Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Olivier Salama

High-Performance Computing mit TIKDIMM

Diplomarbeit DA-2004-06
Wintersemester 2003/2004

Betreuer: Christian Plessl

Verantwortlicher:
Prof. Dr. Lothar Thiele

10.3.2004

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Wintersemester 2003/2004

DIPLOMARBEIT

for

Herrn Olivier Salama (D-ITET)

Main Reader: Christian Plessl

Issue Date: 3. November 2003
Submission Date: 10. März 2004

High-Performance Computing mit TIKDIMM

Contents

1 Problem Task **6**
1.1 Background . . . 6
1.2 Problem Task . . . 7
1.3 Tasks . . . 7
1.4 Organization . . . 8

2 Introduction **9**
2.1 Hardware . . . 9
2.2 Software Components . . . 9
2.3 Acknowledgements . . . 9

3 Architecture **10**
3.1 Hardware Summary . . . 10
3.2 System Overview . . . 10
3.3 Device Driver . . . 13
3.4 TKDM Library . . . 13
3.4.1 Functional Principle . . . 13
3.4.2 Job Processing . . . 14

4 Detailed Implementation **17**
4.1 Firmware . . . 17
4.1.1 Page Zero Controller . . . 17
4.1.2 Memory Backend . . . 17
4.1.3 SelectMAP Controller . . . 18
4.1.4 Firmware version 7 . . . 18
4.1.5 Latest Firmware version . . . 20
4.1.6 Register Block . . . 20
4.2 Target FPGA . . . 23
4.2.1 Connection to the SDRAM . . . 23
4.3 The GNU / Linux device driver . . . 23
4.3.1 Kernel - Mode Programming Interface TKM . . . 31
4.3.2 System Calls . . . 34
4.3.3 ioctl . . . 34
4.3.4 read . . . 36
4.3.5 write . . . 37
4.3.6 Usage of the Driver, an example . . . 37
4.4 Library API . . . 38
4.4.1 Initialisation . . . 38
4.4.2 Main functions . . . 38
4.4.3 More Methods . . . 39
4.4.4 Length of the output data . . . 40
4.4.5 Functions related to the Target FPGA . . . 40
4.5 Reference Application . . . 41
4.6 Service module . . . 43

5 Status and Future Work **45**
5.1 Status 45
5.1.1 Initial firmware and driver version 45
5.1.2 Latest firmware and driver version 45
5.1.3 TKDM Library 45
5.2 Future Work 45

A Recovering from an oop's error **47**

B Benchmark **47**

List of Figures

1	The TIKDIMM expansion module	10
2	the connection of the FIFOs in the Abstraction Layer FPGA.	11
3	Architecture of the Tikdimm	12
4	Environment of the driver	13
5	Data flow of a job	14
6	Write process of the *Wait()* function	15
7	Read process of the *Wait()* function	16
8	Firmware Units	17
9	Demultiplexer for the SDRAM commands	22
10	The main state machine of the memory controller	24
11	Read state machine of the memory controller	25
12	Write state machine of the memory controller	26
13	Refresh state machine of the memory controller	27
14	Structural connection of the four memory controller FSMs.	28
15	Schematic of the Target FPGA top level entity	29

List of Tables

1	Virtuall address space	12
2	Writable registers	20
3	The Target general-purpose registers configuration register	20
4	Readable registers	21
5	Value and meaning of the bus switch control byte.	35
6	Ioctl commans for the service module	43

Abstract

The TKDM module is a PC expansion card for reconfigurable computing applications. It is FPGA-based and provides on-board memory. The card is realised as DIMM module.

Reconfigurable computing takes advantage of the combined strengths of hardware and software. The computationally-intensive part of an application is migrated to the reconfigurable hardware.

This thesis presents a toolkit for the TKDM platform. It includes a driver and a library to write user space applications in a simple way. Furthermore, a design template for the FPGA which holds the reconfigurable processing unit is presented. The TKDM module firmware is discussed.

After an introduction on the background of the TKDM module and the motivation of this thesis, the hardware architecture is summarised. The functional principal of the library is explained.

A further part gives an overview of the firmware units and their function. The initial version of the firmware has caused some problems. They concern the memory controller. One solution which is discussed restricts the refresh abilitie of the memory controller. This makes it impossible to write an efficient driver. An improved memory controller is presented that doesn't limit the driver.

A function reference of the driver API is given. The library interface is discused in detail. Finally reference application demonstrates the functionality of the TKDM library.

Zusammenfassung

Das TKDM Modul ist eine PC-Erweiterungskarte für Reconfigurable Computing Applikationen. Sie ist FPGA basiert und bietet eingebauten Speicher. Die Karte ist als DIMM Modul realisiert.

Reconfigurable Computing bezieht seine Vorteile aus der kombinierten Stärke von Hard- und Software. Der rechenintensieve Teil der Applikation wird in die reconfigurierbare Hardware migriert.

Diese Diplomarbeit präsentiert ein Toolkit für die TKDM Plattform. Dieses umfasst einen Treiber und eine Bibliothek, die es ermöglicht Applikationen auf einfache Weise zu programmieren. Des Weiteren wird ein design Template für das FPAG, das die reconfigurable computing Applikation implementiert, besprochen. Die TKDM firmware wird vorgestellt.

Nach einer Einführung zum Hintergrund des TKDM Moduls und der Motivation dieser Arbeit, wird eine Zusammenfassung der Hardware Architektur gegeben. Das Funktionsprinzip der Bibliothek wird erläutert.

Ein weiterer Teil gibt eine Übersicht der Firmware Teile und deren Funktion. Die ursprüngliche Version der Firmware hatte einige Schwierigkeiten mit dem Memory Controller. Eine Lösung, die besprochen wird, schränkt die Refresh Fähigkeit ein. Das macht es unmöglich einen effizienten Treiber zu schreiben. Danach wird ein verbeserter Memory Controller vorgestellt, der den Treiber nicht einschränkt.

Es gibt eine Funktionsreference der Trreiber API. Die Funktionen der Bibliothek werden im Detail erklärt. Schlussendlich wird die Funktionalität der Bibliothek anhand einer Referenz Applikation demonstriert.

1 Problem Task

This section describes the problem task for my diploma thesis.

1.1 Background

The basic idea of Reconfigurable Computing is to use reconfigurable hardware elements for the acceleration of compute intensive algorithms. Often it is not feasible or simply inefficient to map complete applications to the reconfigurable device. Thus, frequently a system that combines a general purpose CPU and a reconfigurable device is used. Typically, the sequential and control-flow dominated parts of applications are executed on the CPU, whereas the data-flow oriented parts of the application are mapped to the reconfigurable device. The reconfigurable device serves as an application specific co-processor, that is reconfigured on-demand.

Reconfigurable computing systems have been studied for several years and many research systems have been built. Most of these systems are attached to or integrated in a PC. Most high-end systems use PCI extension cards that provide reconfigurable devices—usually FPGAs—and interface logic. Low-end systems use other, slower interfaces like USB, parallel or serial ports. When building custom reconfigurable computing systems, reconfigurable devices are usually attached to the system memory or IO-bus.

Since the the gap between CPU core speed, the memory subsystem speed and IO bus speed is getting larger, attaching reconfigurable units to a peripheral bus like PCI is getting less attractive, due to high latencies and slow speed in comparison with the CPU core frequency. This limits the application of reconfigurable computing systems attached to IO-busses to applications that show moderate sensitivity to communication latency and restricted bandwith constraints.

For applications that require a closer interaction of CPU and the reconfigurable device another option for coupling is desireable. An interessting idea is to use the system memory bus for that purpose. As main memory access speed and latency is a key issue in the design of fast computing systems, all CPUs have advanced, fast memory interfaces. Attaching reconfigurable devices to the memory bus of the CPU has the potential of delivering much lower communication latency and higher bandwidth compared to PCI attached solutions.

The idea of integrating reconfigurable logic in standard PCs by using memory modules has emerged only recently. To our knowledge there is only one research group that has persued this idea in depth: the group of Philipp Leong has developed the *Pilchard* system [3]. Pilchard is a 133 MHz DIMM SDRAM compatible module that uses a Virtex-300 FPGA. Pilchard does not provide on-board RAM and is configured via an attached JTAG programmer. Pilchard has been used in a large number of projects [8][1][2].

In a previous diploma thesis conducted by Andreas Schweizer [7] an FPGA module that attaches to the DIMM bus called *TKDM* (pronounce: tik-dimm) has been built. TKDM provides several architectural and technological enhancements over Pilchard:

- TKDM uses the latest Xilinx Virtex-II FPGA technology
- TKDM features 64MB on-board SDRAM to provide sufficient memory for data-buffers
- TKDM can be configured and powered over the DIMM memory bus, thus no external connectors are needed
- TKDM provides a firmware layer, that facilitates the development of applications by abstracting many system integration details

The TKDM hardware design and an initial TKDM firmware has been completed in the diploma thesis by Andreas Schweizer. The hardware design is fully debugged and tested and the basic firmware and driver functions have been implemented.

Meanwhile, the driver and the firmware have been extended such that a proof-of-concept system is running now. We have published a paper on the TKDM project and its current status that will be published at the FPT'03 conference this year in December [5].

1.2 Problem Task

The goal of this thesis is to build on this proof-of-concept demonstrator developed in the previous work. While the existing demonstrator has shown, that the TKDM system is working there are three open issues, that shall be tackled in this work:

1. **TKDM toolkit:** Overall, the driver and firmware framework is working, but a careful restructuring will make the framework much easier to use. To facilitate the implementation of a real application a TKDM toolkit is needed. The toolkit shall provides both, a useful driver API and user libraries for TKDM access, as well as reconfigurable IP cores for the implementation of application specific co-processors on the FPGA.

2. **Application:** The use of the the TKDM toolkit shall be demonstrated with a real compute intensive application. This application running on the host CPU and TKDM shall show a clear speedup when compared to a CPU only solution. Possible fields of application are hardware-accalerated filters for image-processing, video decoding, data compression or hashcode algorithms.

3. **Performance:** There are some unresolved communication performance issues that result from the interaction of the chipset (northbridge controller) of the PC's Pentium 3 mainboard with the TKDM board. The problem arises due to the deactivation of caching for the memory area used by TKDM. We are confident, that switching to a Pentium 4 based mainboard can help here, due to thee finer degree of cache control available in Pentium 4 CPUs.

1.3 Tasks

1. **Background FPGAs and tools**

   Make yourself familiar with Xilinx Virtex-II FPGAs. The Virtex-II data-sheet [9] is a good starting point. Make yourself familiar with the Xilinx ISE development tools for FPGA design. Recently our department aquired a number of licenses for Synplify Pro, a state-of-the-art FPGA synthesis tool. It makes sense to switch to Synplify for synthesis early in the project.

2. **TKDM board**

   Read the documentation on TKDM as provided in the thesis report [7] and in the FPT'03 paper. In particular, make sure that you have a good understanding how the firmware is working. This is the area where you will work in. Make yourself familiar with the implementation (VHDL code) of the firmware and with the Linux device driver for TKDM.

3. **Implement application withing current firmware**

   Start with the implementation of an application that needs only small modifications to the current firmware and drivers. The experience from this implementation will help to define a good TKDM toolkit.

4. **Definition and implementation of the TKDM toolkit**

   Define a TKDM toolkit based on your experience with the current firmware, drivers and the implementation of the demo application. Write a design-document that describes the interfaces provided by the toolkit in detail. Implement the TKDM toolkit according to the specification.

5. **Implement application**

   Find a suitable application that can be implemented on the TKDM system. The application should have one or several compute intensive kernels that can be mapped to the TKDM board. The resulting application should show a significant speedup when compared to the CPU only system.

