School of Engineering & Design
Electronic & Computer Engineering

MSc in Distributed Computing Systems Engineering

Brunel University

High performance reconfigurable computing environment

Benjamin Krill

Dr. Abbes Amira

09/2007

A Dissertation submitted in partial fulfilment of the requirements
for the degree of Master of Science

School of Engineering & Design
Electronic & Computer Engineering

MSc in Distributed Computing Systems Engineering

Brunel University

High performance reconfigurable computing environment

Student's name: _____

Signature of student: _____

Declaration: I have read and I understand the Department's guidelines on plagiarism and cheating, and I certify that this submission fully complies with these guidelines.

Abstract: FPGA-FS provides a framework for high performance reconfigurable computer environment. The abstraction between the reconfigurable device and the user application is simplified by using a virtual filesystem and a thread abstraction per device context.

FPGA-FS consits of two major layers of abstraction. First, the directory representation to the user application. This interface provides for each context one directory with a specified content to transfer data. The second layer provides an interface for low level driver. It gives the flexibility to register a lot of different low level driver for different accelerators which can be operate at the same time in one target system.

The virtual filesystem and the developed hardware interfaces will be available for the open source community.

Contents

1 Introduction

1.1 Introduction

This is the dissertation for the MSc Dissertation in Distributed Computing Systems Engineering with the topic: Reconfigurable computers and parts in a High Performance Environment. All not explained acronyms are listed in the glossary and all book references are reflected in the bibliography.

The topic of reconfigurable computers and parts is not really new. There are many different approaches developed with different goals. To clarify the definition of a reconfigurable system, [RM98] presented the Olymp classification. This classification does not only define reconfigurable systems it also groups the systems into different categories. [HAW99] shows different criteria which can be used to classify reconfigurable systems:

- Granularity of the logic: It defines the complexity of the lowest layer of the architecture. The configurable architectures can be classified in fine, medium and coarse grained architectures. It describes how the processing unit will be operated.

- Integration to the host: It defines how the reconfigurable part is connected to the Central Processing Unit (CPU) or computer. It can be classified as dynamic, static, closely coupled and static loosely coupled architectures.

- Reconfigurability of the external interconnection network

- Speed of reconfiguration: This item points out the time which is needed to reconfigure the system. The majority of reconfigurable systems is based on Field Programmable Gate Arrays (FPGA) circuits. The FPGA circuits have a slow, serial reconfigurable path. There are several ideas how to solve this issue.
- Speed, size and density of a single node. Whereby speed is the used clock frequency.
- Memory structures and interface.
- Application development tools.

[HAW99] shows the presented Olymp classification [RM98] as a tree like the following listing:

- Fault-tolerance (ATHENA)
- Speed
- Coarse grained
  - Reconfigurable network (DYONISIUS)
  - Fixed network (PAN)
- Medium grained
  - Fixed network
    - Dynamic (HADES)
    - Static
    - Closely coupled (EOS)
    - Loosely coupled (HERA)
  - Reconfigurable network
    - Dynamic (POSEIDON)
    - Static
    - Closely coupled (HELIUS)
    - Loosely coupled (ZEUS)
- Fine grained
  - Fixed network
    - Dynamic (ARTEMIS)
    - Static
    - Closely coupled (DEMETER)
    - Loosely coupled (HEPHESTUS)
  - Reconfigurable network
    - Dynamic (APOLLO)
    - Static
    - Closely coupled (PERSEFONA)

  - Loosely coupled (APHRODITE)

The next chapter will show and describe some developed reconfigurable computer architectures. Chapter 2.2 gives the initial survey of the whole project. An overview over the aims and objectives covers chapter 1.3. The initial design is done in chapter 3.1.

1.2 Aims

The following list shows the aims which motivated me to research this topic and I try to achieve:

- to develop a reconfigurable system.
- to create a well know interface between userspace applications as well as between the kernel space and the real hardware.
- to use existing expertise as well as to learn new techniques to reach the above aims.

1.3 Objectives

During the initial survey 2.2, the following key parts of the system are identified. During the dissertation the target system has been changed. This was caused by absence hardware. So a new system design was developed.

- Target System – As already mentioned, the target system will be a host CPU connected to a southbridge and on the southbridge bus connected to the reconfigurable FPGA part. The FPGA will be connected to a JTAG port which provides a way to reconfigure.
- Interface to the FPGA – The FPGA will be connected to a system bus of the southbridge. To provide a communication interface between the southbridge bus and the computing logic this interface has to be designed. This interface should provide the possibility to send and receive data as well as commands.

- Linux device driver and API – The device driver provides the functionality to operate with the FPGA interface. In later versions the driver should provide scheduling mechanisms and calculations to determine the effective execution time (2.1.2).

1.4 Methodology and Limitations

To achieve the aims in section 1.2, the following methodology was used. A system solving the task will be implemented and analysed. The limitations are that each programme which was implemented during this dissertation will be in this phase a prototype. A further development of the complete system can only be done after the dissertation.

1.5 Dissertations Structure

The dissertation is structured as follows:

- **Chapter 1** contains this introduction.
- **Chapter 2** provides the background on the basis of this dissertation and tries to formulate the aims and objects.
- **Chapter 3** describes the design and methods on which the results of this dissertation are based.
- **Chapter 4** supplies information on the implemented software parts, on why and how parts was programmed. This chapter explains some details of design and structure of programming results without going into every detail of the written code.
- **Chapter 5** provides the conclusions drawn by this disseration.
- **Glossary** contains the glossary.
- **Bibliography** contains all referenced information.

2 Background

2.1 Background to the project

This chapter is divided into two different approaches. The first subsection deals with available reconfigurable computer systems. The other subsection deals with reconfigurable co-processor systems. Both approaches can be used to create a reconfigurable system, whereby the types of the systems are different.

2.1.1 Available Reconfigurable Computer Systems

A lot research teams worked on reconfigurable computer systems. The following subsections reflects a few results of the teams.

RAW Architecture Workstation

The RAW project is founded by the Massachusetts Institute of Technology Laboratory for Computer Science. An unique approach is described in [HAW99] which belongs to the Pan category. The system is based on simple, highly interconnected, replicated processing unit tiles. Each unit consists of a processing unit, a switching processor and a reconfigurable routing processor. The main processing unit works on the programme logic, the switching processor works on branch instructions as well as load and store instructions. The reconfigurable routing processor works autonomous and creates interconnection to other tiles. Additional to the three processors each tile has its own memory for data and instructions. A compiler does the distribution of the program logic.

Xputer

The Xputer was founded by the University of Kaiserslautern and belongs to the Eos category [HAW99]. The System does not have a hard wired Arithmetic logic unit (ALU) or an instruction sequencer. Instead of these it has a data sequencer and a reconfigurable ALU. This three major parts describe the basic structure of the Xputer system [RWH94]:

- two-dimensionally organized data memory
- reconfigurable arithmetic & logic unit (rALU) including several rALU subnets with multiple scan windows
- reconfigurable data sequencer (DS) comprising several generic address generators (GAGs)

A tool chain for the Xputer based on the C programming language has been developed. The first level of the tool chain splits the code into few parts. A code part which is executed on the host machine and the other code is executed on the Xputer. For the next step the Xputer code will be divided into a part for the address generator and a structure for the ALU array. Furthermore the compiler analyzes data dependencies among different operations and tries to find an optimal sequence. This tool chain gives the developer the option to use the Xputer without writing machine-dependent code. However the developer will get much more performance by writing machine-dependent code.

MATRIX

The MATRIX system was founded by the Massachusetts Institute of Technology and belongs to the Eos category. The system is based on configurable 8 bit units with 256 Byte of memory. Each unit could be used as data or instruction memory, as ALU or branch unit. The network between the units is based on three layers. The first layer produces fast connection between neighbours, the second layer connects far distant units and the third layer acts as a type of a global bus system. The third layer connects all units and external peripherals. To the outside such a network of computing units is visible by distributing the instructions.

Splash2

The Splash2 system was founded by the Supercomputing Research Center and belongs to the Hephestus category. The Splash system is organized in linear arrays which consist of 32 Xilinx FPGAs where each node has 128kB of memory. Each node is connected to a 68-bit data-path where the first and the last nodes are connected to the host computer through FIFO arrays. The board is connected to the host computer with two buses, one for data transfers and the other for configuration. Each FPGA can be configured separately which gives the system a better overall performance.
The development tool chain consists of several tools. One of the tools is the Logic Description Generator, which uses a common Lisp programming language for manipulating templates describing logic functions ([HAW99]). Nevertheless all tools allow only low-level programming of the FPGAs. This requires always a deep understanding of the system.

DISC

The DISC was founded by the Birgham Young University and it belongs to the Demeter category. The idea of this system is the partial reconfiguration. As the name says the system deals with a dynamic instruction set. The system handles the instructions as objects which can be allocated and rejected. Very complex operations can be combined with the basic instructions set. A new subset of instructions can also be loaded into the system. If the space requirement of the instructions exceed the capacity of the FPGA, then simply different instructions are removed after the least recently used method (LRU) from the system.

PAM-Blox

The PAM-Blox approach [OM] provides an object-oriented circuit generator on top of a PCI design environment (PamDC). The design is described in C++ and the developer has total control over placement at each level of the design hierarchy. PAM-Blox is divided into two layers, the first one can parameterize simple elements such as counters and adders. The second one can create different optimizations for

specific data-widths such as multipliers and special arithmetic units for encryption. PAM-Blox is available as an open library for the community.

RCMAT

The RCMAT is founded by the Queen's University of Belfast [AA01]. The approach combines a reconfigurable coprocessor with a general-purpose microprocessor. It was developed to as a solution for some computationally intensive tasks. The more specific task is to exploit large amount of fine grain parallelism in computationally intensive applications. A prototype implementation perform some matrix operations, transforms and decompositions.

2.1.2 Reconfigurable Co-Processors

Another approach of reconfigurable systems offers the combination of a standard Personal Computer system and a reconfigurable Co-Processor. As Co-Processor a FPGA will be used to roll out special functions from software in hardware. In the past this solution was often used for embedded systems, to accelerate special functions for those the embedded processor was too slow. New developments in this area are combining faster processors with FPGAs. These systems use a symmetric multiprocessing (SMP) board and connect the CPU to the FPGA via HyperTransport (HT) bus[1] [EET06], [Gue06]. The FPGA will be plugged into the second CPU socket. Another approach to connect a CPU with a reconfigurable module is to use the PCI-X[2] or a PCI-E[3] Bus [FPG06].

By using a FPGA as a Co-Processor to solve specific problems the time of the calculation will be combined of different aspects:

- Configuration Time: The time to configure the FPGA. In a dynamic system the configuration of the FPGA can be changed to solve different problems. This time can vary because it could be, that the FPGA has already been correctly configured or that the configuration has not yet been updated. Furthermore it depends on how it takes to reconfigure the module.

- Transfer Time: The time to transfer all needed data to the device and transfer the result back. This depends on the interconnection between the host CPU and the reconfigurable device. Further it depends on the size of the memory on the device. This means, if the data that should be computed fit not completely into the device, transfers between the computations must take place. The same issue appears also with a result which does not fit into the device memory. A solution for the data transfer can be a double

[1] HyperTransport (HT), is a bidirectional serial/parallel high-bandwidth, low-latency point to point link.

[2] PCI-X has a width of 64-bit, a maximum signaling frequency of 133 MHz (peak transfer rate of 1014 MB/s)

[3] PCI-E based on serial links called lanes and runs with 2.5 GHz, each lane carries 250 MB/s in each direction. (8 GB/s (x32))

buffering solution which guarantees that the device always has enough data to compute.

- Calculation Time: The time to compute the problem. This time depends always on the problem and how the programme of the FPGA is implemented.

Some algorithms can be better implemented in software and some others better in hardware. If algorithms will be implemented in hardware, the described aspects above must be considered. Nowadays the clock frequency of FPGAs are much lesser than the frequency of normal CPUs and a good performance can only be reached if the programme flow does not reconfigure the device very often.
The next subsection shows some developed FPGA based Co-Processor projects.

Synthup

The Synthup sound synthesis system is founded by the France Telecom. It is based on seven computing units based on FPGAs with local memory. Further it has five FPGAs for the data transfer and a control FPGA, as well as a PCI-X Bus interface which is not reconfigurable.

The system can be reconfigured within a few milliseconds. But with a bus frequency of 66 MHz (PCI) these are thousands of bus cycles. The frequency of 40 Mhz is completely enough for the application and can filter 48 audio channels in real time [Rac00].

RTOS Scheduling Co-Processor

The RTOS Scheduling Co-Processor is founded by the University of Rostock. A FPGA based process scheduler can be used in a hard real time operating system. The goal of the project is to remove the scheduling from the main processor to guarantee hard real time requests. For this the FPGA computes the whole time the priorities of all processes in parallel and saves the computed values. To signal a process scheduling it can send an interrupt to the main CPU [Hil00].

2.2 Initial survey

The goal is to develop a system which supports the dynamic, software initiated reconfiguration of a FPGA connected to a bus of a computer system. Illustration 2.1 shows a block diagram of the target system.

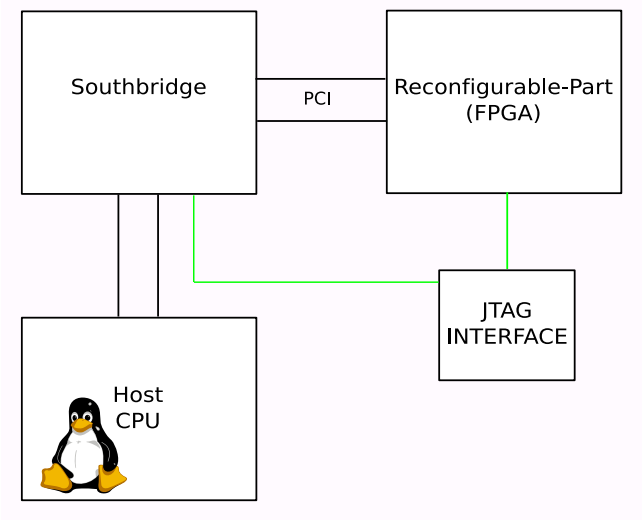

Figure 2.1: Diagram of the Target System

As mentioned in chapter 2.1 different approaches are developed by other research groups. These approaches often have the focus on hardware configurations. This dissertation will have the focus more on the software side and how a reconfigurable part can be used inside a dynamic system environment.

