold similar properties from a design point of view and also communicate with a subset of a given number of interfaces.

Alarm Signal Generator

The Alarm Signal Generator is the less complex module among those we have implemented. The external alarm signals are implemented as prints to <stdout> tagged ASG INFO. An assertion and a deassertion by the WD module look respectively as in Figure 4.6.

```
ASG     INFO External AsgWdAlarm signal asserted
ASG     INFO External AsgWdAlarm signal deasserted
```

Figure 4.6: Alarm Signals

M1553

In M1553 software module the functionality of Bus Controller is simulated. A Bus Controller acts a master on a 1553 bus initiating all data transfers on bus. The Bus Master stores responses from a Remote Terminal and status information on the data transfer (transmission errors, invalid message formats and so on) in the memory.

Implementing the functionality of M1553 BC at function level allowed us to make following simplifications compared to the actual VHDL implementation of the BC module:

- Block transfers from/to memory (bulk memory or CPU local memory). All data block accesses are requested and performed by a single call to the PIM rather than performing several calls to read/write the block. This simplification is possible because of the way in which the BC module is used. The module is set up for data transmission by writing Send List(s) in memory. When the module is enabled the user is not expected to make changes to the Send List(s) until either the module raises an interrupt or the user disables the module.
- Simplified state machine. The implementation of the functional behaviour of the BC module makes it possible to reduce the number of states compared to the actual VHDL implementation. The state machine is implemented as a function described in [4.3.1]. It is activated by writing a M1553 module register and thus enabling the BC. In following states the BC gets the Send List information from the memory, outputs the command, gets the response, handles eventual errors and gets the next Send List. The software module calculates the time to perform tasks in the states and delays itself before going to the next state.
- Bus traffic. The M1553 software module uses a high abstraction level for bus traffic. Since the commands and optional data words to be output on the 1553 bus by the BC are known when the transfer is initiated (a structure containing the command(s) and data words is read as a block) it is output during on clock cycle. In same manner the stimuli, which are parsed at simulator start-up, can be directly analysed and written to memory. The time for a complete message transfer is calculated and the module is delayed for this period of time.

Watchdog

The Watchdog is implemented with all its registers and the WD clock is simulated through the COCOS specific command setclk WD. Alarm assertion on expiration are serviced by the ASG module. A small state machine implements its different states.

4.4 User Interfaces

The simulator is featured with several user interfaces. They will all be presented in this subparagraph. To read about the user interface of TSIM please refer to paragraph [2.4.1].

4.4.1 COCOS Specific TSIM Commands

The TSIM simulator supports the developer of I/O modules to supply commands at the TSIM prompt [2.4.2]. These can be prompted just as if they were a part of the TSIM core. This feature has been taken advantage of when implementing user settings control and commands to retrieve status information.

Table 1: COCOS specific TSIM commands

NAME	DESCRIPTION	EXAMPLE
setclk SYNOPSIS setclk [type] [freq]	setclock sets the type to freq. type can be either M1553Clk or WdClk. freq is (float) MHz:s.	setclk M1553Clk 20 default freq is equal to ERC-32 clock frequency.
cdeb SYNOPSIS cdeb [level]	cdeb sets the COCOS debug level. level can be (int) 0 - 4. 0 = disabled and 4 = detailed	cdeb 3 default debug level is zero.
setstim SYNOPSIS setstim [file]	setstim sets the file where stimuli data is fetched. file must follow the grammar stated in Appendix B.	setstim ../stim/stimfile.stim default is undefined regarding m1553 traffic.
bclogfile SYNOPSIS bclogfile [file]	bclogfile sets the output file where m1553 bus traffic is logged when a m1553 module is acting as busmaster.	bclogfile ../stim/m5bus.out default file is <stdout>
arbitdelay SYNOPSIS arbitdelay [module] [delay]	arbitdelay sets the arbitration delay to use for a module when simulating a DMA request by the module.	arbitdelay M1553A 8 default delay is three (estimated average delay)
page SYNOPSIS page [page no]	page prints a graphical representation of the local address map for page page no. page no can be (int) 0 - 7.	page 0
dumpregs SYNOPSIS dumpregs [module]	dumpregs prints the contents of all registers implemented in module to <stdout>.	dumpregs M1553A

It is recommended that a batch- or .tsimrc file is created for easier execution of these and other standard TSIM commands [1].

The reserved names of different modules to use on the command line are found in Table 2.