6. **Port to Pentium 4 mainboard**

   In the previous Diploma thesis it has been found that the communication performance of TKDM in a Pentium 3 based system is limited because of problems with the mainboards chipset. Initial tests for a different, Pentium 4 based mainboard, have shown that the performance can be largely increased.

   Port the TKDM firmware to this mainboard. The mainboard uses the same DIMM memory-bus protocol but runs at 133MHz instead of 100MHz and uses a different address mapping. The current version of the firmware is slightly to slow to run at 133MHz

1.4 Organization

- **Schedule**

  Make a realistic schedule for your project at the beginning of your work and discuss it with your advisor. Define reasonable milestones and keep track of the work progress underway.

- **Meetings**

  Fix a time for a weekly meeting with your advisor. Prepare for this meeting and present a short summary of the current state of your work and a plan for the next steps. Current problems shall be discussed. Additional meetings will be held on demand.

- **Documentation**

  Please keep in mind that the thesis report is considered as a *major* part of your work. Take your time to do a careful documentation. A thesis report is a scientific report. This implies the use of a specific language and methodology.

  The main goal of the thesis report is to explain to the reader these points:

  - What is the problem?
  - What has been done by others so far?
  - What are the design alternatives?
  - Why did you chose your design?
  - How does your design work?

  Of course not every detail can be discussed in great detail, but it should be possible to get a good idea of how things work without going to look into the code oneself.

2 Introduction

2.1 Hardware

The TKDM module was designed in a previous diploma thesis conducted by Andreas Schwiezer ([7]). It is a PC expansion card for the DIMM bus and provides a FPGA, which acts as application specific co-processor.

2.2 Software Components

The thesis presents the TKDM library. It is suitable for writing applications which wants to use the TKDM module. It provides an easy to use interface. The different steps which are necessary to use the module are hidden from the application.

The driver which supports the library is presented. It is a Linus kernel module. Its interface to the user space is discussed in detail. The reference Application demonstrates the usage of the TKDM library. It also serves as template.

The TKDM module is run by the firmware. The thesis gives an overview of this vhdl design. Especially the parts, which caused problems are explained in great detail.

2.3 Acknowledgements

In spite of the enormous lack of time during the last weeks of the work, it was a lot of fun. It was very interesting to see the interactions of the firmware, driver and user space application.

I wish to express my most sincere thanks to Christian Plessl who was my supervisor for this work. He was always helpful and provided new ideas. I wish to thank Prof. Lothar Thiele for being my supervising professor. Thanks to the institute, TIK, for providing the working environment.

3 Architecture

3.1 Hardware Summary

TIKDIMM is an expansion card for the DIMM slot. It has four 16MB SDRAM blocks on-board. The Abstraction Layer FPGA (Xilinx Virtex-II) implements the firmware. It controls memory transfers from and to the on-board memory and comunicates through the DIMM bus with the northbridge controller of the main board. The Target FPGA is reserved for the reconfigurable computing application. Figure 1 shows the essential hardware components and their interaction regarding the data flow. The on-board memory consists of four 16MB SDRAM blocks. Two of them, the X-side SDRAM, is connected over an bus exchange switch to the Target FPGA and the Abstraction Layer. The Y-side is constructed analogously. A detailed hardware description is given in [7].

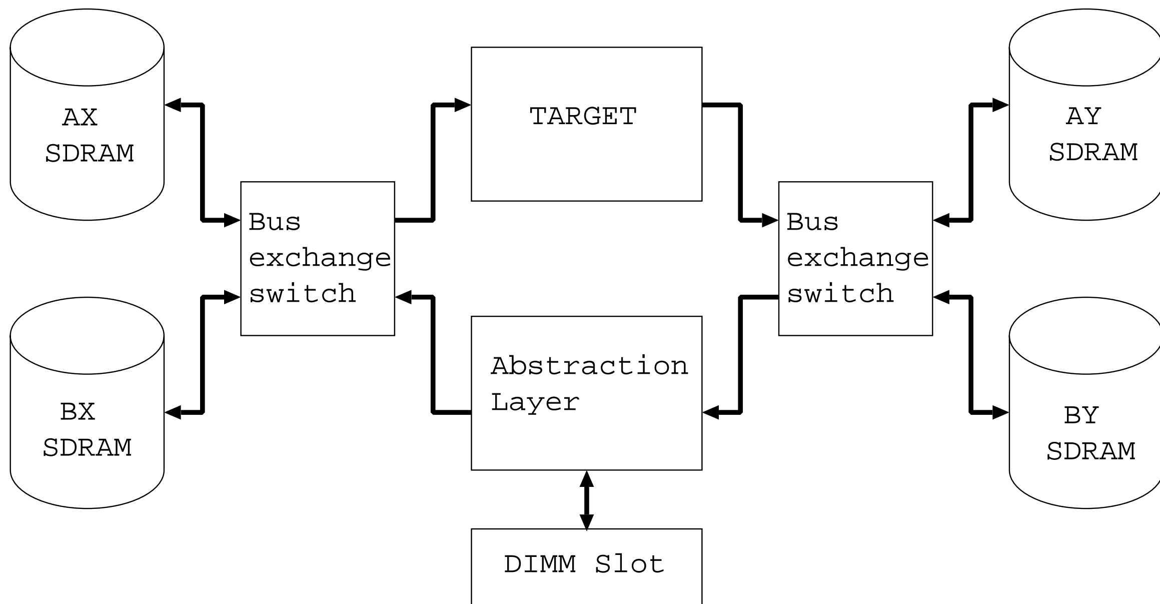

Figure 1: The TIKDIMM expansion module

3.2 System Overview

The hardware of the TIKDIMM platform is connected to the PC through the DIMM bus. The card is controlled over a register block which is mapped to the PC's virtual address space. Figure 3 illustrates the data flow between the DIMM bus and the on-board SDRAMs. There are four on-board memory blocks AX, BX, AY and BY. Each one has its own memory controller and a FIFO[1] buffer for writing and one for reading operations.

There are two bus exchange switches, one for the X-side and one for the Y-side. They switch the data bus of the SDRAM blocks. Each has two positions plain and crossover. The control buses of all four SDRAMs are connected to the Abstraction Layer FPGA.

In position plain the AX SDRAM data bus is connected to the Target FPGA and the BX SDRAM to the Abstraction Layer. This FPGA implements a switch which connects the BX SDRAM to the assigned

[1] First in first out buffer

pair of FIFOs. Depending whether it is a read or write operation one of the FIFOs is used. Figure 2 illustrates the connection of the FIFOs to the data bus that leads to the bus exchange switch. This components (exept the bus exchange switch) are implemented through the Abstraction Layer FPGA. The position of the switch with the lable "switch" is always the same as the bus exchange switch.

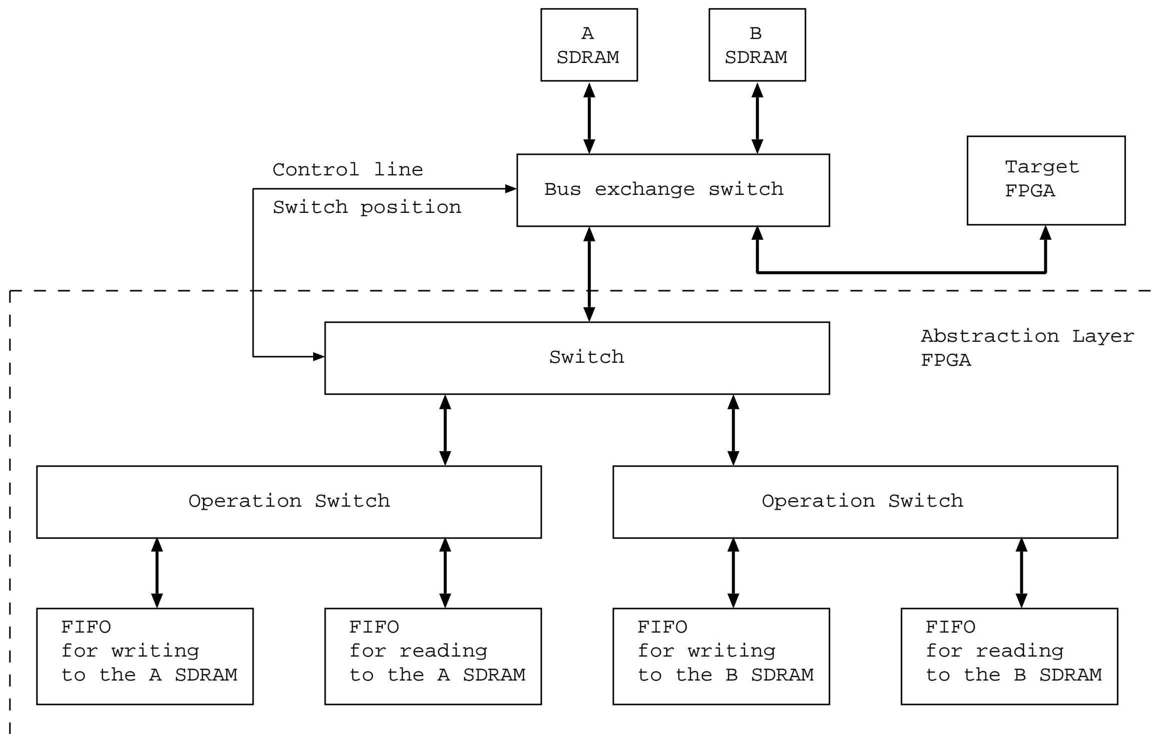

Figure 2: the connection of the FIFOs in the Abstraction Layer FPGA.

If the bus exchange switch is set to crossover the BX SDRAM is connected to the Target FPGA nad the AX SDRAM to the Abstraction Layer. The Y-side is constructed the same way.

In the figure 3 the FIFO buffer for writing and the FIFO for reading operations are displayed as single block with the lable "FIFO AX" etc. The block "Bus Switch" represents the physical bus exchange switch and the "switch" of figure 2.

The bus exchange switch connect either the AX FIFO to the AX SDRAM or the BX FIFO to the BX SDRAM. So for a certain position of the bus switch only one of the FIFO buffers of the X-side is in use. Therefore, it's not reasonable to access the other. The SDRAM block that is assigned to the unconnected FIFO is connected to the Target FPGA. The same applies to the Y-side.

The input port of the FIFO for writing and the output port of the FIFO for reading operations share the same address. Table 1 gives the address offset relative to the TIKDIMM module base in the virtuall address space. Besides the FIFO ports the address of the register block and the Block SelectRAM is shown. The latter is part of the Abstraction Layer FPGA. It is used for debugging. A DIMM module has four banks. The third column shows the bank number which is selected to access the FIFO port, register block or the Block SelectRAM.