The first and core part of the whole architecture is to connect a reconfigurable part to a system bus. This will also cover the interface which provides the possibility to exchange data and commands to and from the device. The configuration of the reconfigurable part will be done with a Xilinx System ACE[4] module. A strategy to reprogramme the System ACE module must be developed as well.

[4]System ACE is a system-level configuration device that can configure all Xilinx FPGAs in a system.[XIL07]

The next step is to get the Linux operating system running on the target system. This also includes the detection of the interface where the reconfigurable device is attached. Further the initialisation is part of the detection process. An abstraction layer between the hardware interface and the real user application should provide as an easy to use API for the application developer. The API includes functionality to reconfigure the device and functionality to exchange data. Also a method to compute the effective execution time of the reconfiguration is part of the layer between application and the hardware interface. Also the calculation of the problem which should be solved is included into this layer. This method should have the possibility to decide when and how the request will be handled. It is conceivable that the method has a scheduler to handle more than one reconfiguration request.

All together a complete application will consists of different parts. First of all the application has a synthesised VHDL/Verilog code which can be used to programme the reconfigurable device. It also has a software part which uses the hardware part. It is conceivable that the application has a fallback part which will be used if the effective execution time of the reconfigurable software is higher than the execution time on the host CPU. Whereby the main calculation part which is the hardware part or the fallback software part can be transparently exchanged.

2.3 Implementation Approaches

This section provides information about the used platform and the used programming languages. The FPGA board was used because of the availability by the manufacturer and the possibility to connect the board directly to a PCI bus. The small price of the board also contributed to the decision. As host platform a standard desktop computer could be used. Almost all newer desktop computers have a PCI bus and provide enough slots to plug in additional cards. The decision to use Linux as operating system was done because of the openness of the kernel. Also the good and documented interfaces of the different areas are very helpful. Because of using Linux the choice of the programming language is fixed. More than 90 percent of the Linux kernel is written in C, so the usage of C is indispensable.

For the development of hardware relevant parts the programming language VHDL was chosen. This was done because the used core from Opencores is written in this language. To have a continuous coding base the same language are used for the other cores.

3 Design

3.1 Initial Design

After the definition of the aims and objectives the following initial design has been created:

3.1.1 Target System

Figure 2.1 shows a block diagram of the used target system. The first task is to get a minimal version of the Linux operating system working. This step is very important for the complete project. Without the operating system working on the target system, the project cannot be finished. The plan is to get an unmodified kernel with a tiny ramdisk working. For this only the RAM initialisation and the basic input/output have to work.

3.1.2 FGPA Interface

As shown in figure 3.1 the system consists of two FPGAs. One is connected to the CPU and the other one is connected to the first FPGA. The first FPGA acts as a southbridge. A logical bus has to be found on which the physical bus between the two FPGAs could be connected. Furthermore, a bus protocol has to be defined between the southbridge FPGA and the reconfigurable FPGA. To handle more than one logic on the reconfigurable FPGA the interface has to provide a mechanism. As a pipe for one logic on the FPGA, first in first out (FIFO) queues for data are suitable. Figure 3.1 illustrates a high level design for the interface on the reconfigurable FPGA. It shows an interface which consists of three FIFO queues connected with three different core implementations.

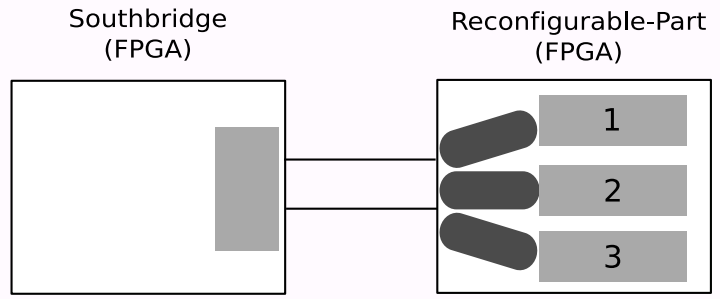

Figure 3.1: FGPA Interface

On the southbridge's side, the interface is a memory mapped input output (MMIO) interface which can be reachable and programmed by the Linux device driver.

3.1.3 Linux Interface

There are three different possible Linux interfaces. One approach is creating a character device to communicate with the kernel driver. Another approach is introducing new system calls to add the functionality. The third approach is a virtual filesystem to provide the functionality. These are the three different approaches to add the desired functionality to the Linux kernel and to create an API for the developer. To decide which interface is best suited for this project, a few arguments will be given below.

The character device approach is the simplest model for device drivers. It relies on ioctl multiplexing. A disadvantage is that resource management must be done in user space. The system call approach is a very restrictive approach because it is hard to define system calls. For very complex functionality, a large number of system calls are required and some functionality will duplicate concepts in the kernel. The virtual filesystem approach is similar to the well known procfs or sysfs. It uses existing system calls like open, read, write, mmap, etc. The abstraction of different reconfigurable parts will be done with different directories. File permissions can be used for access control.

The virtual filesystem approach provides a very good interface for the functionality which is to be provided by this project. Each reconfigurable part in a FPGA can be abstracted by a directory. The permissions of the files can provide access control for different users. Furthermore, extensions like scheduling mechanisms can be added in the background layers of the filesystem. Calculation of execution time can be performed by writing data to the available interface files. The following listing shows an example on what the directory entries might look like. These entries can be seen as the assigned context to the FPGA interface:

- /fpgafs/example/cmd: write only file, this is the interface to send defined commands to the context.
- /fpgafs/example/stat: read only file, from which the context's status can be received.
- /fpgafs/example/din: write only file, sends data to the reconfigurable device.
- /fpgafs/example/dout: read only file, receives calculated data from the device.
- /fpgafs/example/load: write only file, writes data which should be loaded onto the reconfigurable device.
- /fpgafs/foobar/lldrv: write only file, selects the low level driver which should be used.

All entries can be manipulated with standard applications like "xxd", "echo", "cat" or "dd". For a complete solution a userspace library is necessary to provide an easy to use interface for application developers.

3.2 Target System

The initial target device described in section 3.1 was changed to a standard personal computer with a x86 CPU. The update was done because of hardware issues with the described target system. The first target system was a prototype system, developed with the described features 2.2. It could not be used because during the dissertation a lot of problems occurs. Problems like connection problems between the FPGSs and configuration problems with the System ACE interface. Because of this the whole design of the target system must be changed and a standard personal computer with a PCI bus was be used. Also a different reconfigurable part has to be used. A really good FPGA board is developed by the manufacturer Enterpoint from Malvern, United Kingdom. The target system gets as operating system the distribution ArchLinux installed. The base of the distribution is a Linux kernel 2.6.22. The installation was done on a non-removable disk and can be started with the boot loader Grub.

The used FPGA is a board[1] with a Xilinx Spartan3[2] FPGA. The figure 3.2 shows the used board.

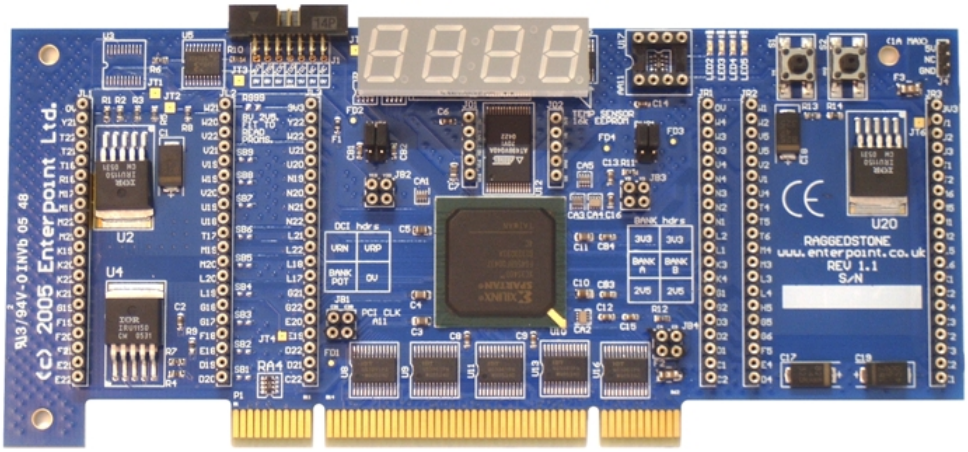

Figure 3.2: FPGA Board (Xilinx Spartan3)

[1] http://www.enterpoint.co.uk/moelbryn/raggedstone1.html
[2] http://www.xilinx.com/products/silicon_solutions/fpgas/spartan_series/index.htm

3.3 FPGA Interface

The FPGA is connected to the PCI Bus of a standard personal computer. Because of this system design, the FPGA must have a PCI Core to communicate with the host CPU. As a PCI Core, an implementation from the OpenCores project [3] was chosen. This core implements a full PCI slave and provides a Whisbone interface to the FPGA logic side. Whisbone is specified in [Her02] and will be used to create flexible designs with a standardised interconnection in order to reuse cores. The following paragraphs deals with the PCI core adaption and the interface which will be connected to the Whisbone bus.

As mentioned before, the PCI core is a fully functional slave with a Whisbone backend. The Whisbone specification [Her02] has different interconnection possibilities to connect masters and slaves. The existing interconnections are, point-to-point, data flow, shared bus, crossbar switch. Point-to-point is the simplest way to interconnect Whisbone cores. Figure 3.3 shows a point-to-point interconnection.

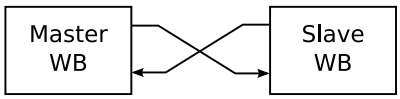

Figure 3.3: Whisbone: Point-to-point interconnection

Another possibility to connect cores is the data flow interconnection. For this solution, a Whisbone master and a slave must be integrated into each core. The arrangement looks like a pipeline. This model will often be used to create streaming solutions where each core solves a fraction of the overall problem. Figure 3.4 shows a data flow interconnection.
The shared bus interconnection method is another connection possibility. This solution connects all masters and slaves to the same bus. An arbiter is used to allow all masters to access the bus. Typical usage of this solution are PCI and VMEbus. Figure 3.5 shows a shared bus interconnection.

[3] http://www.opencores.org/projects.cgi/web/pci32tlite_oc/

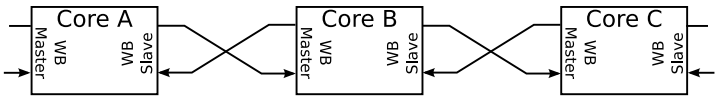

Figure 3.4: Whisbone: Data flow interconnection

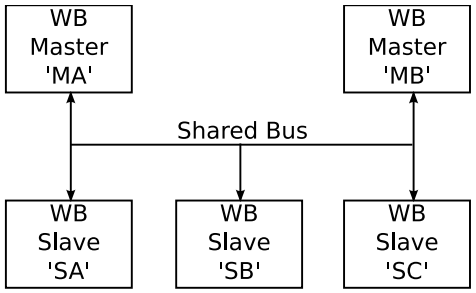

Figure 3.5: Whisbone: Shared bus interconnection

The crossbar switch interconnection is used for two or more masters and two or more slaves. Figure 3.6 shows a sample connection with two active connections. This solution works with addressing and each master can set up a connection to each slave. A arbiter is used to arrange connections between the masters and slaves.

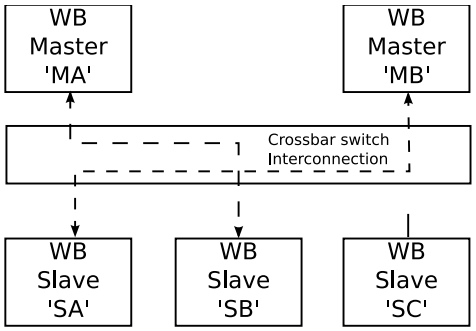

Figure 3.6: Whisbone: Crossbar switch interconnection

For this dissertation a shared bus solution is needed. Because of the different queues, different addresses on the Whisbone bus must be reachable. The PCI core will put the Wishbone address on the bus and an arbiter will decide which part on

the bus will get bus control. In the future a crossbar switch interconnection will be implemented. This gives the possibility to connect more than one accelerator function to the PCI core and consequently to the Linux framework. This means for the first implementation only one accelerator functionality is possible. Directly to the Whisbone interface, two FIFO logic components will be connected. These buffers will be used for incoming and outgoing data as well as for commands. Figure 3.7 shows how the interface looks like.

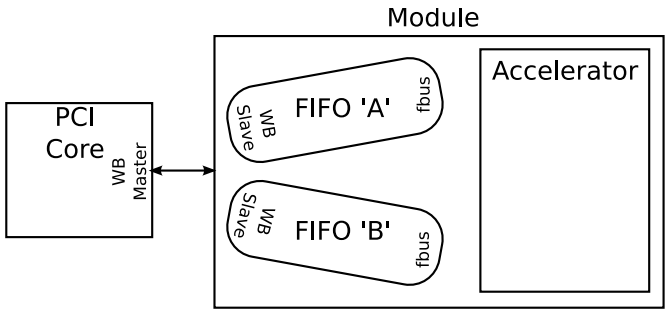

Figure 3.7: Whisbone FIFO interface

The figure shows the PCI core connected to a module which includes two FIFO queues and the accelerator logic. Both FIFO queues are connected to the Whisbone bus and have different addresses to respond to the requests. The Whisbone master writes an address to the bus and the corresponding FIFO queue receives or sends the data.

The connected accelerator hardware is attached to the queues via a data link which always points to the first entry in the queue. There are two different queues, one for incoming and one for outgoing transfers. The incoming queue has an "acknowledge" line to signal that data have been read as well as a counter signal which shows if the queue has any entries. Figure 3.8 shows the accelerator interface for the incoming queue. The accelerator interface has a data bus with a width of 16 Bits. The bus is an output signal. The data count signal gives the total number of data chunks inside the queue. To confirm a chunk of data, the acknowledge line is available. Figure 3.9 shows the accelerator interface for the outgoing queue. The

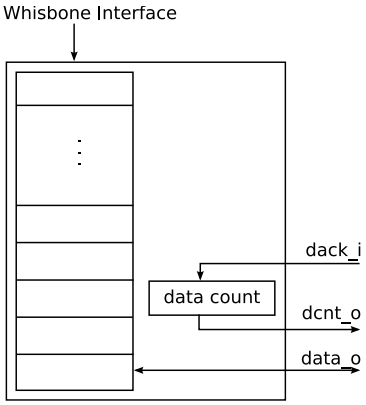

Figure 3.8: Accelerator Incoming Queue Interface

accelerator interface has a 16 bits wide data bus. The bus is an input signal. The select signal is available to signalise that data is available on the data bus and that the queue can fetch the data. This simple design is a queue to the accelerator. So it is possible that the software side can write a lot of data into the queue to guarantee that the accelerator always has enough data to work on. In the other direction, the queue guarantees that the accelerator can store the calculated data.