Table 2: Reserved module names

Module	Name
Alarm Signal Generator	ASG
Processor Interface	CPUIF
Parallel Internal Bus Master	PIM
M1553A	M1553A
M1553B	M1553B
M1553C	M1553C
Watchdog	WD

4.4.2 M1553 Environment Stimuli

The environment of the M1553 units is simulated by writing a stimuli file. The file must comply with a defined grammar in Appendix B which represents responses (see section [2.3.3]) from RT-units on the simulated M1553 buses. It is also possible to simulate transfer errors. The grammar for simulation of M1553 units in bus controller (BC) mode is also implemented. This supports full simulation of the RT-mode M1553 units when these are implemented.

The stimuli file is parsed upon use of the COCOS specific TSIM command setstim. The following example gives an idea of what a simuli file should look like.

In this example the CPU board is communicating with a measuring device. The communication is over MIL-STD-1553B and the CPU board is acting as BC and the measuring device as RT. The measuring device is sampling 32 bit data at 1kHz frequency. This data is transmitted over M1553B to the CPU board every 5ms as a M1553B message. The following stimuli simulate the RT units responses to the BC transmit commands.

```
...
M1553A

RT_Response:
1 0x0800 0x1fff 0xffa1 0x1fff 0xffa5 0x1fff 0xffaa 0x1fff 0xffb3
0x1fff 0xffa3 OK rt=5
1 0x0800 0x1fff 0xffa5 0x1fff 0xffa5 0x1fff 0xffab 0x1fff
0xffb3 0x1fff 0xff99 OK rt=5
1 0x0800 0x1fff 0xffa1 0x1fff 0xffa5 0x1fff 0xffaa 0x1fff
0xffb3 0x1fff 0xffa3 OK rt=5
1 0x0800 0x1fff 0xffa5 0x1fff 0xffa5 0x1fff 0xffab 0x1fff
0xffb3 0x1fff 0xff99 OK rt=5

SpaceWire5
...
```

- '`M1553A`' indicates to the parser that the following stimuli are to be associated with M1553A commands.
- '`RT_Response:`' indicates to the parser that the following stimuli are for response actions on Transmit, Receive and Mode commands to be read in chronological order.
- '`1`' is a repeat parameter. 1 states repeat once, 2 twice... and 0 forever.
- '`0x0800`' is the status word replied stating that the transmit command was detected, valid and that the RT unit address is 1 as stated in Figure 2.5. The status word is replied to all commands but, non-RT/RT Broadcast responses.
- The following ten 16 bit data words are representing the five 32 bit data words per response.
- '`OK`' is the status of the transmitted Response. Simulates the transfer status detected by the BC unit being OK. Can also be typed in hex M1553 syntax [3]. The equivalence for OK would be 0x0.
- '`rt=5`' simulates the response time (5 bit times) detected by the BC unit. Bit time is the M1553 name for the time period of the transmission frequency.

This example only contains responses to Transmit commands. Responses to Receive commands have the same syntax but never contains data words. (i.e. '`1 0x0800 OK rt=5`')

Stimuli File Parser

The parsing of the stimuli file is done using automatically generated lexical analyser produced with flex tool and parser produced using bison tool. Input to the flex tool is file (stim.lex) defining lexical symbols of a stimuli file while the input to the bison is file (stim.yacc) defining the grammar of a stimuli file. The file (stim_actions.c) contains functions (semantic actions) used by parser to produce the output of the parsing described below (these are typically functions building up data structures when the parser matches a rule defined by the grammar).

The result of the parsing is a structure containing lists with BC stimuli (BC commands) or/and RT stimuli (responses from RTs) for each M1553 module. Each element in a list corresponds to a line in a stimuli file which can either be a BC command or a response from a RT.

Parsing of a stimuli file of similar structure to the stimuli file used in example above would result in a structure shown in Figure 4.7. The stimuli file contains RT responses to M1553A module and does not contain any stimuli for M1553B, M1553C or the BC commands to M1553A unit. RT stimuli (responses from RT units) are put in a list where each element corresponds to the number of times the message is repeated and the response from the RT unit containing status word, optional data words, transfer status and response time information.