The register block is used by the driver to communicate with the firmware. See section 4.1.6 for a

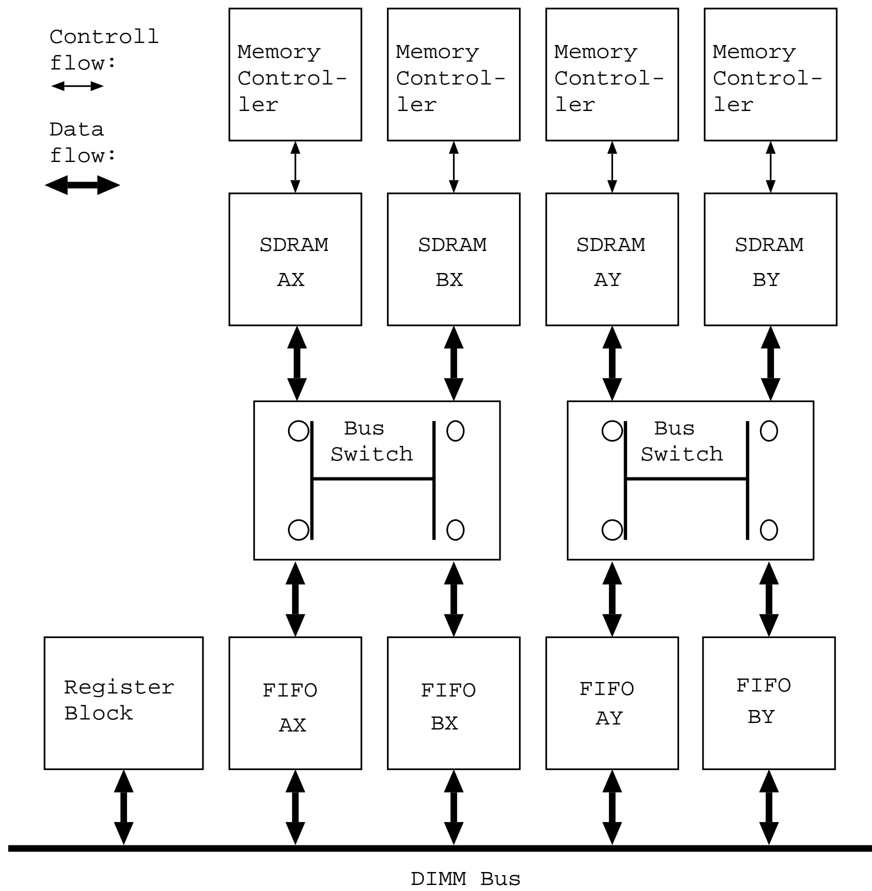

Figure 3: Architecture of the Tikdimm

	Address offset	bank #
Register Block	0x0	0
AX	0xc00	1
BX	0x1400	2
AY	0x800	1
BY	0x1000	2
Block SelectRAM	0x1800	3

Table 1: Virtuall address space

description.

3.3 Device Driver

The driver is the connection between the user space application and the hardware. In Linux the drivers are kernel modules. To the userspace the standard system calls are provided. Other kernel modules can use the driver's functionality through the kernel-mode programming interface. Figure 4 shows the environment of the driver. For a detailed description see section 4. The driver can communicate with other kernel modules through the TKM interface.

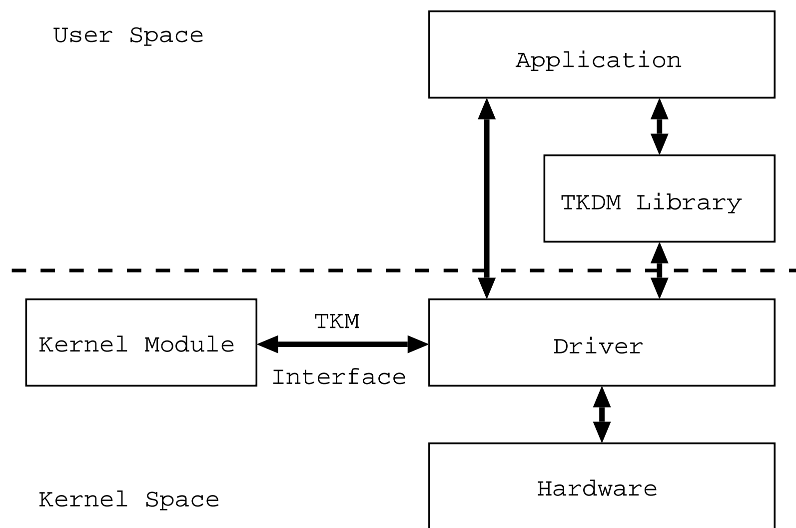

Figure 4: Environment of the driver

3.4 TKDM Library

The TKDM library provides an interface to the TIKDIMM platform for applications running in the user space. The TIKDIMM driver only provides simple operations, such as setting up a transaction, turn the bus switch etc. The user must take care of the correctness of the command. He must ensure that the right SDRAM blocks are connected to the Target if he invokes a Target operation. The library hides this procedures from the application. The source files *tkdm_lib.C* and *tkdm_lib.h* are in the directory *toolkit/tkdm_lib/*.

3.4.1 Functional Principle

The library allows the user to define a job. This is a data structure with a buffer that holds the input data for the Target FPGA. A second buffer takes the output from the TIKDIMM. The data of the job is split into packets. These are small enough to be copied to a single on-board SDRAM block. So a packet is smaller then the size of a SDRAM block (16MB). The jobs are stored in a queue. The library calls a user defined function, when the output buffer overflows.

```
class TKDM_JOB
{

    ...

    unsigned char * ptInBuf;     // input buffer in PC RAM
    unsigned char * ptOutBuf;    // output buffer in PC RAM
    size_t len;                  // number of bytes in the job
    size_t outBufSize;           // size in bytes of the output buffer
    unsigned int inPacketSize;   //Size of the packets on the X-side
    unsigned int outPacketSize;  //Size of the packets on the Y-side

    ...

};
```

3.4.2 Job Processing

The *Wait* function is the heart of the TKDM library. Its task is to perform the job. It takes the first job from the queue and returns when this job is done. Figure 5 illustrates the data flow. The storage with the caption "X Side SDRAMs" represents the AX and the BX SDRAM. The same applies to the Y-side storage. "Job Data" holds input data as well as processed data of the job.

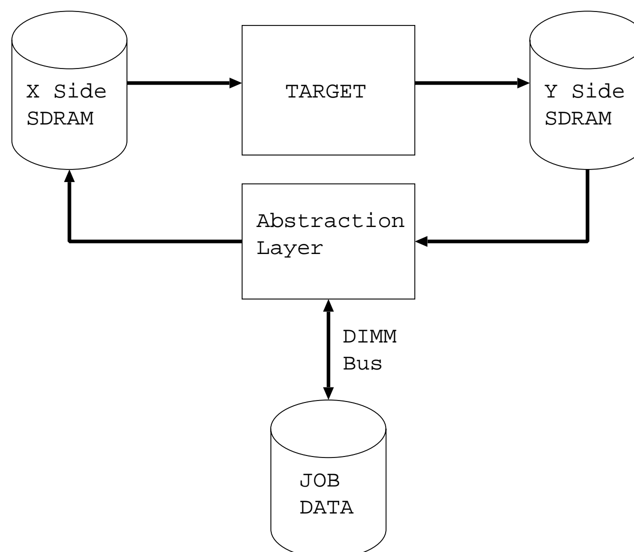

Figure 5: Data flow of a job

The input of the job is written to the X-side SDRAMs, where the Target gets its input. If the bus exchange switch is set to plain the Target uses the AX block while new data is written to the BX SDRAM. In position crossover the library writes to the AX SDRAM and the Target reads from the BX SDRAM.

The Target can start processing after the first packet was written to the X-side and the bus switch was turned. Meanwhile the next packet can be written.

The Target stores its output in the Y-side SDRAMs. In parallel the library reads from this side. This procedure has a similar restriction than the one on the X-side. The Target writes a packet to AY while the packet in BY can be read back. So the read back of the last packet can start not until the Target is done.

Since the write transfer to the TIKDIMM and the read back operation use partly the same channel the DIMM bus, only one of them can be performed at the same time.

The library is anxious to keep the Target busy while it reads or writes to the TIKDIMM. This is achieved by choosing the packet size so that the job contains more than one packet.

Writing As discussed above the *Wait* function takes care of writing the packets to the module. It must set the bus switch and setup the memory transfers between the on-board SDRAMs of the X-side and the concerned FIFO buffers. These are the two FIFOs for writing of the Abstraction Layer assigned to the AX and BX SDRAM and the input FIFO of the Target. In figure 6 this process is described as finite state machine.

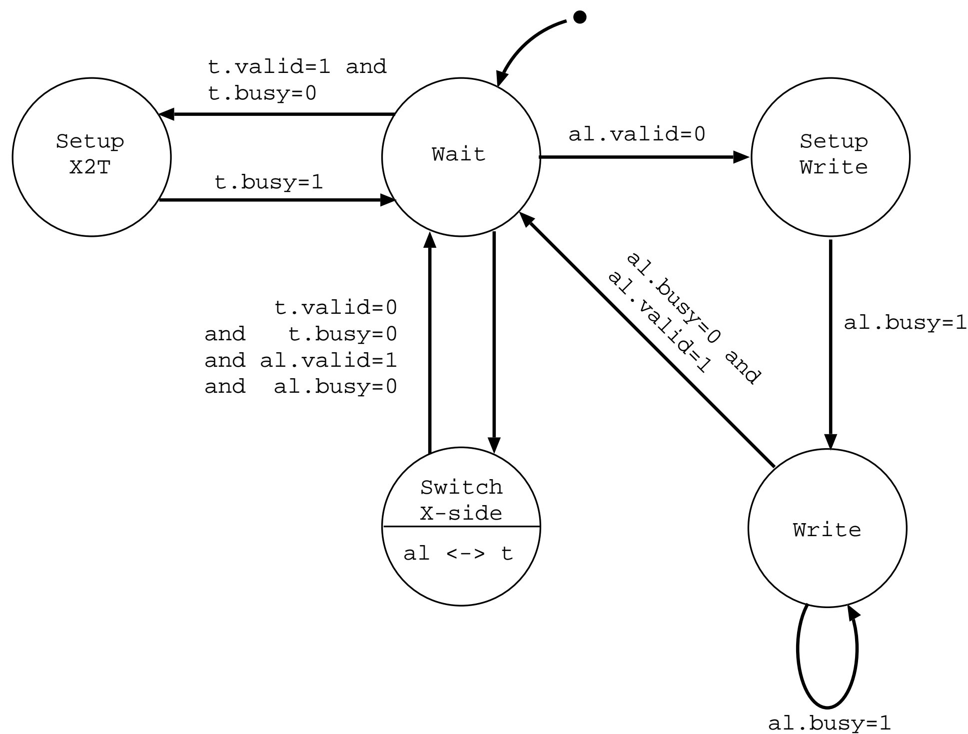

Figure 6: Write process of the *Wait()* function

The signals with the prefix "t." concern the SDRAM block currently connected to the Target FPGA, "al." means the Abstraction Layer. A block is valid if it holds a packet that is ready to be processed by the Target. The SDRAM block is marked as busy when a transfer is pending.

The FSM starts in the "Wait" state. If the SDRAM block connected to the Abstraction Layer is empty ("al.valid=0") the machine goes to "Setup Write" and initialises a write transfer, with the size of a packet, to the TIKDIMM. Now "al.busy" is 1. The state machine remains in "Write" until the packet is completely written to the TIKDIMM, "al.valid" is then 1 and "al.busy" is 0 again. If the Target is connected to a SDRAM block which contains valid data ("t.valid=1") and "t.busy" is 0, the state machine can switch

to "Setup X2T". A data transfer to the Target FPGA is started. The machine goes back to "Wait" with "t.busy" set to 1. The variable keeps this value as long as the Target reads data. When "t.busy" is set to 0, "t.valid" is also set to 0. In the "Switch X-side" state the variables beginning with "t." are exchanged with "al.". The bus switch is turned. If all packets of the job are written the state machine stops.

Reading The *Wait* function also reads back the output of the Target. The state diagram shown in figure 7 describes this operation. It is very similar to the writing process, but the packet size may differ.