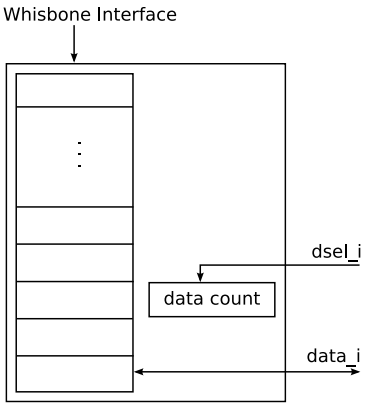

Figure 3.9: Accelerator Outgoing Queue Interface

3.3.1 FPGA Reconfiguration

Some parts of the used FPGA are connected to a JTAG bus. JTAG is standardised by the Institute of Electrical and Electronics Engineers (IEEE) as IEEE 1149.1-1990. It is a 4/5-wire bus and is an interface to test at the wafer, packaged-chip and board/system levels. It is be used to do boundary scans with standardised instructions. The commands can do in-circuit testing and board testing with a automated mechanism. Furthermore additional functionality is added by the manufacturers which allow JTAG to be used as a debug port. For example FPGA manufacturers allow configuring the FPGA trough JTAG.

The four wire bus on the used FPGA board has following signals connected:

- TCK – JTAG clock signal, all signals are synchronous to TCK (raising edge).
- TMS – TMS controls the TAP controller (next slide). If more than one IC is connected all ICs move together.
- TDI/O – Shift data into and out of a device/chain.

The mentioned TAP controller is a state machine to send or receive data to or from the FPGA. Figure 3.10 shows the TAP controller. The little numbers are the values of TMS to change the state.

Instructions which are shifted into the "Shift-IR" state are for boundary scans standardised and for debugging and programming from each manufacturer specified. The following items give a short overview of some instructions:

- IEEE specified instructions:
  - BYPASS: set chip into bypass mode (0xFF)
  - IDCODE: include the device id into DR register
- Manufacturer instructions:
  - XPROGRAM: background flashing mode (Lattice FPGA)

– ERASE: flash erase instruction (Lattice FPGA)

– INIT ADDRESS: initialize the address pointer (Lattice FPGA)

– Voltage ID (VID): include the VID into DR register (Cell/B.E.)

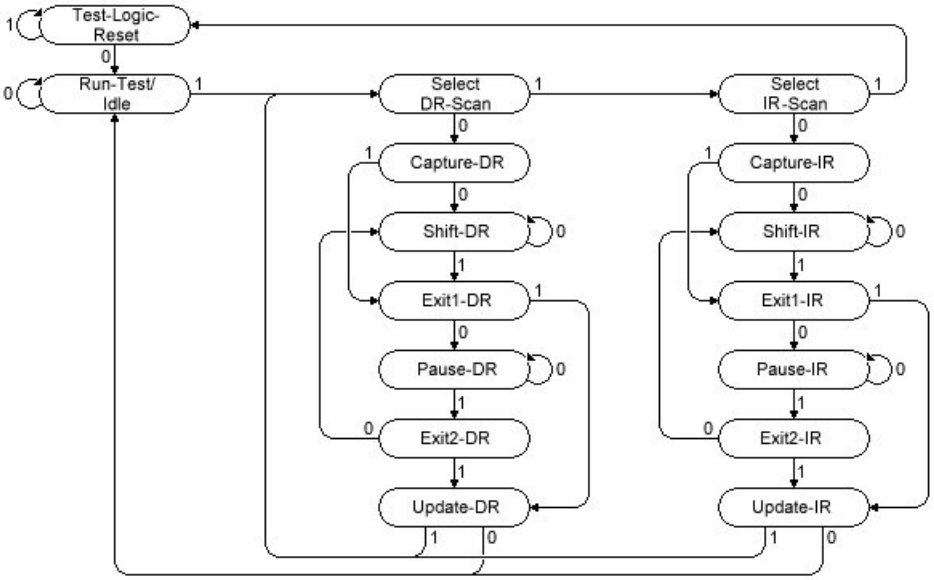

Figure 3.10: JTAG Tap Controller

3.4 Linux Interface

As described in section 3.1.3, the virtual filesystem approach was taken. It provides the needed flexibility and a well defined API, which is needed for this project. The next paragraphs deal with the complete design which will actually be implement.

3.4.1 Overall Design

In general, the complete framework provides a generic interface to the userspace and a flexible and changeable interface to the hardware. Another important part of the framework is management of the different contexts, represented by different directories. Each context has the entries defined in 3.1.3.

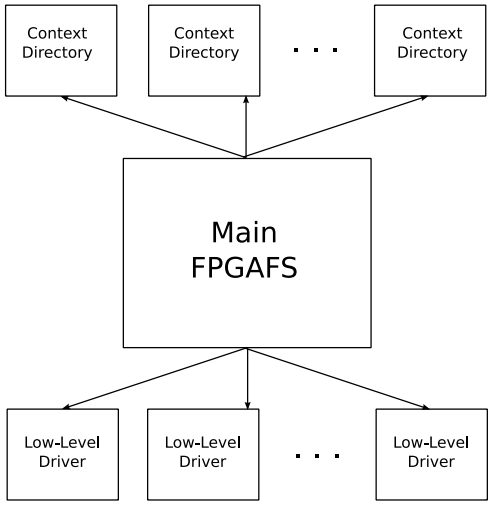

Figure 3.11: Framework overall design

Figure 3.11 shows the design of the framework. At the bottom, there are low level drivers which must be registered on the main FPGAFS. Further information can be found in 3.4.3. On top of the low-level drivers, different contexts are shown which are represented by directories.

3.4.2 Userspace API

The userspace API is represented by a virtual filesystem. The filesystem acts as a generalized abstraction layer with defined interfaces. As described in 3.1.3, the filesystem should be mounted to the location "/fpgafs". During the first phase this location can be used in conjunction with standard applications like "xxd", "echo", "cat" or "dd". During the second phase, a userspace library will be created which can be used to initialize a new context. Additional functionality provided by the library will be sending and receivng data as well as commands to and from the virtual filesystem. For that functionality the underlaying hardware will be totally transparent to the userspace application. The following items show the functions which will be implemented and will be available for userspace applications, all functions have one common parameter. This parameter points to the used directory inside the fpgafs:

- int fpgafs_create_context(const char *dir, int fd, const char *name)
  Creates a new context for the application. Parameter "fd" is a file descriptor to the file which should be loaded to the accelerator. "lldrv" select the low level driver which should be used. The last parameter is the directory of the context under the fpgafs node. Return value will be "0" on success or "-1" on failure.

- int fpgafs_send_data(const char *dir, const void *buf, size_t count)
  Sends data to the accelerator. Parameter "buf" is the pointer to the data which should be send, "count" is the size of the data chunk to be send. Return value will be the amount of data successully sent or "-1" on failure.

- int fpgafs_recv_data(const char *dir, void *buf, size_t count)
  Receives data from the accelerator. Parameter "buf" is the pointer to the location where the data should be stored, "count" is the size of the data chunk to be received. Return value will be the amount data received or "-1" on failure.

- int fpgafs_send_cmd(const char *dir, unsigned int cmd)
  Sends commands to the accelerator or low level driver. Parameter "cmd" is

the command to be send. Return value will be zero on success or "-1" on failure. Depending of the command, the return value may also indicate the status of the accelerator.

- int fpgafs_get_stat(const char *dir)
  Retrieves the status of the accelerator. Return value will be the status of the accelerator or "-1" on failure.

- int fpgafs_set_lldrv(const char *dir, const char* name)
  Sets the low level driver which should be used. Parameter "name" is the corresponding name of the low level driver. Return value will be "0" on success or "-1" on failure.

All the listed functions will be availble with the libfpgafs library and can be linked as static or dynamic functions.

3.4.3 Low Level Driver Management

The main FPGAFS module provides a mechanism to register low level drivers for different hardware accelerators. Each low level driver must implement some defined functions which will be called from the main FPGAFS module. The next sections describe the low level handling in the main FPGAFS module perspective and in a low level driver perspective.

Main FPGAFS Module

The main FPGAFS module provides two functions to register and unregister low level drivers. The module manages the drivers in an array. In the first implementation there is a static array and the module can therefore only manage a given maximum number of low level drivers. On registration of a driver the loaded driver module calls the registration function and delivers a structure with function calls. The listing 3.1 shows the delivered structure on registration and unregistration.

```
1 /* low level driver */
  struct fpgafs_lldrv {
```

```
3       char name[5];

5       int (*init) (void);
        int (*exit) (void);
7
        int (*send) (struct fpga_context *ctx, const char
           __user *buf, int len);
9       int (*recv) (struct fpga_context *ctx, unsigned char *
           buf, int len);

11      int (*cmd) (struct fpga_context *ctx, const char __user
            *buf, int len);
        int (*stat) (struct fpga_context *ctx, unsigned char *
           buf, int len);
13
        int (*read_load) (struct fpga_context *ctx, unsigned
           char *buf, int len);
15      int (*write_load) (struct fpga_context *ctx, const char
            __user *buf, int len);
};
```

Listing 3.1: Low Level registration structure

The structure provides the low level driver name, init and exit functions which will be called when a context will be created or destroyed. These events occur when a new directory is created inside the virtual filesystem or an existing directory is removed. Further content of the structure are the send and recv function pointers which will be called when a userspace program wants to send or to receive data to or from the accelerator. The files which allow sending and receiving data are "dout" and "din". The last two functions provide the functionality to load a bit stream to the accelerator and to read it back. Those functions are needed when a userspace application writes/reads data into/from the file "load". All function pointers will be called from generalized functions. Those functions always receive a parameter specifying which file is called. The file pointer provides a private area

which has a pointer to a FPGAFS context structure. The FPGAFS context structure provides information about the used low level driver. With this mechanism, each directory can have its own low level driver and is able to exchange its data or commands.

All described functionality provides a flexible virtual filesystem which can be used with different underlying hardware interfaces. The un-/registration functions will be available as an exported symbol inside the Linux Kernel. This method allows each driver to register itself with the FPGAFS framework. After loading the FPGAFS framework has no hardware knowledge and returns a busy error value on calling any function of the filesystem. After registration of a low level driver, the userspace application can choose between all available low level drivers.

Low Level Driver

A low level driver implements an interface which will be used by more generalized functions from the low level management. As shown in listing 3.1, the low level driver has to provide at least the defined function pointers. Each low level driver can be loaded and unloaded as a common Linux module. Like every Linux module, the low level driver has an init and an exit function. At load time, the init function will be called. The init function of each low level driver has to call the registration function of the FPGAFS low level management module. Nearly the same procedure will be done on unloading a low level driver. But instead of calling the registration function the unregistration function will be called.

With this concept different hardware accelerators can be supported and each low level driver can implement its own communication functions. So each low level driver brings its own programming routine for the accelerator. This gives the possibility to create accelerators that can handle more than one application at once. For example, the accelerator can reprogram parts of the accelerator and can be used without reconfiguring the whole accelerator. However, this functionality totally depends on the underlying hardware and the programming mechanism. But the framework will support such and other functionalities because of the flexible

layer concept.

4 Results and Implementations

After the design of all the relevant parts of the dissertation was finished, the following results were obtained. This chapter deals with the applications and programming languages used and the results to which they lead.

4.1 Applications and Programming

This section offers a short introduction to the applications and programming languages which were used and why they were used.

4.1.1 Applications

- Xilinx WebPAC 9.2i — The development environment WebPAC was used for the FPGA development because the soldered FPGA is a Xilinx XC3S1500. Included are the synthesis tools and all needed applications to create a complete design.
- GCC 4.1.2, Binutils 2.17.50.0.12-4 — For all programmed part on Linux this compiler and bin utilities were used. The compiler is distributed by the Free Software Foundation (FSF) under the GNU General Public License (GNU GPL).

4.1.2 Programming Languages

- C — C is a general-purpose computer programming language developed in 1972 by Dennis Ritchie and Brian Kernighan. It is popular used for systems programming. The main part of Linux is written in C. And the implemented parts, the virtual filesystem and system library are also written in C.

- VHDL — VHDL is commonly used for FPGA programming. VHDL was developed to describe the behavior of ASICs. It borrows heavily from the programming language Ada in concepts and syntax.

4.2 Linux Interface

This section describes the implementation details of the virtual filesystem and the system library.

4.2.1 Virtual Filesystem

inode.c

This is the main file where the filesystem specific functions are located. When loading the module an inode cache will be created and the filesystem will be registered to the Linux kernel filesystem registration. After this initialisation the filesystem is available and can be found in the proc filesystem under "/proc/filesystems" as FPGAFS. During the unloading the allocated inode cache will be freed and the filesystem will be unregistered.

To implement a virtual filesystem in the Linux VFS layer some functions are needed. For example the implementation must provide functions to manage the super block and functions to manage simple directory calls. The rest of the functions are needed to manage the inodes.
The next items show some functions and a description. Some of these function are directly called when operating with the filesystem and the other are only used internally. The filesystem functions are passed during the registration and creation of the filesystem superblock.

- static struct inode *fpgafs_alloc_inode(struct super_block *sb)
  This function is used to allocate new inodes. Internally the function uses the kernel internal cache allocation function and adds additional queries to detect if the function call was successful. The return value is "NULL" on fail and the pointer to the memory location on success.
- static void fpgafs_destroy_inode(struct inode *inode)
  This function is the contrary to the allocation function. It frees the allocated memory.

- static struct inode *fpgafs_new_inode(struct super_block *sb, int mode)
  To create a new inode, this function will be called. It initialises all needed fields in the inode structure like the user id, group id and mode. The return value is a pointer to the created inode structure.

- static int fpgafs_fill_dir(struct dentry *dir, struct tree_descr *files, int mode, struct fpga_context *ctx)
  This function creates the content of a directory. It will be called from the function which will create a directory. The parameter file is a structure with a list of files which should be created. With this function the FPGAFS gets the content described in 3.1.3.

- static int fpgafs_new_file(struct super_block *sb, struct dentry *dentry, const struct file_operations *fops, int mode, struct fpga_context *ctx)
  The function creates new files. It calls the function which creates new inodes and adds some more initialisations. So it is a further wrapper function to hide the creation of a inode.