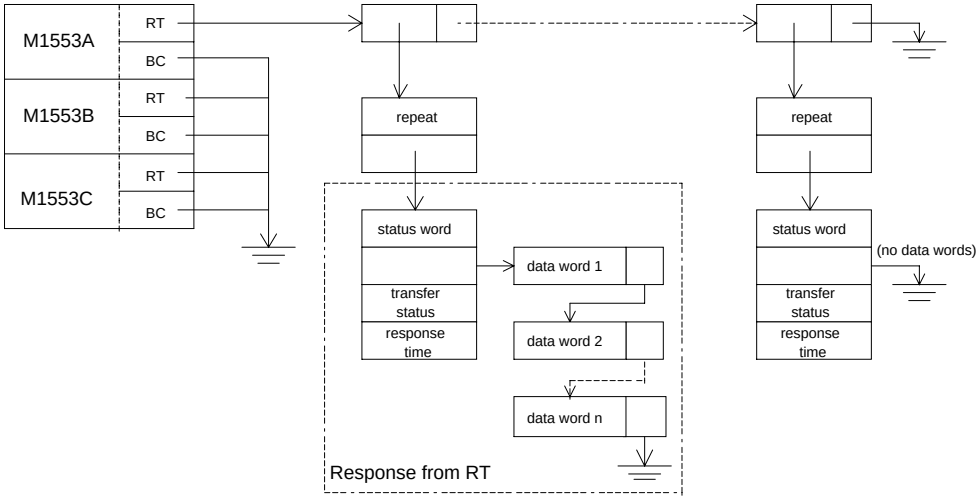

Figure 4.7: Output from the stimuli file parser

A description of the stimuli file syntax can be found in Appendix B.

4.4.3 M1553 Environment Log

A log feature is implemented to present a file containing information about the M1553 bus activity. This file can be analysed by the user after a program has ended or been stopped in the simulator.

A typical log might look something like this

```
...
M1553B:@ 1075234862 bit cycles | 0883 0001 0002 0003 <-> 0800
M1553B:@ 1096811884 bit cycles | 08c8 0011 0012 0013 0014 0015
0016 0017 0018 <-> 0800
M1553A:@ 1675234862 bit cycles | 0883 0001 0002 0003 <-> 0800
...
```

It displays the name of the bus master and the bit cycle in which the first bit of the message was sent. It is followed by the half words that have appeared on the simulated bus in chronological order. A bit cycle is the M1553 name for a specific count of bit times since the start of `main()` program execution.

4.4.4 COCOS Debug Feature

Using the enhanced TSIM command cdeb invokes an extensive logging feature. By setting the cdeb level a selection of these logs are written to <stdout>.

A typical log in debug level three, which is quite detailed, might look something like this.

```
...
INFO M1553A:Sending 1553 message from BC Command block on addr
0x04000100
INFO M1553A:Getting RT response form 1553 bus (stimuli)
INFO M1553A:Writing BC response block on addr=0x04000200
INFO M1553A:Calculating 1553 bit times
INFO M1553A:New 1553 message logged onto bclogfile
INFO M1553A:Getting SL block 1 on address 0x0400000c
INFO M1553A:Sending 1553 message from BC Command block on addr
0x04001100
INFO M1553A:Getting RT response form 1553 bus (stimuli)
INFO M1553A:Writing BC response block on addr=0x04001200
INFO M1553A:Calculating 1553 bit times
INFO M1553A:New 1553 message logged onto bclogfile
INFO M1553A:Getting SL block 2 on address 0x04000018
WARNING M1553A:SL End interrupt masked
...
```

4.4.5 The Local Address Space Visualizer

Setting up the local address space is by most users felt to be the most risky set-up to do in the COCOS. Therefore a local address map visualizer has been implemented to display each of the eight address spaces that can be reconfigured. These are called pages. All registers concerning memory mapping are analysed to render the graphical representation. The default configuration of Page 0 is shown below.

```
tsim> page 0

           COCOS Page 0 Address space
            _____________________________
0x00000000 |                             |
           |  COCOS Register memory      |
           |_____________________________|
0x00010000 |                             |
           |      CPUIf memory           |
           |_____________________________|
0x04000000 |                             |
           |     Bank 0 memory           |
           |_____________________________|
0x08000000 |                             |
           |     Bank 1 memory           |
           |_____________________________|
0x0c000000 |                             |
           |     Bank 2 memory           |
           |_____________________________|
0x0e000000 |                             |
           |     Bank 3 memory           |
0x0fffffff |_____________________________|
           |                             |
           |       Not Used              |
           |_____________________________|
0x20000000 |                             |
           |PCI Direct Access memory     |
0x7fffffff |_____________________________|
           |                             |
           |       Not Used              |
0xffffffff |_____________________________|
tsim>
```

5 Conclusion

In this chapter the project is evaluated. An advised plan for the continuation of the project is also presented.

5.1 Experiences

The project has taught us that our thorough preparations before we started implementing the application were indeed a virtue. This was partially a new experience since the project was of a much greater extent than those of earlier projects at Chalmers[1]. These were of less than one fourth the size. Now we had really looked into the prerequisites and created a plan for the entire project. This we believe has much credit for the success of this project.