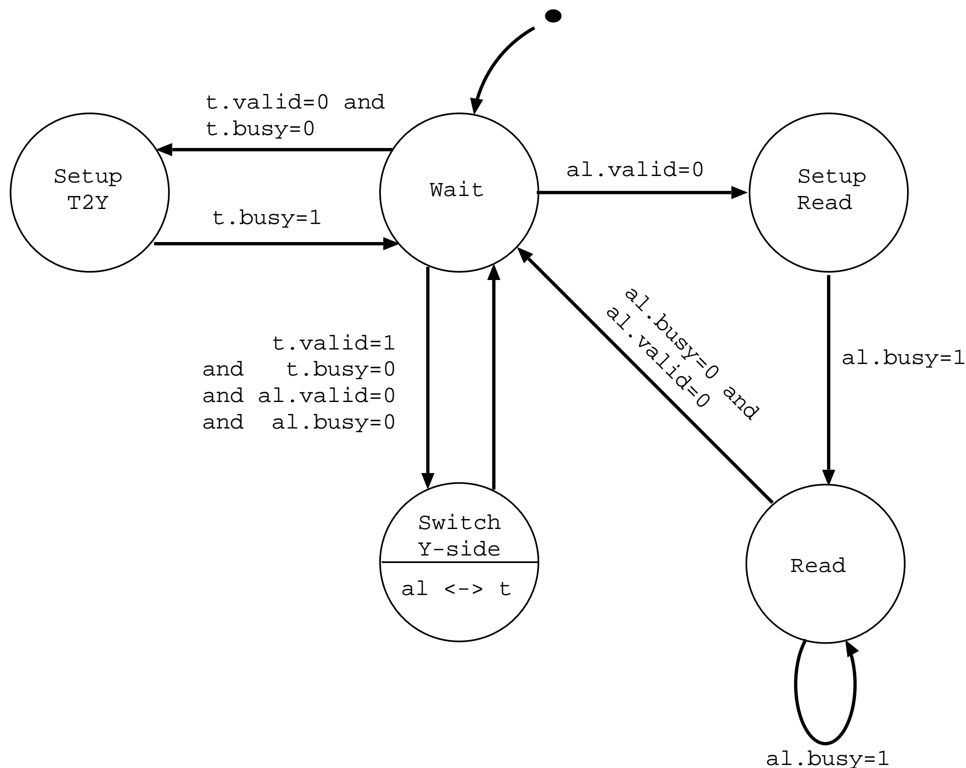

Figure 7: Read process of the *Wait()* function

The machine starts after the Target has started its first read transfer. If "t.valid" and "t.busy" are 0 the FSM goes to "Setup T2Y". This starts a read transfer from the Target. "t.busy" is now 1 and the state machine goes back to "Wait". When the transfer is done "t.busy" is 0 and "t.valid" is 1. If only "t.valid" is 1, the FSM enters "Switch Y-side". Here the variables are exchanged as described above and the Y-side bus switch is turned. In "Setup Read" a read transfer from the TIKDIMM is started. The function read a packet of output data while the FSM is in "Read".

The exit condition of this state machine is tricky. Two different situations are possible depending on the Target design. If input and output length of data have a known, fixed relation the library can determine when the Target is done. But if it's impossible to predict the output length (e.g. data compression) the library waits till the read transfer from the Target times out. Then it assumes that the Target is done.

4 Detailed Implementation

4.1 Firmware

The Firmware is the operating system of the TIKDIMM board. It resides on the Abstraction Layer FPGA. At power up the firmware is loaded from the on-board PROM into the FPGA. It is structured in page zero controller, memory backend and the SelectMAP controller, they are explained below. The parts which are of relevance for this report are discussed in more details. Figure 8 is an overview of the firmware units.

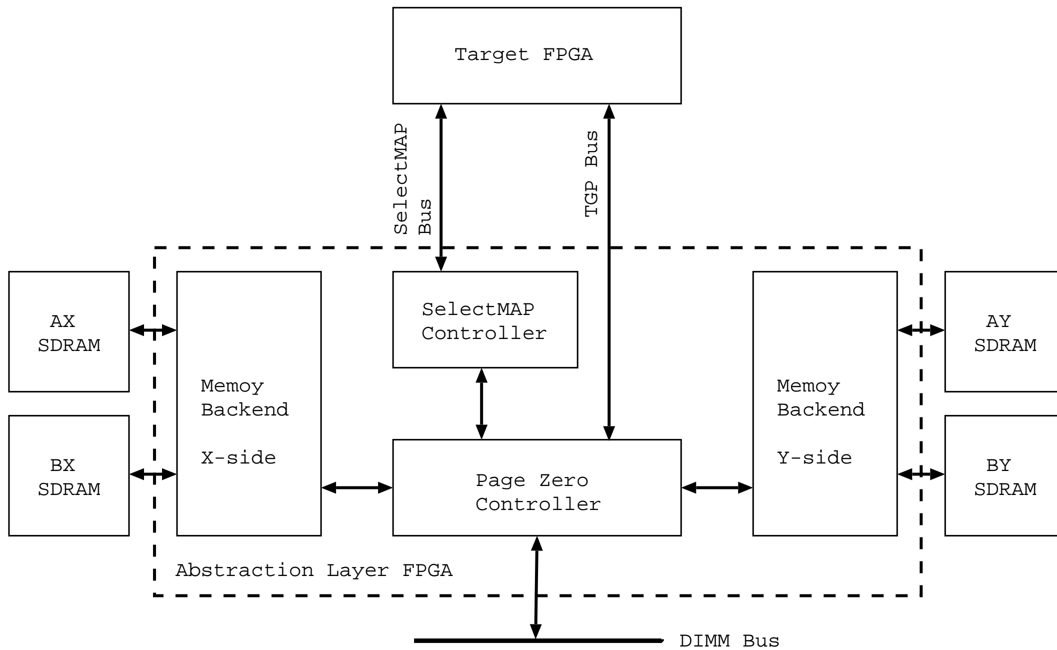

Figure 8: Firmware Units

4.1.1 Page Zero Controller

The page zero controller is concerned with the communication to the PC. The module appears to the PC as DIMM module with four 2 kB pages. The first 32 bytes of the page zero are mapped to a register block (see section 4.1.6). The second and third page contain the entry points for the FIFOs. For debugging 2 kB SelectRAM are mapped to the fourth page (see table 1). The TGP-Bus is the connection to the Target FPGA. The bus is used to access the Target registers. Furthermore it includes controll lines to increment the FIFO buffers of the Target design. The status signals input buffer full and output buffer empty are also part of the bus.

4.1.2 Memory Backend

The memory backend is assigned to a pair of memory blocks. There are two backends one for the blocks on the X-side (AX, BX) and one for the Y-side. The backend provides for each assigned SDRAM block a memory controller and a FIFO for reading operations and one for writing operations. There is a refresh controller, a word counter and an address generater for each SDRAM block. The memory controller

generates the necessary commands for the SDRAMs.

The word counter holds the number of 64-bit quad-words which the memory controller should copy from the FIFO to the SDRAM or vice versa. It is decremented after each copied quad-word. The counter is loaded when the memory controller is in RDPREP or WRPREP (see figures 11 and 12). A transfer is complete when the word counter is zero.

The addres generator is loaded at the same time as the word counter with the start address of the transfer. It calculates the bank, row and column address for the SDRAM access.

Refresh Controller The SDRAM needs 4096 auto refresh commands in 64 μs that makes one every 15.625 μs (see [4]). The refresh controller generates a pulse after 1536 clock cycles and collects them in a counter. Every time an auto refresh command is sent to the SDRAM the refresh controller gets a signal that increments a second counter. If the two counters have a difference of 255 the refresh controller generates a refresh alert. The memory controller starts to send auto refresh commands to the SDRAM. If the two counters of the refresh controller have the same value the refresh controller sends a done signal to the memory controller. This stops the refresh session. The memory controller of the latest firmware version is discussed in more detail in section 4.1.6.

In the discussed firmware versions the auto refresh commands are only generated during a refresh session that means as response to a refresh alert. Therefore, the alert appears after a constant period of time. The clock period of the DIMM bus for the 500 MHz Pentium is 10ns.

$$T_{clk} = 10\ ns$$

$$255 \cdot 1536 \cdot T_{clk} = 3.92\ ms$$

A refresh alert signal has a period of 3.92 ms.

4.1.3 SelectMAP Controller

The SelectMAP controller uses the SelectMAP Bus to write configuration-bitstreams to the Target FPGA. It is also possible to read Target configurations back.

4.1.4 Firmware version 7

This is a debugged version of the firmware which was available at the beginning of the work. It still has some known bugs, which cause serious constraints to the driver's design.

Row boundary bug The SDRAM is organised in banks, rows and columns. To access a memory cell the bank must be opened and the row address specified. The cells of this row are then accessible over the column address. After the transaction the bank must be closed. Only one row per bank can be open at one time.

Read or write operations use sequential addresses. If the row boundary is reached the page must be closed and reopened for the next row.

The memory controller has a problem with the handling of row boundaries, if a refresh alert occurs. Therefore, in version 7, the alert is blocked when the controller is not in IDLE mode. The memory can't be refreshed when the controller performs a read or write operation.

The controller may leave the IDLE state (state diagram on pp. 55 [7]) to perform a memory operation at most for 3.92 ms (period of the refresh alert). The driver must take care that this constraint is met. The following considerations assume that the controller has just finished a refresh session. Therefore, it has 3.92 ms time to read or write.

In case of a read operation, the memory controller reads from the on-board SDRAM right after it has received the read command. Assume there is sufficient space in the FIFO buffer to take all words of the programmed read transfer. This ensures that the controller is not blocked because of a buffer full flag. The period of time ($T_{operation}$) for the whole transfer only depends on the number of bytes (N_{bytes}) requested to read. The TIKDIMM reads 4 bytes per clock cycle.

$$\frac{N_{bytes}}{4} \cdot T_{clk} < T_{operation} = 3.92\ ms$$

$$N_{bytes} < 1.57 \cdot 10^6$$

That is far more than the FIFO can ever hold. The Abstraction Layer FIFO has a capacity of 4kB and the FIFO buffer of the Target is 2kB. So there is no problem with the read operation, provided that no more bytes are requested then the size of the FIFO.

For the write operation the things are different. After the controller has left the IDLE state it waits for the data to arrive. For this reason the driver must take care that the time period between the setup of a write operation and the last data transfer to the TIKDIMM is not more than 3.92 ms. This can be guaranteed when the driver starts the transfer and writes all data during the same call. The kernel module can not be preempted and therefore the worst case execution time of such a call can be predicted.

To keep the Target working the driver must periodically setup new transfers to and from the Target. It is safe to choose the FIFO size as length for the transfer. In this way the controller can not be blocked. The size of the FIFO is 2 kB and the SDRAM block has 16 MB. To transfer a block the driver must setup

$$\frac{16\ MB}{2\ kB} = 8000$$

transfers instead of a single one. The driver must poll the *cmd* or *bst* register to determine when it is possible to setup the next transfer. In the *bst* register the buffer full flag for the read operation and the buffer empty flag of the output buffer is available for the Target. The other two status bits are not connected to the Abstraction Layer FPGA (see figure 15).

If the polling frequency is too low the Target idles and the total execution time of the application is extended. On the other hand if the frequency is too high CPU time is wasted.

The MEMR:cmd register The *cmd* byte is set to start a memory transfer on the TIKDIMM board. If the memory controller isn't in the IDLE state it doesn't accept commands. The driver must assure that its commands are accepted. A refresh session waits 9 clock cycles after each auto refresh command and needs 2 cycles to switch between the states IDLREFR and IDLWREFR. The estimated time for a refresh session is:

$$T_{refr} = 255 \cdot (2 + 9) \cdot T_{clk} = 28.05\ \mu s$$

The probability that a refresh is performed when the driver tries to send a command is approximately:

$$P = \frac{28.05\ \mu s}{3.92\ ms} = 7.16 \cdot 10^{-3} \approx 0.7\ \%$$

There is a probability of $0.7\ \%$ that a transfered command has no effect. The driver is forced to check if its commands are successful and resend them if not. This can take up to $28.05\ \mu s$ (T_{refr}).

The abort command is not concerned by this bug. It is interpreted by another part of the the memory backend (*memctl_a_backend.vhd*) and not the memory controller.

experimental result Experiments with the service module (see 4.6) have proved that it is possible to write and read correctely with this firmware version. The driver version $1.0.0a2$ was used.