- int fpgafs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
  This function will be called on creation of a directory. It also calls the function which creates new inodes. Further it calls the function which fills the directory with the defined FPGAFS content. Additionally this function creates a new context inside the framework.

- static void fpgafs_prune_dir(struct dentry *dir)
  This function removes the complete content inside a directory. This is needed to remove a directory without removing each file inside the directory. It will be called on removing a directory and on any fail on creating a directory with the FPGAFS content.

- static int fpgafs_rmdir(struct inode *dir, struct dentry *dentry)
  As the function name says, the function is called on removing a directory.

- int fpgafs_fill_sb(struct super_block *sb, void *data, int silent)
  This function is called on creating/mounting the filesystem. It sets the ini-

tial blocksize, permissions, allocates the root inode and sets the operation structure. The operation structure includes all functions which can be called during operation with the filesystem (mkdir, rmdir).

- int __init fpgafs_init(void)
  This is the initial module function. It will be called on loading the module, it creates a cache for the inodes and registers the filesystem.

- int __init fpgafs_exit(void)
  This is the exit module function. It will be called on unloading the module, it removes the inode cache and unregisters the filesystem.

A very important function is the directory creation function. Generally it creates a new directory in the virtual filesystem and fills the directory with the specified content (3.1.3). Listing 4.1 shows the implementation of the function. First of all a new inode will be created. After success a mutex will be used to lock the next operations exclusivly for this program path. Inside the mutex lock some general fields like group id and mode will be set for the directory itself. All created directories create a new context inside the FPGAFS framework. This will be done in line 19 and assigned to the context field from the inode. Furthermore the inode of the directory gets the structures with the available operations which can be applied to the directory and the files inside the directory. The next function in line 27 fills the directory with the specified content. After filling the directory some internal used counter and instantiations are called.

```
int fpgafs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
{
        int ret;
        struct inode *inode;
        struct fpga_context *ctx = NULL;

        ret = -ENOSPC;
        inode = fpgafs_new_inode(dir->i_sb, mode | S_IFDIR);
        if (!inode)
                return ret;

        mutex_lock(&inode->i_mutex);

        if (dir->i_mode & S_ISGID) {
                inode->i_gid = dir->i_gid;
                inode->i_mode &= S_ISGID;
        }
```

```
18
        ctx = alloc_fpga_context();
20      FPGAFS_I(inode)->i_ctx = ctx;
        if (!ctx)
22              goto unmutex;

24      inode->i_op = &fpgafs_simple_dir_inode_operations;
        inode->i_fop = &simple_dir_operations;
26
  ret = fpgafs_fill_dir(dentry, fpgafs_dir_contents, mode, ctx);
28
        d_instantiate(dentry, inode);
30      dget(dentry);
        dir->i_nlink++;
32      dentry->d_inode->i_nlink++;

34  unmutex:
        mutex_unlock(&inode->i_mutex);
36      return ret;
    }
```

Listing 4.1: FPGAFS Create Directory

files.c

This file includes the description of the file which will be generated on creating a new context. An overview of the files was given in chapter 3.1.3. The main purpose of the file is to define the context/directory content. Furthermore the different open, read and write functions are defined as well as the permissions of each file. All functions are used from the low level management layer to give the possibility for different accelerators and contexts. Listing 4.2 shows which files should be created during the creation of a directory. The first column is the file name, the second column specifies the file operations which should be used and the last column specifies the file permissions. Listing 4.3 shows for example the file operations for the file "load". All listed functions will be called during working with the filesystem. So the function "fpgafs_write_load" will be called on writing to the file "load".

```
1 struct tree_descr fpgafs_dir_contents[] = {
        { "din",  &fpgafs_din_fops, 0222, },
3       { "dout", &fpgafs_dout_fops, 0444, },
        { "load", &fpgafs_load_fops, 0666, },
5       { "cmd",  &fpgafs_cmd_fops,  0222, },
        { "stat", &fpgafs_stat_fops, 0444, },
7       { "lldrv", &fpgafs_lldrv_fops, 0444, },
        {},
9 };
```

Listing 4.2: Context File Structure

```
1 static const struct file_operations fpgafs_load_fops = {
        .open  = fpgafs_open,
3       .write = fpgafs_write_load,
        .read  = fpgafs_read_load,
5 };
```

Listing 4.3: File Operations Structure

llmgmt.c

This file includes all generalized read and write functions and the low level driver management functions. These functions are mainly called from the file "file.c" where the userspace interface is defined. The low level driver management functions are called from the different low level drivers. These drivers are loaded separately after the main FPGAFS framework is loaded. All functions are exported symbols inside the kernel, this provides the functionality to call the functions from the complete kernel space. Generally all functions have the same implementation because each function calls the more generalised function from the low level driver. Listing 4.4 shows an implementation as example for all other functions.

The first function is used to send data to the accelerator. The prototype of the function is a normal writing prototype, defined inside the Linux kernel. Internally the function gets the private area from the inode, where the FPGA context is stored. Inside the FPGA context the used low level driver is stored. If a low level driver is set, the function uses the array where all driver are registered and calls the specified function. The second function which is used to receive data has nearly the same implementation, the only difference is the called function of the low level driver.

```
1  ssize_t fpgafs_send_data(struct file *file, const char __user *buf,
                            size_t len, loff_t *pos)
3  {
       struct fpga_context *fcur = (struct fpga_context*)file->private_data;
5      return (fcur->lldrv > -1) ?
               lldrv[fcur->lldrv]->send(fcur, buf, len)
7              : -EBUSY;
   }
9
   ssize_t fpgafs_recv_data(struct file *file, char __user *buf,
11                          size_t len, loff_t *pos)
   {
13     struct fpga_context *fcur = (struct fpga_context*)file->private_data;
       return (fcur->lldrv > -1) ?
15             lldrv[fcur->lldrv]->recv(fcur, buf, len)
               : -EBUSY;
17 }
```

Listing 4.4: Low Level Function Generalisation

4.2.2 Low Level Driver

Two working and usable low level driver are available. On the one hand the debug low level driver which is used to debug the virtual filesystem and all the functionalities of the filesystem like the low level driver management, read/write data, creating context and all others. On the other hand there is the low level driver for the used FPGA board available. The next two sections describe each low level driver.

Debug Low Level Driver

This low level driver is implemented in the file "fpgafs_lldrv_dbg.c" of the framework. It is available under the name "dbg" after loading the module into the kernel. Specially this driver is only a debugging driver to verify all functionalities of the FPGAFS framework. During the initialisation the driver creates buffers where transfered data can be stored. The stored data can also be read back with the corresponding read function. This method gives the possibility to verify the complete chain from the low level driver to the userspace application. Listing 4.9 shows some parts of the implementation.

Raggedstone Low Level Diver

This low level driver is implemented in the file "fpgafs_lldrv_rag.c" of the framework. It is available under the name "rag" after loading the module into the kernel. This driver is a working driver for a accelerator solution which was developed in this dissertation. As mentioned in chapter 3.3 the FPGA is connected to the PCI bus of the computer. So the driver has to detect the board during loading the driver into the kernel. This will be done with internal Linux kernel functions. During loading the module the function "pci_register_driver" will be called. As parameter the function gets a structure with function pointer to probe and remove the PCI device, with a name and an additional structure. Inside the additional structure a list of devices is given for which devices the driver will be used. If the Linux kernel detects a device and finds the right kernel module, the kernel calls the PCI probe function inside the responsible module. The module

in this example will enable the device, request regions where the device should be mapped and remap the device to the virtual memory. After probing the device is available to do all specified operations. If the kernel module will be removed the removing function in the kernel module will be called. This function releases all mapped regions and deactivates the device.

To communicate with the device it reserves a 32Mbyte memory mapped region. The region is divided into different functions which are described in chapter 4.3.4. So the driver uses the different functions of the device to send or receive data. To write data into the device the module uses the Linux kernel function "__get_user(kbuf, ubuf)" to copy the userpace data into kernelspace. Afterwords it writes the transfered data with the function "writew(data, addr)" to the specified memory mapped region. Listing 4.5 shows the whole implementation of the Raggestone low level driver.

```
1  #include <linux/pagemap.h>
   #include <linux/kernel.h>
3  #include <linux/module.h>
   #include <linux/init.h>
5  #include <linux/pci.h>
   #include <linux/mm.h>
7  #include "fpgafs.h"

9  #define VENDOR_ID_ALTERA 0x1172
   #define DEVICE_ID_ALTERA 0x0100
11
   static void *vaddr = 0;
13 static unsigned long memstart = 0, memlen = 0;

15 static int fpgafs_send_data_rag(struct fpga_context *ctx, const char __user *buf, int len)
   {
17         int i;
           u8 __user *usr;
19         unsigned char dat[2];

21         if (len < 2)
                   return -EINVAL;
23
           if (!access_ok(VERIFY_READ, buf, len))
25                 return -EFAULT;

27         for(i=0; i < len; i+=2) {
                   usr = (u8*)&buf[i];
29                 if (__get_user(dat[0], usr))
                           return -EFAULT;
31
                   usr = (u8*)&buf[i+1];
33                 if (__get_user(dat[1], usr))
                           return -EFAULT;
35
```

```
                writew((dat[1] << 8)|dat[0], vaddr);
        }

        return i;
}

static int fpgafs_recv_data_rag(struct fpga_context *ctx, unsigned char *buf, int len)
{
        int i; unsigned short d;
        unsigned char b[2];

        if ((len < 2) || ((len % 2) != 0))
                return -EINVAL;

        for(i=0; i < len; i+=2) {
                d=readw((void *)((unsigned int)vaddr | 0x20));
                b[i] = d & 0xff;
                b[i+1] = d >> 8;
                if (copy_to_user(b, ctx->load_buf, 2))
                        return -EFAULT;
        }

        return i;
}

int fpgafs_device_probe_rag(struct pci_dev *dev, const struct pci_device_id *id)
{
        int ret;
        ret = pci_enable_device(dev);
        if (ret < 0) {
                printk(KERN_WARNING "FPGAFS_RAG: unable to initialize PCI device\n");
                return ret;
        }

        ret = pci_request_regions(dev, "FPGAFS_RAG");
        if (ret < 0) {
                printk(KERN_WARNING "FPGAFS_RAG: unable to reserve PCI resources\n");
                pci_disable_device(dev);
                return ret;
        }

        memstart = pci_resource_start(dev, 0); /* 0 for BAR0 */
        memlen = pci_resource_len(dev, 0);
        printk(KERN_WARNING "FPGAFS_RAG: memstart=0x%lx memlen=%li\n", memstart, memlen);

        vaddr = ioremap(memstart, memlen);
        printk(KERN_INFO "FPGAFS_RAG: device probe successful\n");
        return ret;
}

void fpgafs_device_remove_rag(struct pci_dev *dev)
{
        iounmap(vaddr);
        pci_release_regions(dev);
        pci_disable_device(dev);
        printk(KERN_INFO "FPGAFS_RAG: device removed\n");
}

static struct pci_device_id  pci_device_id_fpgafs_rag[] =
{
        {VENDOR_ID_ALTERA, DEVICE_ID_ALTERA, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0},
        {}  /* EOL */
};

struct pci_driver pci_driver_fpgafs_rag =
{
```

```
        name: "FPGAFS_RAG",
        id_table: pci_device_id_fpgafs_rag,
        probe: fpgafs_device_probe_rag,
        remove: fpgafs_device_remove_rag
};

static int fpgafs_init_rag(void)
{
        printk(KERN_INFO "FPGFS_RAG: PCI init\n");
        return pci_register_driver(&pci_driver_fpgafs_rag);
}

static int fpgafs_exit_rag(void)
{
        printk(KERN_INFO "FPGFS_RAG: PCI fini\n");
        pci_unregister_driver(&pci_driver_fpgafs_rag);
        return 0;
}

static struct fpgafs_lldrv fpgafs_lldrv_rag = {
        .name = "rag",
        .init = &fpgafs_init_rag,
        .exit = &fpgafs_exit_rag,
        .send = &fpgafs_send_data_rag,
        .recv = &fpgafs_recv_data_rag,
        .read_load = NULL,
        .write_load = NULL,
};

/* init exit functions ... */
int __init fpgafs_lldrv_rag_init(void)
{
        return fpgafs_register_lldrv(&fpgafs_lldrv_rag);
}

void __exit fpgafs_lldrv_rag_exit(void)
{
        fpgafs_unregister_lldrv(&fpgafs_lldrv_rag);
}

module_init(fpgafs_lldrv_rag_init);
module_exit(fpgafs_lldrv_rag_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Benjamin Krill <ben@codiert.org>");
```

Listing 4.5: Raggedstone Low Level Driver

4.2.3 System Library

The system library is build as shared library. The big advantages of a shared library are that every task shares the same memory space for the library. From this it follows that lesser pages are needed in RAM which cuts down paging and reduced of the overall memory footprint. A second advantage is that bug fixes can be distribute without relinking all applications based on the library. Furthermore not linking libraries into every application can save disk space and also have security advantages.

To build a shared library each file has to be compiled to an object file with the GCC parameter "-fPIC". After each file is compiled, all objects have to be linked with the parameters "shared -Wl,-soname". The listing 4.6 shows the make file which will be used to build the library.

```
CFLAGS=-c -fPIC
CC = gcc

SRCS = libfpga.c
OBJS = $(SRCS:%.c=%.o)

libfpga: ${OBJS}
	${CC} -shared -Wl,-soname,$@.so.1 -o $@.so.1.0.1 $^

clean:
	rm *.o *.so.*
```

Listing 4.6: Shared Library Makefile

4.2.4 Code Samples

This section shows some sample source code to use the system library. Furthermore the section includes some basic operations to use the virtual filesystem. Additionally a sample implementation of a low level driver is shown.