We also learnt how to work with conflicting requirements. The requirements forced us into several trade-off decisions. We discussed a large number of designs during the process of work.

5.2 Results

The results, which are a reflection of our processing of the requirements, are presented in correspondence with chapter [3] as performance, modularity, user interface and usability.

5.2.1 Performance

In order to estimate how the COCOS I/O module affects the performance of TSIM we used profiling tools (Solaris built-in profiling possibilities and gprof[2]) on TSIM while running a typical ERC32 application using a M1553 module in the COCOS as a Bus Controller. The analysis showed that approximately 5-10 % of time was

1. Chalmers University of Technology, Gothenburg, Sweden
2. GNU profiler (http://www.gnu.org)

spent executing I/O module code. The execution times for a single function in the I/O module were in general evenly distributed and most of the functions do not need to be optimized to improve the performance significantly. The functions used for logging and presenting debug information to the user had relatively long execution times and are suitable for optimization to get even better performance for the I/O module.

5.2.2 Modularity

A large amount of work was spent designing the appropriate module boundaries and the interface for interaction between the modules. This resulted in the I/O module core with the interface defining the core interaction with other submodules. This allows future users to easily attach additional submodules by following the design rules in [4.3.1] and Appendix A.

5.2.3 User Interface

A number of COCOS specific commands can be given at the TSIM prompt. The commands are described in section [4.4.1] and allow the user to configure the COCOS I/O module, load stimuli files, extract debugging information or dump COCOS internal registers.

An interface for presenting the debugging information to the user is implemented and described in section [4.2.3].

5.2.4 Usability

When implementing the simulator we were faced with a specific design problem. Since the simulation was to be done in a functional level we had to decide what should be tested. We had already concluded in section [3.2] that timing not was critical. During the evaluation period we could conclude further more that our design caught typical errors such as memory access errors, errors while setting up configuration registers and logical errors.

5.2.5 Conclusion

This design has the following advantages.

- Small effect on TSIM performance (performance without I/O modules).
- Easy to expand even far beyond the topical target hardware
- The application can be ported to many hosting platforms (TSIM today supports: Solaris, Linux and Windows)
- Debugging capabilities

And has the following disadvantages.

- Not an emulation
- Does not simulate true cycles

The conclusion is that we have fulfilled the scope and requirements of this Master's thesis.

5.3 Future work

As mentioned in the introduction this Master's thesis was presented open-ended. This proved to be for a good reason. The simulation of hardware of this complexity can always be expanded or refined. Therefore this subparagraph only will present recommendations on what could be done next.

This project included creating a small user interface. This is an area of knowledge one can become an expert on. We can only hope that the user interface we have created will please as many as possible. It has however been made flexible enough to enable all imaginable (graphical) representations. This can be implemented without too much work since the simulator communicates through files, which easily can be processed by additional software.

The first thing to do though would probably be to implement the other M1553 modes i.e. Bus monitor and Remote terminal to make the M1553 modules complete. The following list is based on the up-to-date known customer needs in priority order. The time typed in italics after each bullet is an estimated time of development for one person of equal knowledge of the system as we have. The estimation is calculated from the complexity of the corresponding VHDL files and in consultancy with the ASIC design group.

- M1553 Bus monitor and Receive terminal. *(7 weeks)*
- The PCI interface *(10 weeks)*
- Remaining modules *(14 weeks)*

Refer to chapter [4.3.1] when implementing additional modules where a general implementation of a submodule is presented.

Acknowledgements

First of all we would like to thank our tutors John Alexandersson and Fredrik Hjalmarsson at Saab Ericsson Space for never hesitating to help us with all our technical questions. They have actively followed our progress with interest, which has felt very supportive.

We also thank the division object manager of GEP Software, Annalena Johansson for the same reason and for trusting us with this project.

Further, we would like to thank our examiner Peter Folkesson for taking on our Master's thesis and Tor Skoglund and Darrel Cullen for giving great feedback on our report.

Finally, we would like to thank the rest of the division for helping us with the important evaluation of the product and giving us feedback on our design choices.

Edvin Catovic & Daniel Hedberg
Gothenburg, 4th of June 2002

References

[1] *TSIM Simulator User's Manual* ver. 1.1, 2002, Gaisler Research

[2] *Rad-Hard 32-bit SPARC Embedded Processor User's Manual* Rev. F, 2001, ATMEL Wireless & uC

[3] *An overview of MIL-STD-1553*, http://www.aim-online.com, AIM GmbH

Appendix A

Users' and Developers' Guide

1 Introduction

This document is an addition to the Master's thesis report *'High level description of an ASIC implementing CPU support and I/O control´* written by Edvin Catovic and Daniel Hedberg during the spring of 2002. The purpose of this document is to reveal more details that could be of interest for users and developers of the created system.