4.1.5 Latest Firmware version

Because of the insufficient experimental results it was decided to improve the firmware. The new version makes it possible to write an efficient driver.

The firmware reacts on a submitted command as soon as possible. The driver doesn't need to check if the command is accepted. The firmware gives priority to ongoing transfers. Only the abort command can stop them. If a transfer is done the controller takes the latest submitted command or idles.

A memory transfers can have a length up to the size of a SDRAM block (16 MB). There is no time limitation between the setup of a transfer and the read or write access to the Abstraction Layer FIFOs.

4.1.6 Register Block

The register block is the communication interface to the operating system of the PC. The block consists of four 64-bit registers. Commands to the module are written in the registers. Table 2 shows the registers which accept commands, writing to the other ones takes no effect. They are marked with an X. The last column is the offset to the module base address in the virtual address space.

<table>
<tr><th></th><th>Byte# 7</th><th>Byte# 6</th><th>Byte# 5</th><th>Byte# 4</th><th>Byte# 3</th><th>Byte# 2</th><th>Byte# 1</th><th>Byte# 0</th><th>Offset</th></tr>
<tr><td>GPR</td><td colspan="8">grp</td><td>24</td></tr>
<tr><td>MEMR</td><td colspan="2">X</td><td>cmd</td><td>sw</td><td colspan="4">X</td><td>16</td></tr>
<tr><td>CFGR</td><td>X</td><td>tgprc</td><td>X</td><td>ccs</td><td colspan="4">X</td><td>8</td></tr>
<tr><td>MCSR</td><td colspan="6">X</td><td colspan="2">tc</td><td>0</td></tr>
</table>

Table 2: Writable registers

The register MCSR:tc sets the firmware timing configuration. This is ignored by the current version. The Target general-purpose registers configuration register (*CFGR:tgprc*) controls the access to the Target registers. The function of the bits is shown in table 3. The address of the Target register is written to *tgprAddr*. A write access is indicated by *tgprWr*='1', otherwise it is a read access. If the flag *tgprKeep* is set, the *CFGR:tgprc* register is loaded from the bits 15..0 of the *GPR* register at the next write access to the register block (see 4.3.3).

Bit(s)	7	6	5..0
flag name	tgprKeep	tgprWr	tgprAddr

Table 3: The Target general-purpose registers configuration register

The SelectMAP configuration controller get its command from the *CFGR:ccs* byte. It also shows status information there. In table 4 the register which hold information are shown. If *CFGR:tgpIdle* is set the controller is in the IDLE state. *CFGR:sm* shows the actual state of the SelectMAP controller.

The *MEMR:cmd* is used to initiate a job request to the memory controller ("01"sets up a write operation, "10"a read operation and "11"is the abort command). The byte is assigned to the four controllers in the order AX, BX, AY, BY, beginning with the most significant bit. The byte can be read to get status information of the memory controller. The pattern "00"indicates that the controller is performing a refresh session, more precisely the main memory state machine (Figure 10) is in REFRESH state. The refresh sessions that are started, while the controller is busy with a read or write transfer, are take no effect no the *MEMR:cmd* byte.

The bus exchange switches are controlled by the byte *MEMR:sw*. The most significant bit controlles the X-side switch, the bit 6 the Y-side. Upon read access, the switch status is read.

	Byte# 7	Byte# 6	Byte# 5	Byte# 4	Byte# 3	Byte# 2	Byte# 1	Byte# 0	Offset
GPR	grp								24
MEMR	bst		cmd	sw	misccfg	X	mode		16
CFGR	tgpIdle	tgprc	sm	ccs	X				8
MCSR	fmv		X				tc		0

Table 4: Readable registers

Memory controller In section 5.2.4 of [7] the memory controller is described as single state machine. In the current version of the firmware this FSM[2] is replaced by four small state machines. These are closely coupled, either the read or the write state machine can leave its IDLE state. The main FSM must be in the corresponding state, READ or WRITE. It can't leave this state until the read resp. write state machine is entered the RDEND resp. WREND state. The refresh FSM starts when one of the state machines read, write or main is in the REFRESH state.

The main state machine (see figure 10) initialises the SDRAM after a *reset* signal. Afterwards it waits in the IDLE state. If a read transaction is requested, the main state machine enters the READ state. The signal *StateForceIdlexCI* goes low. That allowes the read state machine to leave the IDLE state (see figure 11). The write mechanism works similarely. Figure 12 shows the state diagram. The main state machine enters the REFRESH state if a refresh alert comes in and sends an enable signal to the refresh state machine.

The main FSM generates a two bit state info signal according to the current state. The pattern '01' stands for WRITE, '10 for READ, '11' for IDLE and '00' is the REFRESH state. The signal is used to multiplex the commands for the SDRAM, since each state machine generates such commands. The info signal also appears in the *cmd* byte of the register block. In fact the demultiplexer is a sequence of two demultiplexers (see figure 9). The refresh FSM runs if one of the other state machines is in REFRESH state. Therefore, the SDRAM commands of the refresh FSM overwrite the others if a refresh session is active ("RefrAlertxEI"='1').

[2]finite state machine

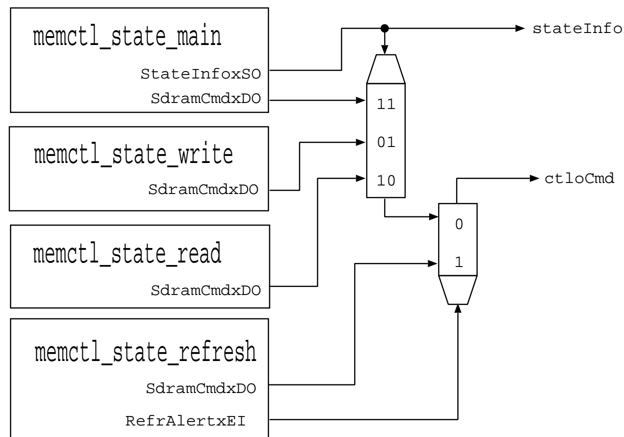

Figure 9: Demultiplexer for the SDRAM commands

In the RDPREP state the read FSM generates a signal to load the address and length of the transfer from the general-purpose register into the counters. In the RDOPEN state the ACTIVE command is sent to the SDRAM. This opens the page, which is addressed. The machine waits in RDWOPEN state for 3 clock cycles, so the timing requirement for the SDRAM is met. For the specification of the SDRAMs see [4]. If a refresh alert is pending the FSM can go from RDWND to RDREFR. In this state the refresh enable signal is set and the refresh state machine starts the session. Note, the *cmd* byte of the register block still indicates that the memory controller is in read mode.

In RDRD1 and RDRD2 the FSM reads from the SDRAM into the FIFO buffer. If the buffer full signal is high the read FSM falls back to RDWND, but the programmed read bursts are on their way. They arrive at the FIFO after the CAS latency. Therefore, the FIFO buffer reports that the buffer is full before there is no space left to prevent a buffer overflow.

In RDCLOSE the row is closed and the row counter incremented, this happens if the row boundary is reached or the word counter is zero. The FSM waits in RDWCLOSE and goes to RDOPEN to continue the transfer with the next row or to RDEND if there are no more words to read. In this state the main state machine is informed that the read operation is complete. It respons with the force idle signal. This mechanism prevents the read state machine from overrunning the IDLE state. The force idle signal (*StateForceIdelxCI*) is an asynchronous reset.

The Write state machine is in many ways equal to the read FSM. If the FIFO buffer is empty it sends a signal to the write FSM, which goes to the WRWDA state. The transition from WRWDA to WRCLOSE is used when the memory controller receives an abort command. In this case the word counter is reseted and the write or read state machines get a complete signal (*WrCompletexSI* or *RdCompletexSI*).

The refresh FSM performs the refresh session. the refresh alert causes the main, read or write state machine to go in the REFRESH state as soon as possible. Then the active machine generates the refresh enable signal (*RefrAlertxEI*). Now the refresh state machine becomes alive (figure 13). If there is an open page (*SdramOpenxSI*='1'), it is closed by the FSM and reopened after the refresh session.

In the state RFREFR an auto refresh command is sent to the SDRAM and a pulse (*ctloRefr*) notifies the refresh controller (*memctl_refr*). The machine waits in RFWREFR as long as the RK signal (*RefrKeepxEI*) is high. If the RD signal (*RefrDonexEI*) is low, the FSM goes back to RFRERF and sends the

next auto refresh command. The process stops when the RD signal (*RefrDonexEI*) is high. The machine goes then to RFEND and sends a refresh done signal to the other state machines. The state RWF ensures that the timing constraint between the refresh and the open command is met. The signals RA, RK and RD are generated by the refresh controller.

Incoming job requests, such as read ('10') or write ('01'), are stored in a queue with one place. This ensures that the memory controller never misses a job request, even if the main state machine is not in IDLE. An abort command causes the read or write state machine to go back to IDLE as soon as possible, they will close an open page. The refresh FSM is not affected by this command.

A structural overview of the memory controller is shown in figure 14. The figure represents the entity defined in *memctl_state.vhd*. The combinational logic, which generates the unconnected output signals, is not shown in order not to overload the figure.

4.2 Target FPGA

The Target FPGA implements the very own functionality of the TIKDIMM device. The FPGA is connected to the data bus of the SDRAM blocks. Beside the application the design includes two FIFO buffers and a set of registers, which are accessed by the *tgp* bus. A schematic of the top level is shown in figure 15. The schematic is not exhaustive.

4.2.1 Connection to the SDRAM

The Target reads data from the X-side SDRAM block. The output is written to the Y-side SDRAM blocks. The SDRAM memory is controlled by the Abstraction Layer FPGA. The Target must act as slave. Therefore, the incoming data is buffered. The Abstraction Layer gets 64-bit quad-word from the DIMM bus. The SDRAM is only 32 bit wide. So the quad-words are split and stored at subsequent address in the SDRAM. The AL memory controller programs the SDRAM to send bursts of two words to the Target. Each write request signal (*tgp(25)*) is doubled in the Target design and then sent to the input FIFO. Because of the CAS latency the signal goes through a shift register.

The input FIFO can hold up to 512 32-bit words. The signal *tgp(24)* goes to low when the buffer has only space free for two more words. This causes the memory controller to go from state RDRD2 to RDWNF. It sends no more read commands to the SDRAM. The buffer has enough space to take the last programmed read burst, which arrives after the CAS latency. The signal *inbufNotEmpty* is not connected to the outside world.

The write transaction from Target to the Y-side SDRAM works very similarly. The output FIFO buffer is of the same design than the input FIFO. The read request signal (*tgp(37)*) indicates that the controller has setup a two word burst. The outbuf FIFO must now deliver these words. The signal *tgp(36)* is low when the FIFO is empty. The write state machine uses this signal to interrupt the write transaction.

4.3 The GNU / Linux device driver

The TIKDIMM is treated by the operating system as character device. The char device driver is realised as kernel module. In [6] a exhaustive description of Linux device drivers is given.