Application

Listing 4.7 shows a sample application using the userspace library libfpga (described in 3.4.2). The userspace library gives a more generalised interface to the application developer. As described in 3.4.2 the function called in line 23 is used to create a new context with the specified name and loads the given file descriptor into the accelerator. After successfully creating the context a loop sends and receives data trough the libfgpa functions (line 34, 39). The loop will stop when the status of the accelerator is bigger than ten (line 29).

```
1  /*****************************************
   * Benjamin Krill <ben@codiert.org>
3  *****************************************/
   #include <stdio.h>
5  #include <stdlib.h>
   #include <sys/types.h>
7  #include <sys/stat.h>
   #include <fcntl.h>
9  #include <errno.h>
   #include <libfpga.h>
11
   int
13 main(int argv, char **argc)
   {
15         int fd, sta, rnd, i;
           char *c, buf[4];
17
           if ((fd = open("tdat", O_RDONLY, 0)) == -1 ) {
19                 printf("ERROR-1: %s\n", strerror(errno));
                   return -1;
21         }

23         if (fpgafs_create_context(fd, "dbg") == -1) {
                   printf("ERROR-2: Cannot create fpgafs context\n");
25                 close(fd);
                   return -1;
27         }

29         while (fpgafs_get_stat() < 0x10) {
                   rnd = rand();
31                 c = (char *)&rnd;
                   printf("write data: %x\n",c[0], c[1], c[2], c[3]);
33                 if ((sta=fpgafs_send_data(&c[0], 4)) != 4) {
                           printf("ERROR-3: Cannot send specified length (%d)\n", sta);
```

```
35                      return -1;
                }
37
                if ((sta=fpgafs_recv_data(&c[0], 4)) != 4) {
39                      printf("ERROR-3: Cannot send specified length (%d)\n", sta);
                        return -1;
41              }
                printf("read data: %x\n",buf[0], buf[1], buf[2], buf[3]);
43      }
        return 0;
45 }
```

Listing 4.7: Sample FPGAFS Application

Filesystem Operations

An example how the framework is working is shown in listing 4.8. The first two lines describe the loading of the framework module and the debug low level driver. Good to see in line 3-5 is the dependency of the low level driver to the framework. Line 6 shows the filesystems registered in the kernel (other listed filesystems are removed). After creating a directory the filesystem is mounted in line 9. To see if the filesystem is mounted line 10-11 provides the currently mounted filesystems. After loading and mounting the filesystem it is possible to create contexts to work with an accelerator. In line 12 a context will be established by creating a directory. After creation the content in line 14 is available. This is the defined content of the framework description. To see if the debug low level driver works operations like "echo" and "dd" can be used. Line 15-17 shows these operations applied on the FPGAFS files.

```
1  $ insmod fpgafs.ko
   $ insmod fpgafs_lldrv_dbg.ko
3  $ lsmod
   fpgafs_lldrv_dbg        7044  0
5  fpgafs                 14876  1 fpgafs_lldrv_dbg
   $ cat /proc/filesystems
7  nodev    fpgafs
   $ mkdir fpgafs
9  $ mount -t fpgafs non fpgafs
   $ mount
```

```
non on /fpgafs type fpgafs (rw)
$ mkdir foobar
$ ls foobar/
cmd  din  dout  lldrv  load  stat
$ echo "1234" > foobar/load
$ dd if=foobar/load of=test bs=4 count=1
$ xxd test
0000000: 1234 0000
```

Listing 4.8: Sample FPGAFS Filesystem Operations

Low Level Driver

To write a low level driver the developer only has to take notice of the FPGAFS low level driver structure. During the registration the structure delivers each function which will be called from the generalised functions. The header file in the development directory has the structure definition included and each low level driver has to include this file. Listing 4.9 shows parts of a sample driver which was implemented during the development phase of the whole FPGAFS to have a low level driver which demonstrates and tests the functionality of the filesystem.

```
#include "fpgafs.h"

#define MEM_SIZE 255
/* some test memory */
static char *mem;

static int fpgafs_send_data_dbg(struct fpga_context *ctx, const char __user *buf, int len)
{
        u32 cp = 0;
        u8 __user *usr;
        [...]
        while (cp < len) {
                usr = (u8*)&buf[cp];
                if (__get_user(mem[cp], usr))
                        return -EFAULT;
                cp++;
        }

        return len;
}

static int fpgafs_recv_data_dbg(struct fpga_context *ctx, unsigned char *buf, int len)
{
        len = (len > MEM_SIZE)?MEM_SIZE:len;
        if (copy_to_user(buf, mem, len))
```

```
26                  return -EFAULT;

28          return len;
    }
30
    static int fpgafs_read_load_dbg(struct fpga_context *ctx, unsigned char *buf, int len)
32  {
            [...]
34          return len;
    }
36
    static int fpgafs_write_load_dbg(struct fpga_context *ctx, const char __user *buf, int len)
38  {
            [...]
40          return len;
    }
42
    static int fpgafs_init_dbg(void)
44  {
            mem = kmalloc(MEM_SIZE,GFP_USER);
46          return 0;
    }
48
    static int fpgafs_exit_dbg(void)
50  {
            kfree(mem);
52          return 0;
    }
54
    static struct fpgafs_lldrv fpgafs_lldrv_dbg = {
56          .name = "dbg",
            .init = fpgafs_init_dbg,
58          .exit = fpgafs_exit_dbg,
            .send = &fpgafs_send_data_dbg,
60          .recv = &fpgafs_recv_data_dbg,
            .cmd  = NULL;
62          .stat = NULL;
            .read_load = &fpgafs_read_load_dbg,
64          .write_load = &fpgafs_write_load_dbg,
    };
66
    /* init exit functions ... */
68  int __init fpgafs_lldrv_dbg_init(void)
    {
70          return fpgafs_register_lldrv(&fpgafs_lldrv_dbg);
    }
72
    void __exit fpgafs_lldrv_dbg_exit(void)
74  {
            fpgafs_unregister_lldrv(&fpgafs_lldrv_dbg);
76  }

78  module_init(fpgafs_lldrv_dbg_init);
    module_exit(fpgafs_lldrv_dbg_exit);
80
    MODULE_LICENSE("GPL");
82  MODULE_AUTHOR("Benjamin Krill <ben@codiert.org>");
```

Listing 4.9: Sample FPGAFS Low Level Driver

It can be seen in line 76-80 the low level driver is an additional Linux Kernel module. It can be loaded and unloaded during the operation of the kernel. The

only and important dependency are the functions to register and unregister the driver which will be done in line 68 and 75. These functions only need the low level driver structure as parameter. The structure is filled in line 55-63 with all needed members. All members except the "name" field are function pointers, so they will be set with a pointer to a defined function in this file. From line 7 to 55 all functions are defined which are included in the structure.

The sample gives a short overview how to write a low level driver for the FPGAFS. A more specific driver was developed for the used FPGA. Functionality inside the driver is to search the FPGA on the PCI bus and map the memory regions inside a kernel space area. Furthermore the driver handles the send and receive functions.

4.3 FPGA

This section is about the implementation details of the VHDL cores.

4.3.1 Top Design (top_acc.vhd)

This file is the top design file for the whole project. It integrates all different cores into one design. This design can be synthesized with a proper pin assignment and programmed to the FPGA board. The entity of the top design has all needed signals like the PCI signals and some signals for a digital 7-segment display. The programming will be done with a JTAG cable.

4.3.2 Whisbone FIFO Queue

As described in 3.3 there are two different FIFO queue implementations. One for incoming data and one for outgoing data. Both implementations have different interfaces to the accelerator, but both have the same Whisbone interface.

wb_fifo_in.vhd

This file includes the Whisbone FIFO incoming implementation. The entity describes the Whisbone interface which will be used for the shared bus implementation of the bus. Further it describes the accelerator interface which consists of three signals. The "data_o" signal is the 16 Bit width signal to the accelerator. The signal always points to the first position of the queue. "dcnt_o" is a counter which shows the number of pending data in the queue. With the "dack_i" signal the accelerator can acknowledge the currently pending data on the bus. This will remove the entry in the queue and the next data will be available.

Furthermore the entity has a generic entry. This entry sets the address of the core on the Whisbone bus. With this method it is possible to use a lot of instances of this core with different addresses on the bus. Whenever this core will be instantiated the generic mapping should be done. If no generic mapping is given the

default value will be used. Listing 4.10 shows the implementation.

```
------------------------------------------------------------------------
-- Benjamin Krill <ben@codiert.org>
------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.numeric_std.all;

entity wb_fifo_in is
        generic (
        pci_addr_off : std_logic_vector(24 downto 1):= x"000000"
        );
        port(
        clk_i     : in std_logic;
        nrst_i    : in std_logic;
        wb_adr_i  : in std_logic_vector(24 downto 1);
        wb_dat_o  : out std_logic_vector(15 downto 0);
        wb_dat_i  : in std_logic_vector(15 downto 0);
        wb_sel_i  : in std_logic_vector(1 downto 0);
        wb_we_i   : in std_logic;
        wb_stb_i  : in std_logic;
        wb_cyc_i  : in std_logic;
        wb_ack_o  : out std_logic;
        wb_err_o  : out std_logic;
        wb_int_o  : out std_logic;

        data_o    : out std_logic_vector(15 downto 0);
        dack_i    : in std_logic;
        dcnt_o    : out std_logic_vector(3 downto 0)
        );
end wb_fifo_in;

architecture wb_fifo_behav of wb_fifo_in is
        type mem is ARRAY(0 to 15) of std_logic_vector(15 downto 0);
        signal fifo          : mem;
        signal dcnt,dpos     : integer range 0 to 15;
        signal ws_fi, ws_fid : std_logic;
        signal ws_start      : std_logic;
begin

        -- create a tiny writestrobe
        process(clk_i, nrst_i, wb_stb_i)
        begin
                if nrst_i = '0' then
                        ws_fi    <= '0';
                        ws_fid   <= '0';
                        ws_start <= '0';
                elsif (clk_i'event and clk_i = '1') then
                        if wb_stb_i = '1' then
                                ws_fi <= '1';
                        else
                                ws_fi <= '0';
                        end if;
                        ws_fid   <= ws_fi;
                        ws_start <= ws_fi and not ws_fid;
                end if;
        end process;

        -- data in/output process
        process(clk_i, nrst_i)
        begin
```

```
                if nrst_i = '0' then
64                      dcnt <= 0;
                        dpos <= 0;
66              elsif (clk_i'event and clk_i = '1') then
                        if ws_start = '1' and wb_we_i = '1' and wb_adr_i = pci_addr_off then
68                              fifo(dcnt) <= wb_dat_i;
                                dcnt <= dcnt + 1;
70                      end if;
                        if dack_i = '1' then
72                              dpos <= dpos + 1;
                        end if;
74              end if;
        end process;
76
        wb_ack_o <= ws_start when wb_adr_i = pci_addr_off else '0';
78      dcnt_o <= conv_std_logic_vector(dcnt - dpos, 4);
        data_o <= fifo(dpos);
80  end wb_fifo_behav;
```

Listing 4.10: VHDL: Outgoing FIFO

wb_fifo_out.vhd

The entity of the FIFO queue for outgoing transfers consists of the Whisbone description and two signals for the accelerator interface. The "data_i" signal is the 16 Bit width signal to the accelerator, the signal points to the next position in the queue. "dsel_i" is the signal to signalise the queue that the data signal has valid data and the data can be put from the queue. This will set the data pointer to the next free position inside the queue.

This implementation has a generic entity. Which can be used for more instances with different addresses on the Whisbone bus. Listing 4.11 shows the implementation.

```
--------------------------------------------------------------------------------
2   -- Benjamin Krill <ben@codiert.org>
--------------------------------------------------------------------------------
4   library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
6   use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
8   use IEEE.numeric_std.all;

10  entity wb_fifo_out is
        generic (
12      pci_addr_off : std_logic_vector(24 downto 1):= x"000000"
        );
14      port(
        clk_i     : in std_logic;
16      nrst_i    : in std_logic;
```

```
        wb_adr_i  : in std_logic_vector(24 downto 1);
        wb_dat_o  : out std_logic_vector(15 downto 0);
        wb_dat_i  : in std_logic_vector(15 downto 0);
        wb_sel_i  : in std_logic_vector(1 downto 0);
        wb_we_i   : in std_logic;
        wb_stb_i  : in std_logic;
        wb_cyc_i  : in std_logic;
        wb_ack_o  : out std_logic;
        wb_err_o  : out std_logic;
        wb_int_o  : out std_logic;

        data_i    : in std_logic_vector(15 downto 0);
        dws_i     : in std_logic
        );
end wb_fifo_out;

architecture wb_fifo_behav of wb_fifo_out is
        type mem is ARRAY(0 to 15) of std_logic_vector(15 downto 0);
        signal fifo           : mem;
        signal dcnt,dpos      : integer range 0 to 15;
        signal ws_fi, ws_fid : std_logic;
        signal ws_start       : std_logic;
begin

        -- create a tiny writestrobe
        process(clk_i, nrst_i, wb_stb_i)
        begin
                if nrst_i = '0' then
                        ws_fi    <= '0';
                        ws_fid   <= '0';
                        ws_start <= '0';
                elsif (clk_i'event and clk_i = '1') then
                        if wb_stb_i = '1' then
                                ws_fi <= '1';
                        else
                                ws_fi <= '0';
                        end if;
                        ws_fid   <= ws_fi;
                        ws_start <= ws_fi and not ws_fid;
                end if;
        end process;

        -- data in/output process
        process(clk_i, nrst_i)
        begin
                if nrst_i = '0' then
                        dcnt <= 0;
                        dpos <= 0;
                elsif (clk_i'event and clk_i = '1') then
                        if ws_start = '1' and wb_we_i = '0' and wb_adr_i = pci_addr_off then
                                wb_dat_o <= fifo(dpos);
                                dpos <= dpos + 1;
                        end if;
                        if dws_i = '1' then
                                fifo(dcnt) <= data_i;
                                dcnt <= dcnt + 1;
                        end if;
                end if;
        end process;

        wb_ack_o <= ws_start when wb_adr_i = pci_addr_off else '0';
end wb_fifo_behav;
```

Listing 4.11: VHDL: Outgoing FIFO

4.3.3 PCI Core

As already mentioned in 3.3 the PCI Core is a core developed by the OpenCores project. The core is licensed to the LGPL license. The following list shows all files describing the PCI core.

- pci32tlite.vhd
  PCI Core top file. This core is limited to 32 Bit buses. It provides on BAR0 an address space of 32 MByte. This 32 MByte address space is directly connected to the Whisbone bus. The core has a 16 bit data bus.

- pcidec.vhd
  This file includes the memory and configuration decoding. It also serves some of the control PCI signals.

- pcipargen.vhd
  This is the parity generator for PCI transactions. This will be done during the data phase of a read cycle.

- pciwbsequ.vhd
  This file includes the final state machine for the PCI to Whisbone interface.

- sync.vhd
  Synchronization functions.