The document is more straightforward on how to use the system and how to create additional modules for the simulator but is not stand-alone from the thesis report.

First the file structure will be presented to help the user or developer recognize the home directory `Tsim_IO`, in which the simulator is located. This is followed by instructions on how to compile and run the simulator system. Last in this chapter the source file `m1553X.c` is included. The abbreviated file, which implements the main part of the m1553 modules is thoroughly discussed.

2 The File Structure

The file structure of the created systems source is represented in Figure 2.1. AlarmSigGen and m1553 are expanded down to file level.

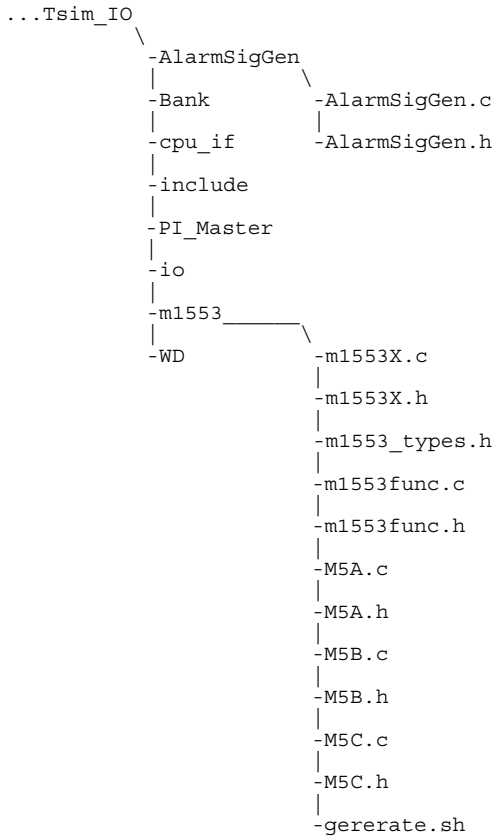

Figure 2.1: The file structure

As seen in the figure the M1553 module is implemented in a way that distinguishes it from the general case. This is an implementation suitable also for the SpaceWire, PWx and UART modules. They have in common with M1553 multiple instances of the module implemented in the COCOS.

The `m1553X.c` and `m1553X.h` files are the developer edit files. In these files generic names are used (i.e. `M1553X` and `M5X`). In the Makefile these files are processed by the shell script `generate.sh` which outputs `M5A.c, M5A.h, M5B.c, M5B.h, M5C.c` and `M5C.h`. These files are used by the compiler as sources for the three modules M1553A, M1553B and M1553C.

The files `m1553_types.h, m1553func.c` and `m1553func.h` contain help functions and data structures used by the M1553 modules.

3 Compiling and Running

The system is compiled by running

```
% make
```

in ...`Tsim_IO/io/`. This is where the `Makefile` is located.

The system is preferably run by making a soft link of the `io.so` file also located in this directory to your working directory. To start the simulator simply type

```
% tsim-erc32
```

in your working directory.

As mentioned in paragraph [4.4.1] it is useful to execute TSIM commands by using batch- or .tsimrc files [1]. This could be loading the ERC32 executable, setting up debug levels and defining stimuli files.

4 The Creation of a Submodule

The easiest way of giving an outline on how to create a submodule is probably to make an example. First of all it is advised that the section [4.3.1] on general implementation of a submodule is read.

This example is based on one of the M1553 modules. Some code has been left out due to its irrelevance in this appendix. It holds most of the possible properties in a submodule and therefore is a good example. Reading about its design in section [2.3.3] might give you an idea of what parts have similarities to your design. Remember that this implementation is just an advice and that we hold no constraints on the design of a submodule except for its interfaces to PIM. Breaking that constraint would probably make the over-all design less serviceable.

4.1 The M1553X.c File

Our environment offers many data structures that can be included. Some are optional and some are necessary. You must consult the corresponding files to decide what needs your particular module has.

```
#include <stdio.h>
#include "../include/cocos_types.h"
#include "../cpu_if/cpu_if.h"
#include "m1553_types.h"
#include "../io/clog.h"
#include "m1553.h"
```

Figure 4.1: Includes

If the module implements registers these are declared as seen in Figure 4.2.

```
/************************************************************
* M5C internal registers
 ************************************************************
static unsigned int
  M5C_REG1,
  M5C_REG2,
  M5C_REG3,
  ...
  M5C_REG<n>;

#define M5C_Virtual1    (M5C_REG1 & M5C_REG2)
#define M5C_Virtual2     (M5C_REG1 & M5C_REG2)
#define M5C_Virtual3     M5C_REG1
```

Figure 4.2: Register declarations

The defines are the way virtual registers are implemented. These have their own addresses but reflect other registers. This could include boolean operations on more than one register as seen above.