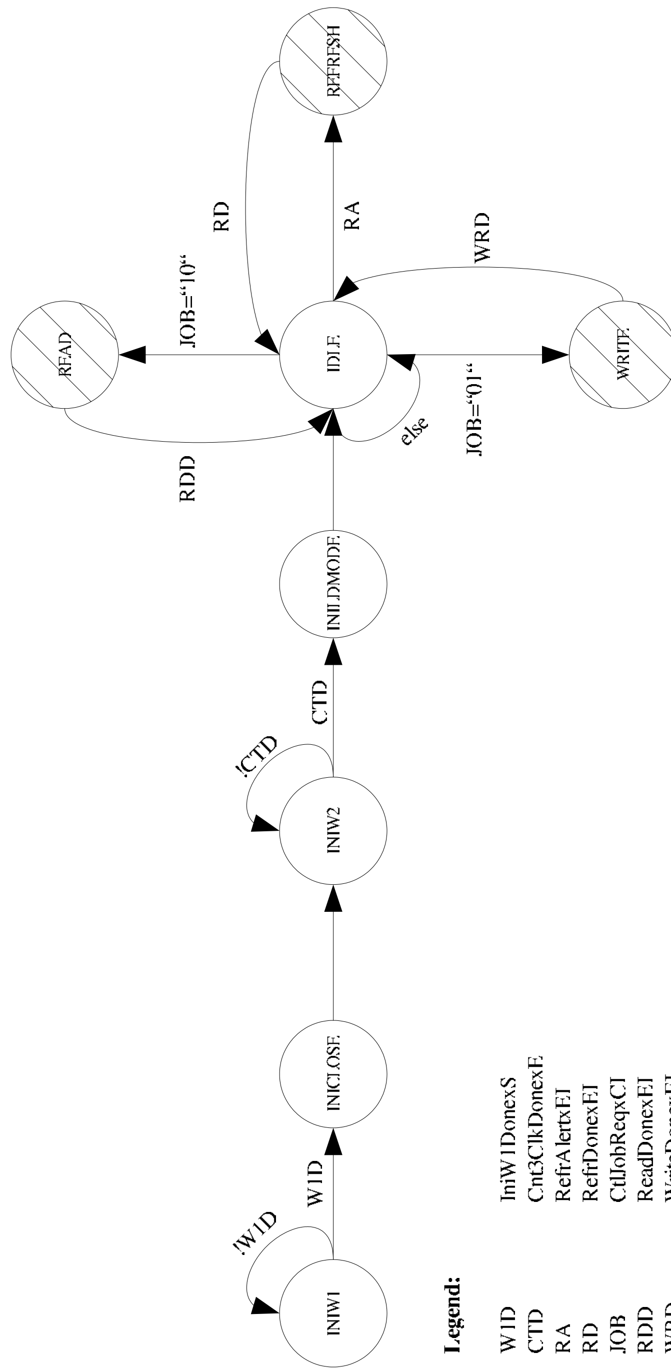

Figure 10: The main state machine of the memory controller

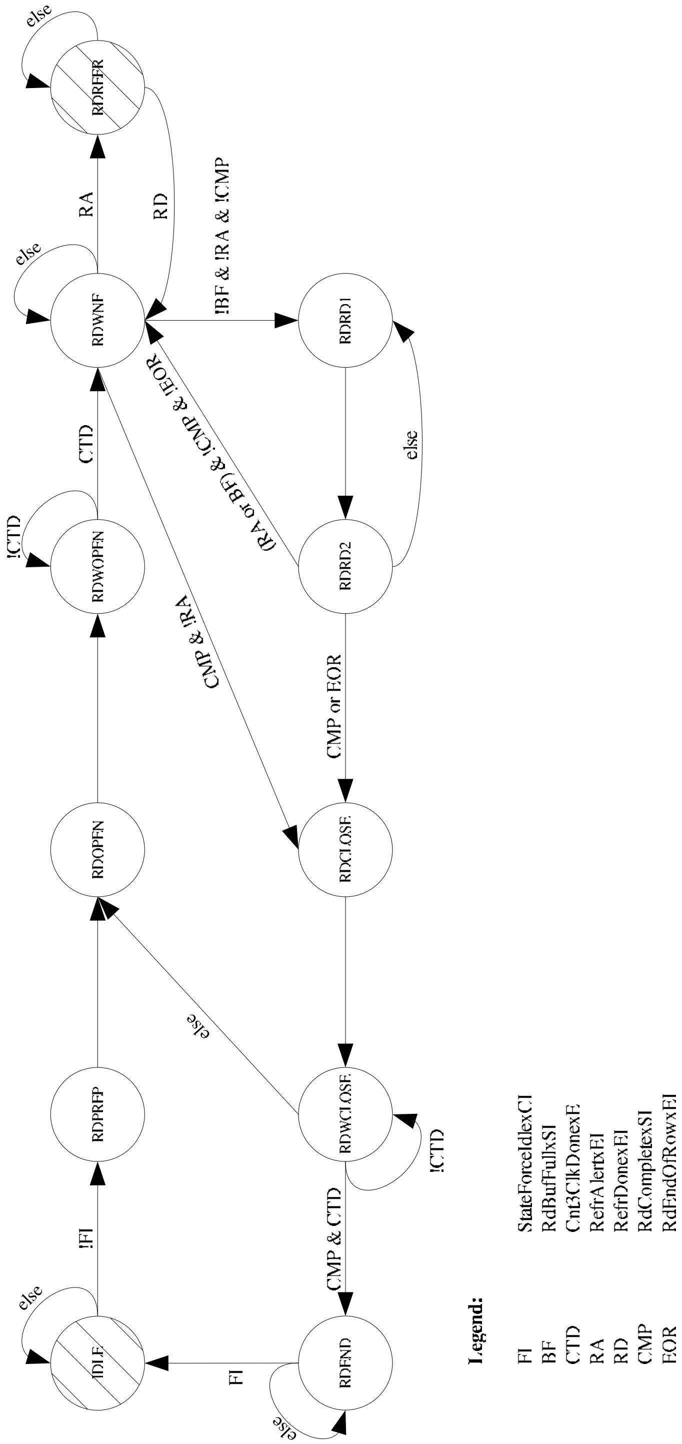

Figure 11: Read state machine of the memory controller

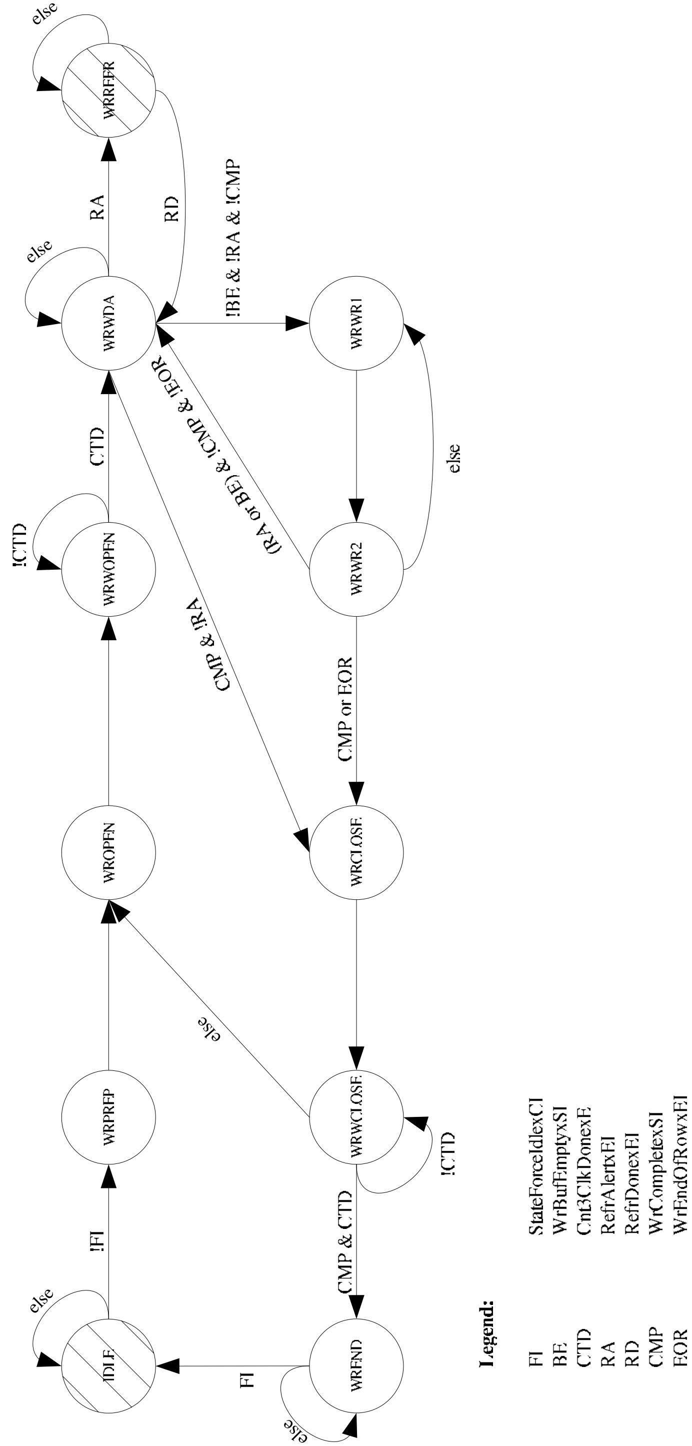

Figure 12: Write state machine of the memory controller

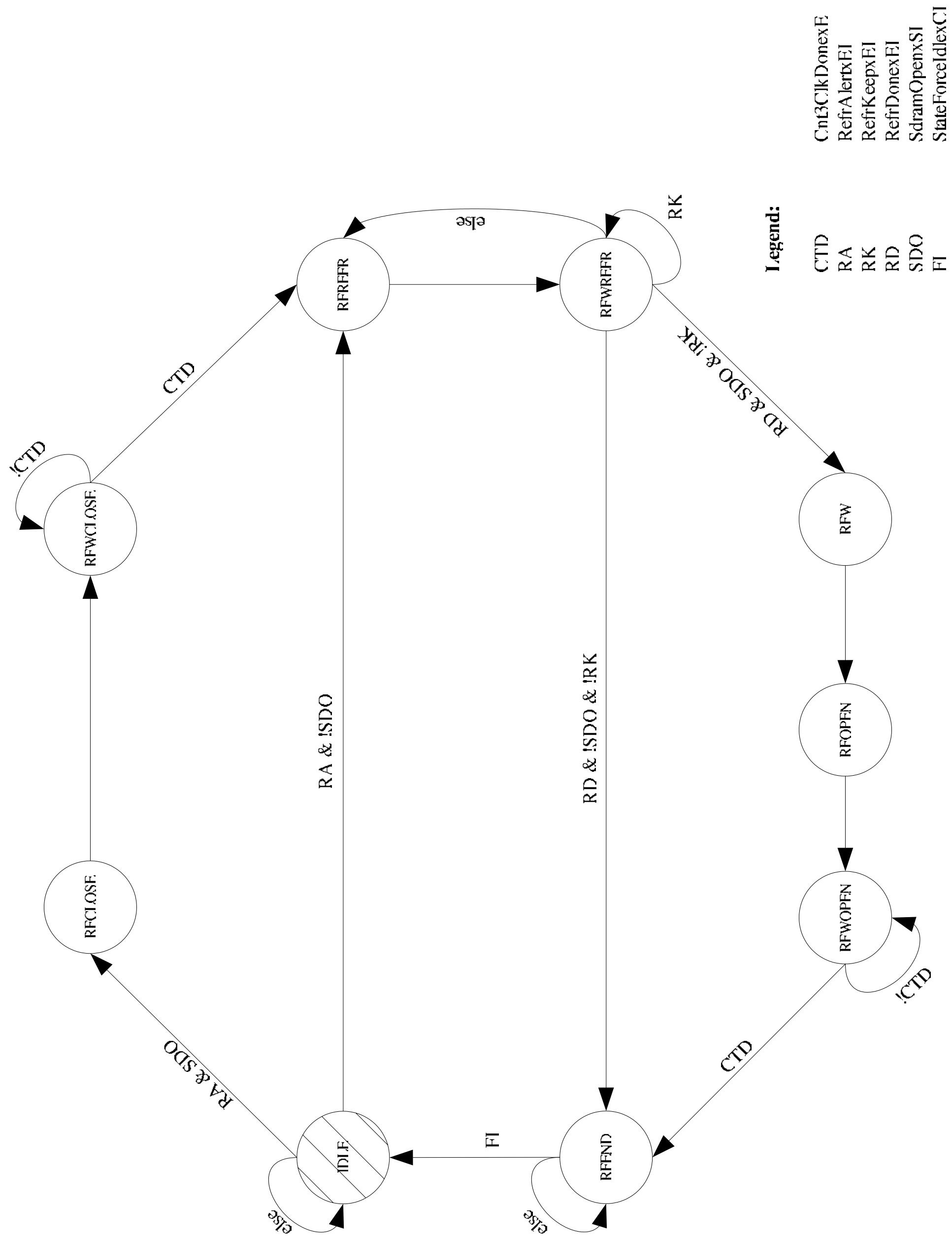

Figure 13: Refresh state machine of the memory controller

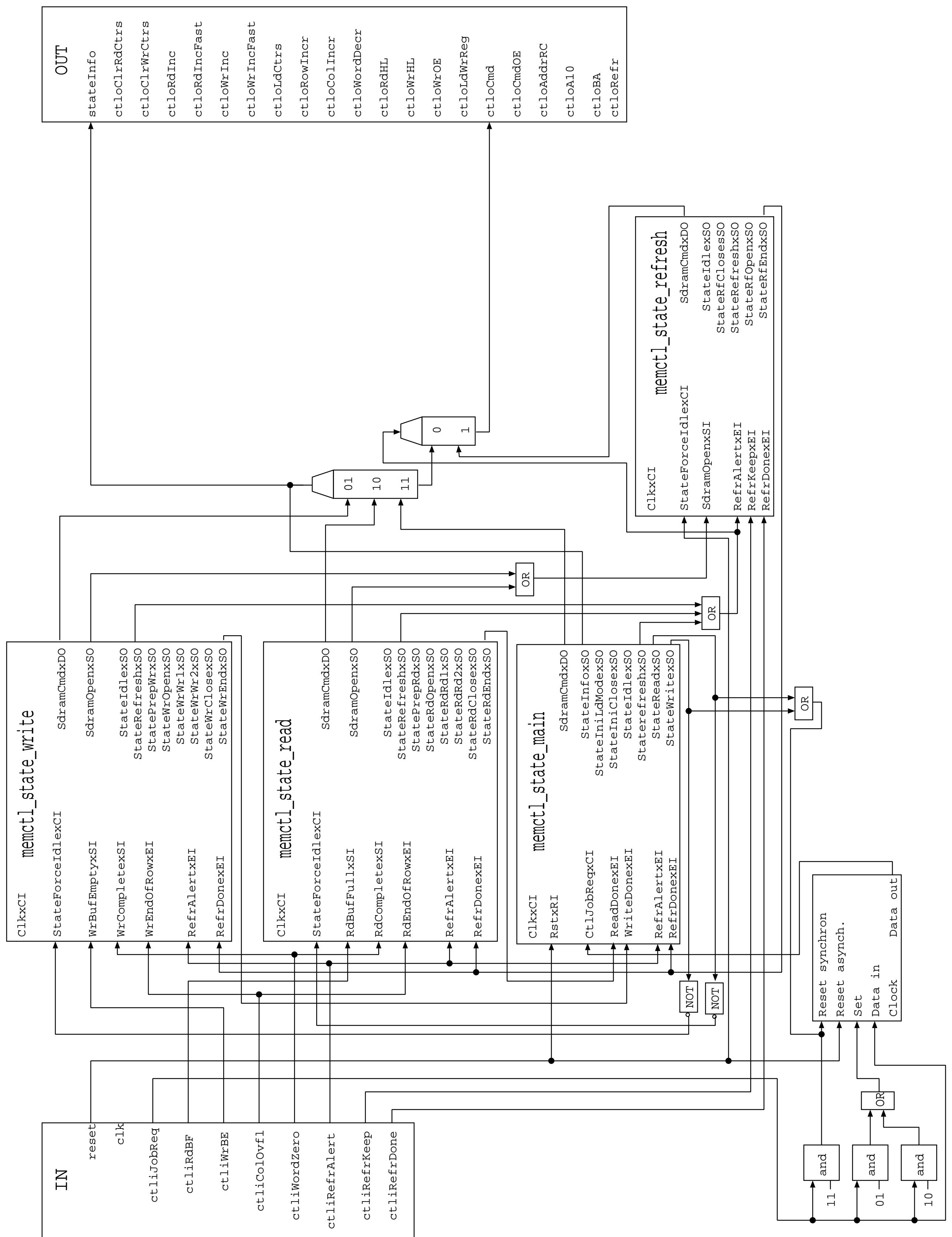

Figure 14: Structural connection of the four memory controller FSMs.

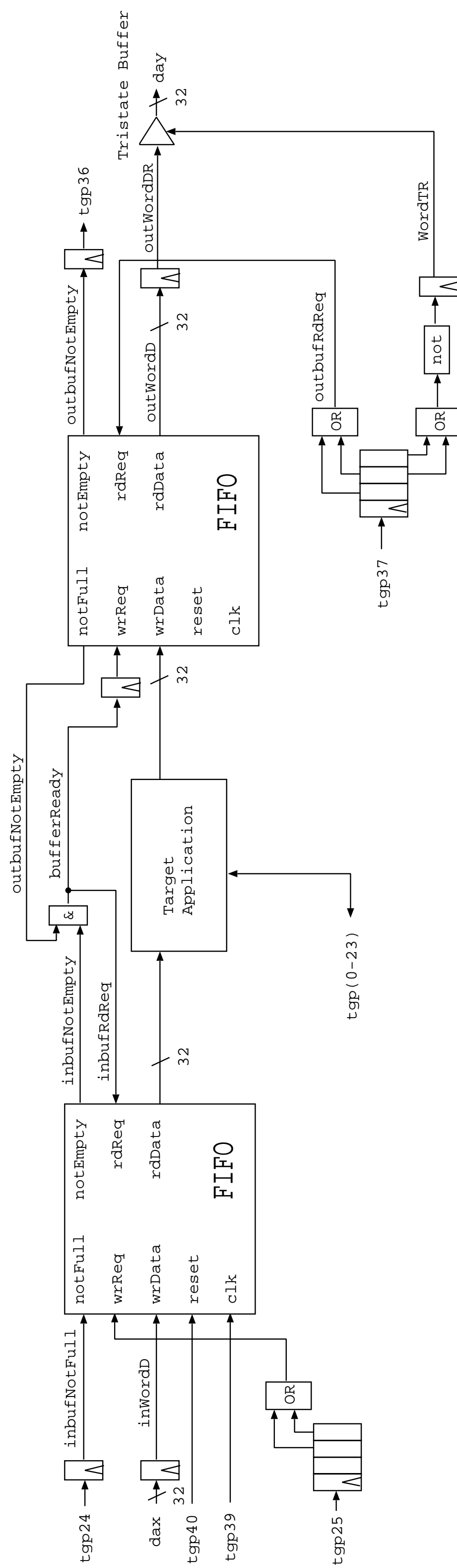

Figure 15: Schematic of the Target FPGA top level entity

Kernel modules are loaded into the linux kernel. As any other part of the system kernel the module is non preemptive.

The Driver is build from the files located in *toolkit/driver/*. The Makefile builds the kernel module called *tikdimmkm.o* and the command line tool *tikdimm*. Among other things this tool can load a configuration bit-stream into the Target FPGA. The "C"source file is *tikdimm_cli.c*. It is described in [7].

The Makefile knows two precompiler flags for the driver. The option *DEBUG* defines *TIKDIMM_-DEBUG_MK*, this will cause the driver to print a lot of messages. Furthermore it uses the /proc filesystem and creates the entry */proc/tikdimmkm*.

```
# Set to yes to activate debug messages (printk's)
DEBUG=yes

#Set to yes to make the driver simulate a TIKDIMM module
SIMULATE=no
```

The driver is compiled with *TKDM_SIMULATION_MK* if *SIMULATE* is set to *yes*. This option is supposed to use when no real TIKDIMM device is available. Such a driver never tries to access the TIKDIMM hardware.

The module needs to be loaded into the kernel. Since it doesn't have a GPL compatible MODULE-_LICENSE string, it taints the kernel. The system automatically assigns a major device number to the module. This number can be found in *proc/devices* under the module's name *tikdimmkm*. To access the driver from user space a device node is used. The node identifies the driver through the major device number, the name is irrelevant.

The *bash* script *load.sh* takes the above described actions. The device file is */dev/tikdimm*. With *unload.sh* the module is removed from the kernel.

Organisation of the source code

```
tikdimm_driver.c
tikdimm_tkm.c
tikdimm_csa.c
tikdimm_simul.c
tikdimm_driver.h
tikdimm_tkm.h
tikdimm_simul.h
tikdimm_regs.h
tikdimm.h
debug_macros.h
```

The sourcecode of the driver is split in several files.

- The kernel module's entry points are *init_module* and *cleanup_module*. These functions are located in *tikdimm_driver.c*. Moreover the system calls and the handler for the /proc filesystem entry is defined here.
- All chipset dependent code is located in *tikdimm_csa.c*.

- The implementation of the TKM kernel mode programming interface is in *tikdimm_tkm.c*. Function prototypes and constants for the for the arguments are defined in the header *tikdimm_tkm.h*. It is shared with every module that wants to use the TKM interface.