- pcidmux.vhd
  This file includes the multiplexer between the 16 Bit Whisbone data interface and the 32 Bit PCI bus.

- pciregs.vhd
  This file implements PCI registers like VendorID, DeviceID, RevisionID, ClassCode, SubsystemID and SubsystemVID.

In general the complete core was not modified. Only little changes were made to get the core running. Also the pin assignment had to be done to get the board working. With this core it is possible to reach a theoretical bandwidth of 0,132

GByte/s. The bandwidth is calculated with the following formula.

$$32bit * 33MHz = \frac{32*33*10^6}{8}[Byte/s] = 0,132[GByte/s]$$

The used PCI bus has a width of 32 bit and runs with 33 MHz with these values the stated bandwidth can be calculated.

4.3.4 Whisbone Accelerator (wb_acc.vhd)

This file includes the two FIFO queues, the accelerator logic as well as the Whisbone arbiter. The arbiter manages the shared Whisbone bus. The arbitration is done by addresses which are delivered by the PCI core. The arbiter handles all signals going from the Whisbone slaves to the master. More specific, the handled signals are "wb_ack_o", "wb_err_o", "wb_int_o" and "wb_dat_o" signal. These signals will be multiplexed to guarantee that only one slave communicates on the bus.

The current implementation integrates the incoming FIFO queue and the outgoing FIFO queue. With this implementation it is possible to write data into the incoming queue. The incoming queue is mapped to the address offset "0x0". If the Linux driver writes into this region, the written data is filled into the queue. Further the implementation transfers the data directly to the outgoing queue. The outgoing queue is mapped to the address offset "0x20". So the Linux driver can read back the data written into the incoming queue. For further development a acceleration logic should be integrated between the incoming and outgoing queue. An integration of a PCI interrupt which will activated when new data is available in the outgoing queue is also thinkable. This will enhance the overall performance of the whole framework. With this method the userspace application doesn't need to poll the whole time. This extension will influence each part of the framework. The low level driver for the Raggedstone board needs the possibility to receive an interrupt and also the PCI core needs the extension to trigger the interrupt line. Listing 4.12 shows the instantiation of the queues and the Whisbone arbiter.

```
--------------------------------------------------------------------------------
-- Benjamin Krill <ben@codiert.org>
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity wb_acc is
        port (
        clk_i     : in std_logic;
        nrst_i    : in std_logic;
        wb_adr_i  : in std_logic_vector(24 downto 1);
        wb_dat_o  : out std_logic_vector(15 downto 0);
        wb_dat_i  : in std_logic_vector(15 downto 0);
        wb_sel_i  : in std_logic_vector(1 downto 0);
        wb_we_i   : in std_logic;
        wb_stb_i  : in std_logic;
        wb_cyc_i  : in std_logic;
        wb_ack_o  : out std_logic;
        wb_err_o  : out std_logic;
        wb_int_o  : out std_logic
        );
end wb_acc;

architecture wb_acc_behav of wb_acc is

component wb_fifo_in
        generic (
        pci_addr_off : std_logic_vector(24 downto 1):= x"000000"
        );
        port (
        clk_i     : in std_logic;
        nrst_i    : in std_logic;
        wb_adr_i  : in std_logic_vector(24 downto 1);
        wb_dat_o  : out std_logic_vector(15 downto 0);
        wb_dat_i  : in std_logic_vector(15 downto 0);
        wb_sel_i  : in std_logic_vector(1 downto 0);
        wb_we_i   : in std_logic;
        wb_stb_i  : in std_logic;
        wb_cyc_i  : in std_logic;
        wb_ack_o  : out std_logic;
        wb_err_o  : out std_logic;
        wb_int_o  : out std_logic;

        data_o    : out std_logic_vector(15 downto 0);
        dack_i    : in std_logic;
        dcnt_o    : out std_logic_vector(3 downto 0)
        );
end component;

component wb_fifo_out
        generic (
        pci_addr_off : std_logic_vector(24 downto 1):= x"000000"
        );
        port (
        clk_i     : in std_logic;
        nrst_i    : in std_logic;
        wb_adr_i  : in std_logic_vector(24 downto 1);
        wb_dat_o  : out std_logic_vector(15 downto 0);
        wb_dat_i  : in std_logic_vector(15 downto 0);
        wb_sel_i  : in std_logic_vector(1 downto 0);
        wb_we_i   : in std_logic;
        wb_stb_i  : in std_logic;
        wb_cyc_i  : in std_logic;
```

```
66          wb_ack_o  : out std_logic;
            wb_err_o  : out std_logic;
68          wb_int_o  : out std_logic;

70          data_i    : in std_logic_vector(15 downto 0);
            dws_i     : in std_logic
72          );
end component;
74          type acc_states is (IDLE,TRANS);
            signal acc_sm           : acc_states;
76          signal acc_sm_nxt       : acc_states;

78          signal fin_data_o       : std_logic_vector(15 downto 0);
            signal fin_dack_i       : std_logic;
80          signal fin_dcnt_o       : std_logic_vector(3 downto 0);
            signal fout_data_i      : std_logic_vector(15 downto 0);
82          signal fout_dws_i       : std_logic;

84          signal wb_ack_in_o      : std_logic;
            signal wb_err_in_o      : std_logic;
86          signal wb_int_in_o      : std_logic;
            signal wb_dat_in_o      : std_logic_vector(15 downto 0);
88
            signal wb_ack_out_o     : std_logic;
90          signal wb_err_out_o     : std_logic;
            signal wb_int_out_o     : std_logic;
92          signal wb_dat_out_o     : std_logic_vector(15 downto 0);

94  begin
            process(nrst_i, clk_i)
96          begin
                    if nrst_i = '0' then
98                          acc_sm <= IDLE;
                    elsif rising_edge(clk_i) then
100                         acc_sm <= acc_sm_nxt;
                    end if;
102         end process;

104         process(acc_sm, fin_dcnt_o)
            begin
106                 fin_dack_i <= '0';
                    fout_dws_i <= '0';
108                 case acc_sm is
                    when IDLE =>
110                         if fin_dcnt_o /= x"0" then
                                    acc_sm_nxt <= TRANS;
112                         else acc_sm_nxt <= IDLE;
                            end if;
114                 when TRANS =>
                            fin_dack_i <= '1';
116                         fout_dws_i <= '1';
                            acc_sm_nxt <= IDLE;
118                 end case;
            end process;
120
            fout_data_i <= fin_data_o;
122
fin: wb_fifo_in
124         generic map (pci_addr_off => x"000000")
            port map (
126         clk_i     => clk_i,
            nrst_i    => nrst_i,
128         wb_adr_i  => wb_adr_i,
            wb_dat_o  => wb_dat_in_o,
130         wb_dat_i  => wb_dat_i,
            wb_sel_i  => wb_sel_i,
```

```
132         wb_we_i   => wb_we_i,
            wb_stb_i  => wb_stb_i,
134         wb_cyc_i  => wb_cyc_i,
            wb_ack_o  => wb_ack_in_o,
136         wb_err_o  => wb_err_in_o,
            wb_int_o  => wb_int_in_o,
138
            data_o    => fin_data_o,
140         dack_i    => fin_dack_i,
            dcnt_o    => fin_dcnt_o
142         );

144 fout: wb_fifo_out
            generic map (pci_addr_off => x"000010")
146         port map (
            clk_i     => clk_i,
148         nrst_i    => nrst_i,
            wb_adr_i  => wb_adr_i,
150         wb_dat_o  => wb_dat_out_o,
            wb_dat_i  => wb_dat_i,
152         wb_sel_i  => wb_sel_i,
            wb_we_i   => wb_we_i,
154         wb_stb_i  => wb_stb_i,
            wb_cyc_i  => wb_cyc_i,
156         wb_ack_o  => wb_ack_out_o,
            wb_err_o  => wb_err_out_o,
158         wb_int_o  => wb_int_out_o,

160         data_i    => fout_data_i,
            dws_i     => fout_dws_i
162         );

164         wb_ack_o <= wb_ack_out_o when wb_adr_i = x"000010" else  wb_ack_in_o;
            wb_err_o <= wb_err_out_o when wb_adr_i = x"000010" else  wb_err_in_o;
166         wb_int_o <= wb_int_out_o when wb_adr_i = x"000010" else  wb_int_in_o;
            wb_dat_o <= wb_dat_out_o;
168
end wb_acc_behav;
```

Listing 4.12: VHDL: Accelerator Implementation

5 Conclusions

5.1 Overall Aims

Compared with the aims stated in section 1.2, I rate that this dissertation reached a lot of the aims. The first aim, to develop a system which can be reconfigured, was reached. The only deduction is that the current implementation does not have a low level driver include for the reconfiguration during runtime. This is due to the hardware change during the thesis. The planned originally method was to utilise a System ACE interface to transfer the design into the FPGA. Currently the used FPGA board only has a JTAG interface to programme the board. However, also for this board it should be possible to programme the board during runtime. In further implementations the JTAG interface should be used to programme the board.

The second aim, to create interfaces between all know related parts has been fulfilled. Each interface has a well defined programming interface. The userspace library provides an API for the developer and creates a further abstraction level to the virtual filesystem. The virtual filesystem provides an interface between userspace and kernelspace. It provides a well defined structure to communicate with the different hardware components which could be integrated with the hardware low level driver abstraction interface. The low level driver interface provides the possibility to write a specific driver for each hardware and provides a defined interface to the FPGAFS framework. Also the design of the internal reconfigurable device is solidified the internals of the low level driver. Thus the designed FPGA interface could be totally hidden to the upper layers. The designed FPGA interface can be changed and only an upgrade of the low level driver is necessary. In combination, the three interfaces provide a very flexible framework to create a to-

tal free programmable and well defined programming environment. The complete framework gets more flexiblity with the low level driver interface, because every new hardware component can provide its own low level parts and initialisations.
The third and last aim, to use existing expertise as well as learn new techniques, was the easiest goal to fulfil and was of course reached. A good example of fulfilling this goal is the programming effort required to implement the different parts of this project.

5.2 Implementation Specific Goals

The aims and objectives formulated during section 1.3 have proved to be quite reasonable. During the whole time the dissertation was under preparation these aims did not change and could therefore be followed more or less pretty closely. Although the target system has changed completely during the interim report and the writing this document. An issue on the target system occurs and the target has to be changed and not the whole dissertation scope could be done. The change required a change of the FPGA interface which is now a PCI bus between the host CPU and the FPGA. Also the reconfiguration of accelerator has been changed. The first target should provide a Xilinx System ACE interface to reconfigure the part, but the board used in this dissertation only had a JTAG port to reconfigure the parts on the board. This is a change which is not solved within this disseration and will be left for further developments.
The next aim from section 1.3 is to provide an FPGA interface between the host CPU/southbridge and the reconfigurable part. This interface is necessary to send and receive data and commands to control the logic on the reconfigurable part. This goal is reached by using a PCI core which provides an abstraction between a PCI bus and a Whisbone bus. The complete logic for sending and receiving data is done behind the Whisbone interface. To connect more than one slave to the bus a shared bus implementation has been done, so a lot of slaves can be connected to the bus and each slave has a different address which can be called from the PCI bus. To buffer commands or data a FIFO queue implementation has been developed. It also provides an easy to use interface to an accelerator.

Each accelerator can have multiple incoming and outgoing queues to transfer data and status to the low level driver and also to receive data and commands. The implemented queues have the possibility to get instantiated multiple times. This is done with a generic value in VHDL which must be set on each instantiation. This is a common used technique in the VHDL implementation.
The next aim from section 1.3 is to create a device driver for the used hardware and an API for the developer. These aims are both fulfilled. A complete framework has been developed which provides an interface through the virtual filesystem. As well as an interface through the system library. The virtual filesystem is a combination of a management module, a filesystem representation and low level drivers. The filesystem representation provides the interface between userspace and kernelspace with standard systemcalls. Also a usage permission is done by file permissions, so each accelerator context can be secured by file permissions. The representation also gives a standardised interface for each context. To provide a framework for several hardware parts the management module gives the possibility to register new low level driver to communicate with the hardware. The low level driver has all hardware specific functions included and exports the functions needed by the framework. So each layer hides the more specific implementations behind generalised functions. Additional a system library has been implemented. This library implements all standard calls to a developer which relieves work of the developer and reduces code duplication. Another important point is that the source is extensible to fulfil the further extension for this aim.
During the whole development phase different short test implementations were created. For this different testcases were developed and implemented to verify each programmed component. Furthermore an application was developed which make use of the system library and of the used FPGA board. This application test the whole chain from the system library to the real hardware. So with this test it can be shown that the implemented source is fully working.
For further developments the FPGAFS will be published to the Open Source community in the next month. Also an implementation of a functional accelerator will be done, to show the benefits of such a framework.

5.3 Final Conclusion

All together the dissertation was a very good experience where every layer of a computer system was touched. Not only the software side was esteemed but also the hardware perspectives are mentioned. This gave a very deep insight into the development and engineering of a whole target system which not only mention the hardware side. The dissertation gave also a very deep insight into the Linux kernel and its different layers like the virtual filesystem layer or the PCI subsystem. Not only the understanding of already available subsystems was necessary but also the designing and developing of new parts was done. The complete interacting of all parts was very fantastic and the leverage of the reconfigurable devices was a very great experience. So, the hope is that after releasing the project to the Open Source Community, further developments will be done and more accelerator boards can be supported.