Since this module also depends on a state machine the different states are declared as seen in Figure 4.3.

```
typedef enum {
  BC_SLDisabled,
  BC_GetFirstSLBlock,
  BC_CommandOutput,
  BC_GetNextSLBlock
} m1553State_T;

static m1553State_T  m1553State;
```

Figure 4.3: State declarations

If the module needs to be initialized in some way an extern function should be produced. This function shall be called from `io_init()`, in `io.c`. For this module it looks like in Figure 4.4

```
/************************************************************
 * M5C_Init
 ************************************************************
 *
 * WHAT DESCRIPTION:
 * M5C_Init initializes M5C module.
 *
 * PARAMETERS:
 * None.
 *
 * RETURN VALUES:
 * None.
 *
 */

extern void M5C_Init()
{
  CLog(M1553C, INFO, 1, "Initialized");
  M5C_REG1 = 0x0;
  M5C_REG2 = 0x0;
  M5C_REG3 = 0x0;
  ...
  M5C_REG<n> = 0x0;
}
```

Figure 4.4: Init() function

In the beginning of this function the first call to an external function is made. `CLog(M1553C, INFO, 1, "Initialized");` is a call to the built in event logging feature. `CLog()` should be used whenever something happens in the execution that could be useful to log. The first parameter must be the name of the module. The module name must be a subset of `Module_T;` in `cocos_types.h`. The second is the type of message (INFO, ERROR or WARNING). The third is its pri-

ority (1, 2 or 3). A rule of thumb should be that priority one is used for things that should not have happened during normal execution or has great relevance (e.g. initialization). Most important is to not spam the user in priority one. Priority two could be anything that is out of the ordinary such as reading write only registers or putting the watchdog in an undefined configuration. Priority three is the rest of the messages including details for debugging purposes.

For more information refer to the `CLog()` in `CLog.c`.

The core of this module is the function m1553_Sim() in Figure 4.5 which implements a state machine controlling the more complex behaviouristics of the module. Pseudo code replaces some code blocks and are preceded by '--'.

```
/*************************************************************
 * m1553Sim()
 *************************************************************
 * WHAT DESCRIPTION:
 * m1553Sim() is a statemachine for the M5C module
 *
 * RETURN VALUES:
 * none
 *
 *************************************************************/

static void m1553C_Sim() {

   --Declarations

  switch (m1553State)
  {
/*************************
 * BC_SLDisabled:
 *************************/

    case BC_SLDisabled:
      CLog(M1553C, INFO, 2, "BC Disabled/halted");
      break;

 /* The BC is activated. */
/*************************
 * BC_GetFirstSLBlock:
 *************************/

     case BC_GetFirstSLBlock:
   --Get the first block and store it a data structure
   --Calculate CommandDelay

      /* Goto next state */
      m1553State = BC_CommandOutput;
      simif.event(m1553C_Sim, 0, (uint64) CommandDelay);

      break;

/*************************
 * BC_CommandOutput:
 *************************/

    case BC_CommandOutput:

  --Generate bustraffic

       /* Go to next state */
       m1553State = BC_GetNextSLBlock;
       simif.event(m1553C_Sim, 0, 0);

    break;

/*************************
 * BC_GetNextSLBlock:
 *************************/

    case BC_GetNextSLBlock:

  --Get the next block and store it a data structure
  --Calculate CommandDelay
```

```
      /* Go to next state */
      m1553State = BC_CommandOutput;
      simif.event(m1553C_Sim, 0, (uint64) CommandDelay);
      break;

/*************************
 * default:
 *************************/

  default:
    CLog(M1553C, ERROR, 1, "m1553C_Sim()");
    break;
  }

}
```

Figure 4.5: Sim() function

As seen above a state machine is implemented by updating a state variable i.e. m1553State and then schedule a function call of `m1553C_Sim()` through the TSIM function `simif.event()` with a calculated delay.