- The header *debug_macros.h* provides macros to print messages and check pointers or variables if they are on a 8-byte boundary in memory. These macros are intended to use for debugging.
- In *tikdimm.h* constants which are used in kernel and user space are defined, such as *ioctl* commands or the size of the TIKDIMM FIFOs.
- The module *tikdimm_simul.c* implements functions to simulate a TIKDIMM device. The prototypes are in *tikdimm_simul.h* defined.
- The header file *tikdimm_regs.h* defines macros which are used to access the registers or a FIFO of the TIKDIMM. Here the flag *TKDM_SIMULATION_MK* decides whether the macros are substituted by macros from *asm/io.h* to access the DIMM bus or by functions from *tikdimm_simul.c*

4.3.1 Kernel - Mode Programming Interface TKM

The TKM interface is a set of functions in the kernel space. It makes the driver functionality available for other kernel modules. The function bodies are located in *tikdimm_tkm.c*, which is part of the driver building tree. The header file *tikdimm_tkm.h* contains the function prototypes and constants to use with the functions.

All tkm-functions return 0 on success. Negative values describe an error. An invalid handle is indicated by $-EBADF$. Some functions can return further error codes.

Get access

```
int tkm_open(int oflag)

Parameter:
Return value:

void tkm_release(int tkm_handle)
```

To use the TKM interface a handle to the TIKDIMM platform must be gained. A subsequent call of a *tkm_* function takes this handle as argument. The *tkm_* function can determine which TIKDIMM device should be used. On successes *tkm_open* returns a handle and in case of failure a negative value. The current version of the driver can only deal with one TIKDIMM module and the parameter *oflag* is not used. Each call to *tkm_open* must be balanced with a call to *tkm_release*. The parameter is the handle of the device returned by *tkm_open*.

Miscellaneous function

```
int tkm_firmware_version(int tkm_handle, uint32_t *fwv)
```

The firmware version number is stored in *fwv*. A successful call returns with 0. An error is indicated by a negative number.

Target FPGA Register

```
int tkm_reg_read(int tkm_handle, int addr, uint16_t *val)
int tkm_reg_write(int tkm_handle, int addr, uint16_t val)

Error code: -EBADF : invalid handle
            -EFAULT: invalid pointer
            -EINVAL: invalid address
```

This two functions deal with the Target registers. The functions assume a number of registers. The number is defined in the macro *NR_OF_TARGET_REGISTERS* from *tikdimm.h*. *tkm_reg_read* copies the contents of the register *addr* into the 16-bit variable pointed to by *val* . An invalid pointer *val* is indicated by $-EFAULT$. Both functions return $-EINVAL$ if the address is out of range. *tkm_reg_-write* copies the argument *val* to the specified register. *Addr* can be in the range 0..*NR_OF_TARGET_-REGISTERS*-1. The meaning of the registers is fully application-dependant.