Glossary

ALU	Arithmetic logic unit, 6
API	Application programming interface, 12, 15, 24, 25, 57, 59
CPU	Central Processing Unit, 1, 3, 9, 10, 12, 14, 17, 18, 58
FIFO	First In First Out, 14, 20, 48, 50, 53, 58
FPGA	Field Programmable Gate Array, 2–4, 7, 9–12, 14, 16, 18, 30, 31, 38, 47, 48, 57–59
HT	HyperTransport, 9
Inode	A data structure on a traditional Unix-style file system, 32
JTAG	Joint Test Action Group, 22, 48, 57, 58
LGPL	GNU Lesser General Public License, 52
LRU	Least recently used, 7
MMIO	Memory Mapped Input Output, 15
PCI	Peripheral Component Interconnect, 12, 18–20, 38, 47, 48, 52, 53, 58, 60

PCI-E Peripheral Component Interconnect Express, 9

PCI-X Peripheral Component Interconnect Extended, 9, 10

SMP Symmetric multiprocessing, 9

VHDL Very High Speed Integrated Circuit Hardware Description Language, 13, 31, 48, 59

VMEbus VMEbus International Trade Association, 18

Bibliography

[AA01] A. Amira, P. M. ; Computing Machinery, Association for (Ed.): *RCMAT: A Reconfigurable Coprocessor for Matrix Algorithms*. Version: 2001. http://www.sigda.org/Archives/ProceedingArchives/Compendiums/papers/2001/fpga01/htmfiles/sun_sgi/frames/fpgaabs.htm#pos17. – Online–Resource, checked: 09/24/2007

[EET06] Bursky, David (Ed.): *FPGA-based accelerators boost Opteron*. Version: 2006. http://www.eetimes.com/showArticle.jhtml?articleID=188702712. – Online–Resource, checked: 06/16/2007

[FPG06] Semiconductor, Lattice (Ed.): *Lattice and Northwest Logic Deliver PCI Express X1, X4 and X8 Solutions*. Version: 2006. http://www.fpgajournal.com/news_2006/06/20060605_01.htm. – Online–Resource, checked: 06/16/2007

[Gue06] Guenther, Thomas (Ed.): *FPGA-Coprocessors inside AMD64-Systems*. Elektronik, 24/2006. – p. 72–75

[HAW99] Radunovic, Bozidar (Ed.): *An Overview of Advances in Reconfigurable Computing Systems*. Version: 1999. http://csdl2.computer.org/comp/proceedings/hicss/1999/0001/03/00013039.PDF. – Online–Resource, checked: 06/10/2007

[Her02] Herveille, Richard (Ed.): *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Rev B.3*. OpenCores Organization, 9/7/2002

[Hil00] Hildebrandt, D.: *An FPGA Based Scheduling Coprocessor for Dynamic Priority Scheduling in Hard Real-Time Systems. In: Field Programmable Logic and Applications.* Proceedings of 10th International Conference, FPL 2000. Springer Verlag, 2000. – p. 777–780

[Kop96] Kopka, Helmut: *LaTeXEinführung.* Band 1. Addison-Wesley, 1996

[Lov05] Love, Robert: *Linux Kernel Development 2nd.* Novell Press, 2005. – ISBN 0–672–32720–1

[OM] Oskar Mencer, Michael J. F.: PAM-Blox: High Performance FPGA Design for Adaptive Computing. In: *IEEE*

[Rac00] Raczinksi, S.: *The Modular Architecture of Synthip, FPGA Based PCI Board for Real-Time Sound Synthesis and Digital Signal Processing. In: Field Programmable Logic and Applications.* Proceedings of 10th International Conference, FPL 2000. Springer Verlag, 2000. – p. 834–837

[RM98] Radunovic, Bozidar ; Milutinovic, Veljko M.: A Survey of Reconfigurable Computing Architectures. In: Hartenstein, Reiner W. (Ed.) ; Keevallik, Andres (Ed.): *FPL* Bd. 1482, Springer, 1998. – ISBN 3–540–64948–4, p. 376–385

[RWH94] Reiner W. Hartenstein, Karin S.: A Restructuring Compilation Method for the Xputer Paradigm. In: *IWPP94*, 1994

[XIL07] Xilinx, Inc. (Ed.): *System ACE.* Version: 1994-2007. http://www.xilinx.com/products/silicon_solutions/proms/system_ace/. – Online–Resource, checked: 06/21/2007

List of Figures

Listings

A Appendix

VHDL Top Design (top_acc.vhd):

```
1  ------------------------------------------------------------------------------
   -- Benjamin Krill <ben@codiert.org>
3  ------------------------------------------------------------------------------
   library ieee;
5  use ieee.std_logic_1164.all;

7  entity pci_acc is
   port (
9          -- General
           PCI_CLK      : in std_logic;
11         PCI_nRES     : in std_logic;

13         -- PCI target 32bits
           PCI_AD       : inout std_logic_vector(31 downto 0);
15         PCI_CBE      : in std_logic_vector(3 downto 0);
           PCI_PAR      : out std_logic;
17         PCI_nFRAME   : in std_logic;
           PCI_nIRDY    : in std_logic;
19         PCI_nTRDY    : out std_logic;
           PCI_nDEVSEL  : out std_logic;
21         PCI_nSTOP    : inout std_logic;
           PCI_IDSEL    : in std_logic;
23         PCI_nPERR    : inout std_logic;
           PCI_nSERR    : inout std_logic;
25         PCI_nINT     : out std_logic;
           PCI_nREQ     : in std_logic;
27         PCI_nGNT     : in std_logic;

29         -- 7seg
           DISP_SEL     : inout std_logic_vector(3 downto 0);
31         DISP_LED     : out std_logic_vector(6 downto 0);

33         -- debug signals
           LED_INIT     : out std_logic;
35         LED_ACCESS   : out std_logic;
           LED_ALIVE    : out std_logic;
37
           PREVENT_STRIPPING_OF_UNUSED_INPUTS : out std_logic
39 );
   end pci_acc;
41
   architecture pci_acc_arch of pci_acc is
43
   component pci32tlite
45         port (
           -- General
47         clk33        : in std_logic;
           nrst         : in std_logic;
```

```
49
        -- PCI target 32bits
51      ad          : inout std_logic_vector(31 downto 0);
        cbe         : in std_logic_vector(3 downto 0);
53      par         : out std_logic;
        frame       : in std_logic;
55      irdy        : in std_logic;
        trdy        : out std_logic;
57      devsel      : out std_logic;
        stop        : out std_logic;
59      idsel       : in std_logic;
        perr        : out std_logic;
61      serr        : out std_logic;
        intb        : out std_logic;
63
        -- Master whisbone
65      wb_adr_o    : out std_logic_vector(24 downto 1);
        wb_dat_i    : in std_logic_vector(15 downto 0);
67      wb_dat_o    : out std_logic_vector(15 downto 0);
        wb_sel_o    : out std_logic_vector(1 downto 0);
69      wb_we_o     : out std_logic;
        wb_stb_o    : out std_logic;
71      wb_cyc_o    : out std_logic;
        wb_ack_i    : in std_logic;
73      wb_err_i    : in std_logic;
        wb_int_i    : in std_logic;
75
        -- debug signals
77      debug_init  : out std_logic;
        debug_access: out std_logic
79      );
    end component;
81

83  component wb_7seg
        port (
85      -- General
        clk_i       : in std_logic;
87      nrst_i      : in std_logic;

89      -- Master whisbone
        wb_adr_i    : in std_logic_vector(24 downto 1);
91      wb_dat_o    : out std_logic_vector(15 downto 0);
        wb_dat_i    : in std_logic_vector(15 downto 0);
93      wb_sel_i    : in std_logic_vector(1 downto 0);
        wb_we_i     : in std_logic;
95      wb_stb_i    : in std_logic;
        wb_cyc_i    : in std_logic;
97      wb_ack_o    : out std_logic;
        wb_err_o    : out std_logic;
99      wb_int_o    : out std_logic;

101     -- 7seg
        DISP_SEL    : inout std_logic_vector(3 downto 0);
103     DISP_LED    : out std_logic_vector(6 downto 0)
        );
105 end component;

107 component wb_acc
        port (
109     -- General
        clk_i       : in std_logic;
111     nrst_i      : in std_logic;

113     -- Master whisbone
        wb_adr_i    : in std_logic_vector(24 downto 1);
```

```
115         wb_dat_o    : out std_logic_vector(15 downto 0);
            wb_dat_i    : in std_logic_vector(15 downto 0);
117         wb_sel_i    : in std_logic_vector(1 downto 0);
            wb_we_i     : in std_logic;
119         wb_stb_i    : in std_logic;
            wb_cyc_i    : in std_logic;
121         wb_ack_o    : out std_logic;
            wb_err_o    : out std_logic;
123         wb_int_o    : out std_logic
            );
125 end component;

127         signal  wb_adr      : std_logic_vector(24 downto 1);
            signal  wb_dat_out  : std_logic_vector(15 downto 0);
129         signal  wb_dat_in   : std_logic_vector(15 downto 0);
            signal  wb_sel      : std_logic_vector(1 downto 0);
131         signal   wb_we       : std_logic;
            signal  wb_stb      : std_logic;
133         signal  wb_cyc      : std_logic;
            signal  wb_ack      : std_logic;
135         signal  wb_err      : std_logic;
            signal  wb_int      : std_logic;
137
    --attribute s: string; -- SAVE NET FLAG
139 --attribute s of PCI_nREQ: signal is "yes";
    --attribute s of PCI_nGNT: signal is "yes";
141 begin
            LED_ALIVE <= '1';
143         PREVENT_STRIPPING_OF_UNUSED_INPUTS <= PCI_nREQ and PCI_nGNT;
            --PCI_nREQ <= 'Z';
145         --PCI_nGNT <= 'Z';

147         -- PCI Core
            u_pci: component pci32tlite
149         port map(
            clk33           => PCI_CLK,
151         nrst            => PCI_nRES,
            ad              => PCI_AD,
153         cbe             => PCI_CBE,
            par             => PCI_PAR,
155         frame           => PCI_nFRAME,
            irdy            => PCI_nIRDY,
157         trdy            => PCI_nTRDY,
            devsel          => PCI_nDEVSEL,
159         stop            => PCI_nSTOP,
            idsel           => PCI_IDSEL,
161         perr            => PCI_nPERR,
            serr            => PCI_nSERR,
163         intb            => PCI_nINT,
            wb_adr_o        => wb_adr,
165         wb_dat_i        => wb_dat_out,
            wb_dat_o        => wb_dat_in,
167         wb_sel_o        => wb_sel,
            wb_we_o         => wb_we,
169         wb_stb_o        => wb_stb,
            wb_cyc_o        => wb_cyc,
171         wb_ack_i        => wb_ack,
            wb_err_i        => wb_err,
173         wb_int_i        => wb_int,
            debug_init      => LED_INIT,
175         debug_access    => LED_ACCESS
            );
177
            -- fifo
179         u_wb_fifo: component wb_acc
            port map(
```

```
181         clk_i    => PCI_CLK,
            nrst_i   => PCI_nRES,
183         wb_adr_i => wb_adr,
            wb_dat_o => wb_dat_out,
185         wb_dat_i => wb_dat_in,
            wb_sel_i => wb_sel,
187         wb_we_i  => wb_we,
            wb_stb_i => wb_stb,
189         wb_cyc_i => wb_cyc,
            wb_ack_o => open,
191         wb_err_o => wb_err,
            wb_int_o => wb_int
193         );

195         -- 7 segment
            u_wb_7seg: component wb_7seg
197         port map(
            clk_i    => PCI_CLK,
199         nrst_i   => PCI_nRES,
            wb_adr_i => wb_adr,
201         wb_dat_o => open,
            wb_dat_i => wb_dat_in,
203         wb_sel_i => wb_sel,
            wb_we_i  => wb_we,
205         wb_stb_i => wb_stb,
            wb_cyc_i => wb_cyc,
207         wb_ack_o => wb_ack,
            wb_err_o => open,
209         wb_int_o => open,
            DISP_SEL => DISP_SEL,
211         DISP_LED => DISP_LED
            );
213 end pci_acc_arch;
```