The registers in Figure 4.2 must be accessible by PIM through external functions such as the two in Figure 4.6 and Figure 4.7.

```
/*********************************************************
 * M5C_RegWrite()
 *********************************************************
 * WHAT DESCRIPTION:
 * M5C_RegWrite() writes registers inside the module
 *
 * PARAMETERS:
 * Addr => 32 bit address to the register.
 * Data => Points to the data to be written to the
 * register
 *
 * RETURN VALUES:
 * 0 on success, 1 otherwise.
 *
 *********************************************************/

extern int M5C_RegWrite(unsigned int Addr, unsigned int *Data) {

  CLog(M1553C, INFO, 3, "Writing %s register addr=0x%08x  value=0x%08x",
       "M1553C", Addr, *Data);

  switch (Addr) {
  case M5C_REG1_RegByteAddr:
  case M5C_REG2_RegByteAddr:
  case M5C_REG3_RegByteAddr:
    CLog(M1553C, WARNING, 1, "M5C_RegWrite:Write violation.
                              Writeprotected reg addr=0x%08x", Addr);
    break;
  case M5C_REG4_RegByteAddr      :
    M5C_REG4 = (*Data) | M5C_REG4;
    break;
  case M5C_REG5_RegByteAddr      :
    M5C_REG5 = *Data;
    break;
  case M5C_REG6_RegByteAddr      :
    M5C_REG6 = (*Data) | M5C_Stat;
    if ( (m1553State == BC_SLDisabled) &&
         (GET_BIT(M5C_REG3, 7) == 0x1) &&
         (GET_BIT(M5C_REG6,0) == 0x1) &&
         (GET_BIT(M5C_REG6,3) == 0x1) )
    {
      CLog(M1553C, INFO, 1, "SLAct set, BC active");
      m1553State = BC_GetFirstSLBlock;
      m1553C_Sim();
    }
```

```
    break;

  ...

  case M5C_REG<n>_RegByteAddr    :
    M5C_REG<n> = *Data;
    break;
  default:
    CLog(M1553C, WARNING, 2, "Write access of unimplemented or
                              nonexisting register.");
    return 1;
    break;
  }
  return 0;
}
```

Figure 4.6: RegWrite() function

As seen above in the beginning of the function a `CLog()` function may have a variable number of user defined parameters succeeding the obliged ones.

```
/*********************************************************
 * M5C_RegRead()
 *********************************************************
 * WHAT DESCRIPTION:
 * M5C_RegRead() reads registers inside the module
 *
 * PARAMETERS:
 * Data => Variable where the function should return the
 * value of the requested register.
 *
 * RETURN VALUES:
 * 0 on success, 1 otherwise.
 *
 *********************************************************/

extern int M5C_RegRead(unsigned int Addr, unsigned int *Data) {

  switch (Addr) {
  case M5C_PIMSR_RegByteAddr     :
    *Data =M5C_PIMSR;
    break;
  case M5C_PIMR_RegByteAddr      :
    *Data =M5C_PIMR;
    M5C_PIR = 0;
    CpuIf_DeassertInterrupt(M1553C); /* Clear corresponding
bit in CPUIF_IPR */
    break;
  case M5C_REG4_RegByteAddr      :
    *Data =0x0;
    CLog(M1553C, WARNING, 1, "Reading write-only M5REG4 Register");
    break;
    caseM5C_REG4_RegByteAddr     :
    *Data =0x0;
    CLog(M1553C, WARNING, 1, "Reading write-only M5REG4 Register");
    break;
    caseM5C_REG4_RegByteAddr     :
    *Data =0x0;
    CLog(M1553C, WARNING, 1, "Reading write-only M5REG4 Register");
    break;
    caseM5C_REG5_RegByteAddr     :
    *Data =M5C_REG5;
    break;
    caseM5C_REG6_RegByteAddr     :
    *Data =M5C_REG6;
    break;

    ...

    caseM5C_REG<n>_RegByteAddr   :
    *Data =M5C_REG<n>;
    break;
   default:
    CLog(M1553C, WARNING, 2, "Read access to unimplemented or
                              nonexisting register.");
    return 1;
    break;
  }
  return 0;
}
```

Figure 4.7: RegRead() function

To support the COCOS specific TSIM command dumpregs, a function of the syntax in Figure 4.9 must be implemented. The **bold** lines in `io_command()` in `io_command.c` must also be added as in Figure 4.8.

```
else if (strcmp(cmd1, "dumpregs") == 0)
{
  token = strtok(NULL, " \t\n\r");
  if (token != NULL)
  {
    if (strcmp(token, "CPUIF") == 0)
     CpuIf_DumpRegs();
    else if (strcmp(token, "PIM") == 0)
     PIM_DumpRegs();
    else if (strcmp(token, "M1553A") == 0)
     M5A_DumpRegs();
    else if (strcmp(token, "M1553B") == 0)
     M5B_DumpRegs();
    else if (strcmp(token, "M1553C") == 0)
     M5C_DumpRegs();
    else if (strcmp(token, "ASG") == 0)
     ASG_DumpRegs();
    else if (strcmp(token, "WD") == 0)
     WD_DumpRegs();
    else
     return(0);
    return 1;
  }
  else
    return(0);
}
```

Figure 4.8: Subset of io_command() function

```
/*********************************************************
 * M5C_DumpRegs()
 *********************************************************
 * WHAT DESCRIPTION:
 * M5C_DumpRegs prints the contents of all registers
 * implemented in M5C.
 *
 * PARAMETERS:
 * None
 *
 *********************************************************/

extern void M5C_DumpRegs()
{
  printf("M5C_REG1\t= 0x%08x\n", M5C_REG1                  );
  printf("M5C_REG2\t= 0x%08x\n", M5C_REG2                );
  printf("M5C_REG3\t= 0x%08x\n", M5C_REG3                 );
  ...
  printf("M5C_REG<n>\t= 0x%08x\n", M5C_REG<n>                 );
}
```

Figure 4.9: DumpRegs() function

Appendix B

Stimuli File Syntax

1 Stimuli File Syntax Definition

The bus traffic coming from modules on a 1553 bus such as RTs or a BC outside the COCOS I/O module is simulated by writing a stimuli file. An informal description of the stimuli file syntax is given in this appendix. Input files to flex (lexical analyser tool) and bison (parser tool) define the syntax of the stimuli files.
Chapter [4.4.1] describes the setstim command used at the TSIM prompt to read a stimuli file.

The stimuli file contains one section for traffic to each M1553 module in the COCOS I/O module:

```
M1553A:
     <M1553 bus traffic>

M1553B:
     <M1553 bus traffic>

M1553C:
     <M1553 bus traffic>
```

If some of M1553 modules (M1553A, M1553B, M1553C) is not implemented or not simulated the corresponding section can be left out.

For each module, the bus traffic coming from RT and/or BC unit can be described. If there are no messages coming from RT or BC units the stimuli can be left out.

```
<M1553 bus traffic> ::= <RT stimuli>
                        <BC stimuli>
```

RT Stimuli

The RT stimuli describe messages coming from RTs on the simulated bus. The RT stimuli are typically used when a BC is simulated in the COCOS I/O module in order to simulate responses from RTs on a 1553 bus.

```
<RT stimuli> ::=
RT_Response:
     <repeat> <status word> <data word 1> ... <data word n> <transfer
status> <response time>
     ...
```

Each line corresponds to a message from a RT unit and contains information on how many times the message is repeated, status word, data words, transfer status and response time.

```
<repeat> ::= <integer>
```

A message is repeated for <repeat> number of times. If set to 0, the message is repeated forever.

```
<status word> ::= 0x0 - 0xffff
```

1553 status word in hexadecimal representation. See Figure 2.5 in [2.3.3] for bit mapping.

```
<transfer status> ::=  OK
                     | error=0x0-0xffffffff
```

A hexadecimal value describing the status of the message transfer. OK indicates that no errors occurred during the message transfer otherwise the value following “error=” is mapped to the relevant bits of the local message word in the response block. If a bit is set in the <transfer status> word the corresponding bit in local message word.

```
<data word> ::= 0x0 - 0xffff
```

Data words are 16 bit values. A message from a RT unit can contain 0 to 15 data words. The data word list is left out in stimuli file if the message does not contain any data words.

```
<response time> ::=rt=<integer>
                  | NO_RT_RESPOSNE
```

Inter-message gap in bit times between the message and the next message. Special value NO_RT_RESPONSE is used to simulate lost message.

BC Stimuli

Stimuli from a BC unit is a list of messages having following format:

```
<BC stimuli> ::=
BC_Command:
     <time> <BC command> <1553 bus> <RT address> <RT subaddress> <data
word count> <data word 1> .. <data word n>

<time> ::= T<unsigned int value>
```

Time when the command should be transferred on the bus.

```
<BC command> ::= TRANSMIT
                |RECEIVE
```

Transmit or receive command.

```
<1553 bus> ::= BUS_A |BUS_B
```

The bus on which the message is sent.

```
<RT address> ::= 0 - 31
```

The value of RT address field of a 1553 command word.

```
<RT subaddress> ::= 0 - 31
```

The value of RT Subaddress field of a 1553 command word.

```
<data word count> ::= 0 - 31
```

The value of Data Word Count/Mode Code field of a 1553 command word.

```
<data word 1> ... <data word n>
```

List of 16 bit data words following a 1553 command. The list is left out if the command is not followed by data words.