Listing A.1: VHDL: Top Design

Main Filesystem Implementation (inode.c):

```
1  /*****************************************
    * Benjamin Krill <ben@codiert.org>
3   *****************************************/
   #include <linux/module.h>
5  #include <linux/version.h>
   #include <linux/init.h>
7  #include <linux/fs.h>
   #include <linux/pagemap.h>
9  #include <linux/slab.h>

11 #include "fpgafs.h"

13 static struct inode *fpgafs_alloc_inode(struct super_block *sb);
   static void fpgafs_destroy_inode(struct inode *inode);
15 static void fpgafs_delete_inode(struct inode *inode);
   static struct inode *fpgafs_new_inode(struct super_block *sb, int mode);
17
   static void fpgafs_prune_dir(struct dentry *dir);
19 int fpgafs_mkdir(struct inode *dir, struct dentry *dentry, int mode);
   static int fpgafs_rmdir(struct inode *dir, struct dentry *dentry);
21
   static int fpgafs_new_file(struct super_block *sb, struct dentry *dentry,
23                            const struct file_operations *fops, int mode,
                              struct fpga_context *ctx);
25 static int fpgafs_setattr(struct dentry *dentry, struct iattr *attr);
```

```
27  const struct file_operations fpgafs_context_foperations = {
            .open     = dcache_dir_open,
29          //.release  = fpgafs_dir_close,
            .llseek   = dcache_dir_lseek,
31          .read     = generic_read_dir,
            .readdir  = dcache_readdir,
33          .fsync    = simple_sync_file,
    };
35
    const struct inode_operations fpgafs_simple_dir_inode_operations = {
37          .lookup = simple_lookup
    };
39
    const struct inode_operations fpgafs_dir_inode_operations = {
41          .lookup = simple_lookup,
            .mkdir  = fpgafs_mkdir,
43          .rmdir  = fpgafs_rmdir
    };
45
    static struct kmem_cache *fpgafs_inode_cache;
47
    /* basic inode operations */
49  static struct inode *fpgafs_alloc_inode(struct super_block *sb)
    {
51          struct fpgafs_inode_info *fsi;

53          fsi = kmem_cache_alloc(fpgafs_inode_cache, GFP_KERNEL);
            if (!fsi)
55                  return NULL;

57    fsi->i_ctx = NULL;

59    return &fsi->vfs_inode;
    }
61
    static void fpgafs_destroy_inode(struct inode *inode)
63  {
            kmem_cache_free(fpgafs_inode_cache, FPGAFS_I(inode));
65  }

67  static void fpgafs_delete_inode(struct inode *inode)
    {
69          //struct fpgafs_inode_info *fsi = FPGAFS_I(inode);

71  //      XXX: do some fpga interface operations....
    //      if (fsi->i_ctx)
73  //              put_fpga_context(ei->i_ctx);
            clear_inode(inode);
75  }

77  static struct inode *fpgafs_new_inode(struct super_block *sb, int mode)
    {
79          struct inode *inode;

81          inode = new_inode(sb);
            if (!inode)
83                  return inode;

85          inode->i_mode = mode;
            inode->i_uid = current->fsuid;
87          inode->i_gid = current->fsgid;
            inode->i_blocks = 0;
89          inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
      return inode;
91  }
```

```
93  /* FPGA FS specific functions */
    static int fpgafs_fill_dir(struct dentry *dir, struct tree_descr *files,
95                             int mode, struct fpga_context *ctx)
    {
97          struct dentry *dentry;
            int ret;
99
            while (files->name && files->name[0]) {
101                 ret = -ENOMEM;
                    dentry = d_alloc_name(dir, files->name);
103                 if (!dentry)
                            goto out;
105                 ret = fpgafs_new_file(dir->d_sb, dentry, files->ops,
                                          files->mode & mode, ctx);
107                 if (ret)
                            goto out;
109                 files++;
            }
111         return 0;
    out:
113   fpgafs_prune_dir(dir);
      return ret;
115 }

117
    /* specific operations */
119 static int fpgafs_setattr(struct dentry *dentry, struct iattr *attr)
    {
121         struct inode *inode = dentry->d_inode;

123         if ((attr->ia_valid & ATTR_SIZE) &&
                (attr->ia_size != inode->i_size))
125                 return -EINVAL;
            return inode_setattr(inode, attr);
127 }

129 static int fpgafs_new_file(struct super_block *sb, struct dentry *dentry,
                               const struct file_operations *fops, int mode,
131                            struct fpga_context *ctx)
    {
133         static struct inode_operations fpgafs_file_iops = {
                    .setattr = fpgafs_setattr,
135         };
            struct inode *inode;
137         int ret;

139         ret = -ENOSPC;
            inode = fpgafs_new_inode(sb, S_IFREG | mode);
141         if (!inode)
                    goto out;
143
            ret = 0;
145         inode->i_op = &fpgafs_file_iops;
            inode->i_fop = fops;
147         inode->i_private = FPGAFS_I(inode)->i_ctx = ctx;
            d_add(dentry, inode);
149 out:
            return ret;
151 }

153 int fpgafs_mkdir( struct inode *dir, struct dentry *dentry, int mode)
    {
155         int ret;
            struct inode *inode;
157         struct fpga_context *ctx = NULL;
```

```
159             ret = -ENOSPC;
                inode = fpgafs_new_inode(dir->i_sb, mode | S_IFDIR);
161             if (!inode)
                        return ret;
163
                mutex_lock(&inode->i_mutex);
165
                if (dir->i_mode & S_ISGID) {
167                     inode->i_gid = dir->i_gid;
                        inode->i_mode &= S_ISGID;
169             }

171             ctx = alloc_fpga_context();
                FPGAFS_I(inode)->i_ctx = ctx;
173             if (!ctx)
                        goto unmutex;
175
                inode->i_op = &fpgafs_simple_dir_inode_operations;
177             inode->i_fop = &simple_dir_operations;

179       ret = fpgafs_fill_dir(dentry, fpgafs_dir_contents, mode, ctx);
                ret = 0;
181       //if (ret) {
                //      FREEE CONTENT
183             //      goto unmutex;
                //}
185
                d_instantiate(dentry, inode);
187             dget(dentry);
                dir->i_nlink++;
189             dentry->d_inode->i_nlink++;

191     unmutex:
                mutex_unlock(&inode->i_mutex);
193             return ret;
        }
195
        static void fpgafs_prune_dir(struct dentry *dir)
197     {
                struct dentry *dentry, *tmp;
199             //mutex_lock(&dir->d_inode->i_mutex);
                list_for_each_entry_safe(dentry, tmp, &dir->d_subdirs, d_u.d_child) {
201                     spin_lock(&dcache_lock);
                        spin_lock(&dentry->d_lock);
203                     if (!(d_unhashed(dentry)) && dentry->d_inode) {
                                dget_locked(dentry);
205                             __d_drop(dentry);
                                spin_unlock(&dentry->d_lock);
207                             simple_unlink(dir->d_inode, dentry);
                                spin_unlock(&dcache_lock);
209                             dput(dentry);
          } else {
211                             spin_unlock(&dentry->d_lock);
                                spin_unlock(&dcache_lock);
213       }
        }
215     shrink_dcache_parent(dir);
        //mutex_unlock(&dir->d_inode->i_mutex);
217     }

219     static int fpgafs_rmdir(struct inode *dir, struct dentry *dentry)
        {
221             /* remove all entries */
                free_fpga_context(FPGAFS_I(dentry->d_inode)->i_ctx);
223             fpgafs_prune_dir(dentry);
```

```
        //fpgafs_remove_inode

        return simple_rmdir(dir, dentry);
}

/* SUPERBLOCK operations */
int fpgafs_fill_sb(struct super_block *sb, void *data, int silent)
{
  struct inode *inode;
  int ret = -ENOMEM;

        static struct super_operations fpgafs_ops = {
                //.read_inode = fpgafs_read_inode,
                //.put_super  = fpgafs_put_super,
                .alloc_inode = fpgafs_alloc_inode,
    .destroy_inode = fpgafs_destroy_inode,
    .statfs = simple_statfs,
    .delete_inode = fpgafs_delete_inode,
    .drop_inode = generic_delete_inode,
        };

  sb->s_maxbytes = MAX_LFS_FILESIZE;
  sb->s_blocksize = PAGE_CACHE_SIZE;
  sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
  sb->s_magic = FPGAFS_MAGIC;
  sb->s_op = &fpgafs_ops;

        inode = fpgafs_new_inode(sb, S_IFDIR | 0775);
        if (!inode)
                return ret;

  inode->i_op = &fpgafs_dir_inode_operations;
  inode->i_fop = &simple_dir_operations;
  FPGAFS_I(inode)->i_ctx = NULL;

  sb->s_root = d_alloc_root(inode);
  if (!sb->s_root)
                return ret;

  return 0;
}

static int fpgafs_get_sb(struct file_system_type *fs_type,
     int flags, const char *dev_name, void *data, struct vfsmount *mnt)
{
        return get_sb_nodev(fs_type, flags, data, fpgafs_fill_sb, mnt);
}

static struct file_system_type fpgafs_type = {
        .owner   = THIS_MODULE,
        .name    = "fpgafs",
        .get_sb  = fpgafs_get_sb,
        .kill_sb = kill_anon_super
};

static void fpgafs_init_once(void *p, struct kmem_cache * cachep,
                             unsigned long flags)
{
  struct fpgafs_inode_info *fsi = p;
        inode_init_once(&fsi->vfs_inode);
}

/* init exit functions ... */
int __init fpgafs_init(void)
{
        fpgafs_inode_cache = kmem_cache_create("fpgafs_inode_cache",
```

```
                        sizeof(struct fpgafs_inode_info), 0,
291                     SLAB_HWCACHE_ALIGN, fpgafs_init_once, NULL);
        if(!fpgafs_inode_cache)
293             return -ENOMEM;

295     register_filesystem(&fpgafs_type);
        printk("fpgafs: Benjamin Krill <ben@codiert.org>\n");
297     printk("fpgafs: successfull loaded ...\n");
        return 0;
299 }

301 void __exit fpgafs_exit(void)
    {
303     kmem_cache_destroy(fpgafs_inode_cache);
        unregister_filesystem(&fpgafs_type);
305 }

307 module_init(fpgafs_init);
    module_exit(fpgafs_exit);
309
    MODULE_LICENSE("GPL");
311 MODULE_AUTHOR("Benjamin Krill <ben@codiert.org>");
```

Listing A.2: FPGAFS: Main Filesystem Implementation

Low Level Driver Management (llmgmt.c):

```
1   /*********************************************
     * Benjamin Krill <ben@codiert.org>
3    *********************************************/
    #include <linux/pagemap.h>
5   #include <linux/kernel.h>
    #include <linux/module.h>
7   #include <linux/init.h>
    #include <linux/mm.h>
9   #include "fpgafs.h"

11  #define FPGAFS_MAX_LLDRV 3

13  static struct fpgafs_lldrv *lldrv[FPGAFS_MAX_LLDRV];
    static struct fpgafs_lldrv *lldrv_cur;
15  static int lldrv_count = 0x0;
    static DEFINE_SPINLOCK(fpgafs_lldrv_lock);
17
    struct fpga_context* alloc_fpga_context(void)
19  {
            struct fpga_context *ctx;
21          ctx = kzalloc(sizeof *ctx, GFP_KERNEL);
            if (!ctx)
23                  goto out;

25          /* initialize the struct*/
            ctx->lldrv = -1;
27
    out:
29          return ctx;
    }
31
    void free_fpga_context(struct fpga_context *ctx)
33  {
            if (ctx->load_buf)
35                  kfree(ctx->load_buf);
```

```
37          kfree(ctx);
            return;
39  }

41  ssize_t fpgafs_send_data(struct file *file, const char __user *buf,
                                size_t len, loff_t *pos)
43  {
            struct fpga_context *fcur = (struct fpga_context*)file->private_data;
45          return (fcur->lldrv > -1) ?
                    lldrv[fcur->lldrv]->send(fcur, buf, len)
47                  : -EBUSY;
    }
49  //EXPORT_SYMBOL_GPL(fpgafs_send_data);

51  ssize_t fpgafs_recv_data(struct file *file, char __user *buf,
                                size_t len, loff_t *pos)
53  {
            struct fpga_context *fcur = (struct fpga_context*)file->private_data;
55          return (fcur->lldrv > -1) ?
                    lldrv[fcur->lldrv]->recv(fcur, buf, len)
57                  : -EBUSY;
    }
59  //EXPORT_SYMBOL_GPL(fpgafs_recv_data);

61  ssize_t fpgafs_write_load(struct file *file, const char __user *buf,
                                size_t len, loff_t *pos)
63  {
            struct fpga_context *fcur = (struct fpga_context*)file->private_data;
65          return (fcur->lldrv > -1) ?
                    lldrv[fcur->lldrv]->write_load(fcur, buf, len)
67                  : -EBUSY;
    }
69  //EXPORT_SYMBOL_GPL(fpgafs_write_load);

71  ssize_t fpgafs_read_load(struct file *file, char __user *buf,
                                size_t len, loff_t *pos)
73  {
            struct fpga_context *fcur = (struct fpga_context*)file->private_data;
75          return (fcur->lldrv > -1) ?
                    lldrv[fcur->lldrv]->read_load(fcur, buf, len)
77                  : -EBUSY;
    }
79  //EXPORT_SYMBOL_GPL(fpgafs_read_load);

81  /* set/get current low level driver */
    ssize_t fpgafs_read_lldrv(struct file *file, char __user *buf,
83                              size_t len, loff_t *pos)
    {
85          struct fpga_context *fcur = (struct fpga_context*)file->private_data;
            size_t l = (len > 5)?5:len;
87          if (fcur->lldrv > -1) {
                    if (copy_to_user(buf, lldrv[fcur->lldrv]->name, l))
89                          return -EFAULT;
            } else {
91                  return -EBUSY;
            }
93          return l;
    }
95  //EXPORT_SYMBOL_GPL(fpgafs_read_lldrv);

97  ssize_t fpgafs_write_lldrv(struct file *file, const char __user *buf,
                                size_t len, loff_t *pos)
99  {
            struct fpga_context *fcur = (struct fpga_context*)file->private_data;
101         u32 cp = 0, i;
            u8 __user *usr;
```

```
103         unsigned char tmp[5];
            size_t l = (len > 5)?5:len;
105
            /* get name */
107         while (cp < l) {
                    usr = (u8*)&buf[cp];
109                 if (__get_user(tmp[cp], usr))
                            return -EFAULT;
111                 cp++;
            }
113
            for(i=0; i < FPGAFS_MAX_LLDRV; i++)
115                 if (lldrv[i] != NULL) {
                            for (cp = 0; cp < l; cp++) {
117                                 if (lldrv[i]->name[cp] != tmp[cp])
                                            break;
119                         }
                            if (cp == l) {
121                                 fcur->lldrv = i;
                                    return l;
123                         }
                    }
125
            fcur->lldrv = -1;
127         return -1;
    }
129 //EXPORT_SYMBOL_GPL(fpgafs_write_lldrv);

131 ssize_t fpgafs_read_stat(struct file *file, char __user *buf,
                             size_t len, loff_t *pos)
133 {
            struct fpga_context *fcur = (struct fpga_context*)file->private_data;
135         return (fcur->lldrv > -1) ?
                    lldrv[fcur->lldrv]->stat(fcur, buf, len)
137                 : -EBUSY;
    }
139
    ssize_t fpgafs_write_cmd(struct file *file, const char __user *buf,
141                          size_t len, loff_t *pos)
    {
143         struct fpga_context *fcur = (struct fpga_context*)file->private_data;
            return (fcur->lldrv > -1) ?
145                 lldrv[fcur->lldrv]->cmd(fcur, buf, len)
                    : -EBUSY;
147 }

149 /* low level un-/register functions */
    int fpgafs_register_lldrv(struct fpgafs_lldrv *drv)
151 {
            unsigned long flags;
153         int i;

155         spin_lock_irqsave(&fpgafs_lldrv_lock, flags);
            if (lldrv_count == FPGAFS_MAX_LLDRV)
157                 return -EBUSY;

159         /* find free space */
            for(i=0; i < FPGAFS_MAX_LLDRV; i++)
161                 if (lldrv[i] == NULL) {
                            lldrv[i] = drv;
163                         break;
                    }
165
            if (lldrv[i]->init)
167                 lldrv[i]->init();
```

```
169         lldrv_cur = lldrv[i];

171         lldrv_count++;

173         spin_unlock_irqrestore(&fpgafs_lldrv_lock, flags);
            return 0;
175 }
    EXPORT_SYMBOL_GPL(fpgafs_register_lldrv);
177
    int fpgafs_unregister_lldrv(struct fpgafs_lldrv *drv)
179 {
            unsigned long flags;
181         int i,k;

183         spin_lock_irqsave(&fpgafs_lldrv_lock, flags);

185         for(i=0; i < FPGAFS_MAX_LLDRV; i++) {
                    if (lldrv[i] == drv) {
187
                            /* call the exit function */
189                         if (lldrv[i]->exit)
                                    lldrv[i]->exit();
191
                            lldrv[i] = NULL;
193
                            /* if current, search another low level driver */
195                         if (lldrv_cur == drv) {
                                    lldrv_cur = NULL;
197                                 for(k=0; k < FPGAFS_MAX_LLDRV; k++)
                                            if (lldrv[k]) {
199                                             lldrv_cur = lldrv[k];
                                                break;
201                                         }
                            }
203                         break;
                    }
205         }

207         lldrv_count--;

209         spin_unlock_irqrestore(&fpgafs_lldrv_lock, flags);
            return 0;
211 }
    EXPORT_SYMBOL_GPL(fpgafs_unregister_lldrv);
```

Listing A.3: FPGAFS: Low Level Driver Management