On-board Memory Access The following functions deal with the memory controllers or the associated FIFO buffers of the TIKDIMM. The controller or FIFO is selected by the parameter flag. It is a bitwise OR of *TKM_RAM_X*, *_Y* or *_XY* and *TKM_RAM_AL*, *_T* or *_ALT*. If *TKM_RAM_X* is part of the flag the command is aimed to the X-side, *_Y* means the Y-side and *_XY* both sides. *_AL* stands for Abstraction Layer and means the SDRAM blocks currently connected to this FPGA. The other two SDRAM blocks are connected to the Target FPGA and can be reached with the *TKM_RAM_T* bit set in the parameter flag.

```
int tkm_ram_control(int tkm_handle, uint8_t flags, uint32_t cmd)

Parameter : flags  : TKM_RAM_X, _Y, _AL, _T
                cmd : TKM_ABORT or TKM_RAM_AL2A, TKM_RAM_AL2B
Error code: -EBADF : invalid handle
```

This function is used to abort an ongoing memory operation on the TIKDIMM board. The argument cmd is set to *TKM_ABORT* and the flag determines which SD RAM controller is to stop.

The other functionality of *tkm_ram_control* is to set the bus switches. For this purpose cmd is set to *TKM_RAM_AL2A*, *_AL2B*, *_T2A* or *_T2B*. Of course *_AL2A* implies *_T2B* and *_AL2B* implies *_T2A*.

```
inline int
tkm_ram_status(int tkm_handle, uint8_t flags, uint32_t *stat)
```

The function *tkm_ram_status* copies the configuration status register to the variable pointed by *stat*. The parameter *flag* is ignored. It shall be set to (*TKM_RAM_XY* | *TKM_RAM_ALT*) in all invocations. The status register consists of the full and empty flags of the FIFO buffers, status information of the four memory controllers and the position of the bus switches.

To mask the buffer status bits of the 32-bit status register 16 macro constants are defined. For example *TKM_RAM_AX_RDBE* stands for the buffer empty bit attached to the FIFO buffer for read operations on the AX SDRAM block. *_RDBF* means the buffer full bit. *_WRBE* and *_WRBF* concern the FIFO'S for write operations. The macros exists for all four blocks depending on the infix *_AX*, *_BX*, *_AY* and *_BY*.

Each memory controller has two status bits. They show whether the controller is in IDLE, READ, WRITE or REFRESH mode. *TKM_RAM_AX_JOBMASK* masks the two bits associated with the AX controller. The pattern can be equal to *TKM_RAM_AX_IDLE*, *_READING*, *_WRITING* or *_REFRESHING*. This analogously works for the other SDRAM blocks.

Finally, the position of the bus switch can be checked with *TKM_RAM_AL2BX* and *TKM_RAM_-AL2BY*. If the bit is set the BX resp. BY SDRAM block is connected to the Abstraction Layer and AX resp. AY to the Target. If the bit is clear the connection is vice versa.

```
int
tkm_ram_rdprep(int tkm_handle, uint8_t flags, uint32_t addr, size_t len)

int
tkm_ram_wrprep(int tkm_handle, uint8_t flags, uint32_t addr, size_t len)

Parameter : flags  : TKM_RAM_X, _Y, _AL, _T
              addr : byte address
              len  : length in bytes
Error code: -EBADF : invalid handle
```

This pair of functions initiates a transaction between the on-board memory and a FIFO buffer. Each memory block has its own address space beginning at 0. The parameter *addr* takes the source resp. Destination address in bytes. The length of the transaction must be a multiple of 8 and represents a number of bytes. It is passed through the parameter *len*. *tkm_ram_rdprep* setup a read transfer from the SDRAM block and *tkm_ram_wrprep* setup a write transfer to the SDRAM block.

```
ssize_t
tkm_ram_read(int tkm_handle, uint8_t flags, uint8_t *buf, size_t nbyte)

Parameter : flags  : TKM_RAM_X, _Y, _AL, _T
            nbytes : numer of bytes
Error code: -EBADF : invalid handle
```

This command always affects a buffer that is linked to a SDRAM block which is currently connected to the Abstraction Layer. Therefore, the *TKM_RAM_AL* and *TKM_RAM_T* bit in flags are ignored. The function reads from the specified FIFO buffer into the buffer pointed to by *buf*. The parameter *nbytes* must be divisible by eight. Upon successful completion, *tkm_ram_read* returns the number of bytes read. This number is at most the size of the FIFO buffer (*TKM_RDBUF_SIZE8* defined in *tikdimm.h*). The caller of *tkm_ram_read* is responsible that the FIFO buffer holds enough quad-words, otherwise the function reads the last valid entry repeatedly.

```
ssize_t
tkm_ram_write(int tkm_handle, uint8_t flags, uint8_t *buf, size_t nbyte)

Parameter : flags  : TKM_RAM_X, _Y, _AL, _T
            nbytes : numer of bytes
Error code: -EBADF : invalid handle
```

The write operation work very similarly. The function writes at most *TKM_WRBUF_SIZE8* bytes. Nevertheless, the buffer can overflow if there were quad-words from a previous call in the buffer. The latest entry will be overwritten.

4.3.2 System Calls

The TIKDIMM driver provides the following calls: *open*, *release*, *read*, *write* and *ioctl* . The prototypes in a C/C++ environment are shown below. The *release* call is attached to *close*.

```
int open(const char *pathname, int flags);
int close(int handle);
ssize_t read(int handle, void *buf, size_t nbytes);
ssize_t write(int handle, const void *buf, size_t nbytes);
int ioctl(int handle, int command, ...);
```

open Before any of the other calls can be invoked, open must be called. As usual it returns a handle to the TIKDIMM device. The argument **pathname* points to a string with the device path */dev/tikdimm*. This device file is created by the load script *toolkit/driver/load.sh*. The parameter *flags* is ignored. If something is wrong the return value is -1.

release This call is invoked when all copies of the file structure associated with the handle are closed. Multiple copies exist when the process that called open forks after the call. The return value always 0. Each call to *open* should be balanced with a call to *release*.

4.3.3 ioctl

```
int ioctl (int handle, int command, ...)
```

The system call is implemented by the TIKDIMM driver. The type of the third argument depends on the used command. It is passed as reference. In this section it is referred to as argument. On success the function returns 0. If ioctl gets a command which is not supported it returns $-$*ENOTTY*. If the driver detect some problem with the hardware or the argument is not accessible the return value is $-$*EFAULT*. The command *TIKDIMM_IOC_SET_CONFIG_MODE* returns $-$*EINVAL* if the argument is unknown.

All constants with the prefix *TIKDIMM_IOC_* (macros) are supposed to use as ioctl commands. They are defined in tikdimm.h.

Version information The command *TIKDIMM_IOC_GET_DRIVER_VERS* requires a reference to a 256 byte *char* array. *TIKDIMM_IOC_GET_FIRMWARE_VERS* needs only a eight byte wide *char* array. The first command gives the driver version as string with date and time of compilation. The second returns a string with the firmware version.

SelectMAP configuration controller This controller handles the configuration of the Target FPGA. Status information is read from the *ccs* byte, which is part of the module register. This command is called *TIKDIMM_IOC_GET_CONFIG_MODE*. It copies the *ccs* byte to the argument. The bits *DONE*, $\overline{INIT}$, respectively *cs0* can be masked with the constants *TIKDIMM_CONFIG_FLAG_INIT*, *_DONE* or *_CS*. The mask *TIKDIMM_CONFIG_MODE_MASK* delivers the bitfield *mode*.

The command *TIKDIMM_IOC_SET_CONFIG_MODE* sends instruction to the SelectMAP controller by setting the *ccs* byte. Valid arguments for this ioctl call are *TIKDIMM_CONFIG_CANCEL*, *_RECONFIGURE*, *_READ* and *_WRITE*.

The configuration bit-stream itself is written bytewise with the command *TIKDIMM_IOC_WRITE-_CONFIG_BYTE*. The driver writes the argument of type *char* to the fifth byte of the general purpose

register. The command *TIKDIMM_IOC_READ_CONFIG_BYTE* retrieves the configuration bit-stream from the Target FPGA. For more detailed information see [7].

Bus Switches The TIKDIMM has two bus switches. They connect the on board SDRAMs either to the Target FPGA or the Abstraction Layer (see [7]).

The command *TIKDIMM_IOC_SET_MEM_SWITCHES* set both bus switches according to the provided argument. The argument is of type *char*. It's value and meaning are described in table 5. The first column holds the arguments's value. The second tells whether the RAM block AX is connected to the Abstraction Layer (AL) or the Target FPGA (T).

The command *TIKDIMM_IOC_GET_MEM_SWITCHES* reads the switch control byte. It represents the current switch position according to table 5.

Argument: *char* &arg	AX	AY	BX	BY
0	AL	AL	T	T
TIKDIMM_MEMSWITCH_X	T	AL	AL	T
TIKDIMM_MEMSWITCH_Y	AL	T	T	AL
TIKDIMM_MEMSWITCH_X & TIKDIMM_MEMSWITCH_Y	T	T	AL	AL

Table 5: Value and meaning of the bus switch control byte.

Register of the Target FPGA To write to a 16-bit register of the Target two *ioctl* calls are required. The first command is *TIKDIMM_IOC_SET_TGPRC*. It takes an argument of type *char* with the address and the two most significant bits set. The second step is *TIKDIMM_IOC_SET_GPRW0*. This call has an argument of type *unsigned int*. The bytes 0 and 1 are copied to the register. If the flag *tgprKeep* is set, the byte 2 is the next *tgprc* register. So the next Target register can be set with the command *TIKDIMM-_IOC_SET_GPRW0*. The preparation with *TIKDIMM_IOC_SET_TGPRC* is not needed.

The read operation starts also with *TIKDIMM_IOC_SET_TGPRC* command, but the argument holds only the address. The command *TIKDIMM_IOC_GET_GPRW0* copies the register value to the lower two bytes of the argument of type *unsigned int*.

All registers can be read by the command *TIKDIMM_IOC_SET_TGPRC* with 80(hex) as argument. Subsequent *TIKDIMM_IOC_GET_GPRW0* commands will read the register values beginning a the address 0.

The command *TIKDIMM_IOC_GET_TGPRC* reads the tgpr register byte from the configuration status register.

Transactions to the Target FPGA All commands that set up a transaction to a SDRAM block take an argument of type *ioctlDataDesc*. The argument is passed as reference. This structure is defined in tikdimm.h. The field *addr* contains the address of the on-board SDRAM. Each 16MB SDRAM block has its own address space form 0 (hex) up to 1'000'000 (hex). The address is in bytes. The length in bytes of the transaction stands in the field *nr*. The ioctl function doesn't write to the structure.

```
typedef struct {
    unsigned int addr;
```

```
    unsigned int nr;
} ioctlDataDesc;
```

The user interface of the driver is designed to support data processing from the X-side of the TIKDIMM to the Y-side. There exist two commands to make the Target FPGA work. The command *TIKDIMM-_IOC_SETUP_X2T* initiates a read transaction from X-side. whether the Target reads from AX or BX depends on the position of the bus switch. This command doesn't touch the switch.

The Target must write the processed data to the Y-side. This is achieved by using the command *TIKDIMM_IOC_SETUP_T2Y*. Depending on the bus switch position the Target writes to AY or BY.

With the following three commands the Target activity can be supervised. The command *TIKDIMM-_IOC_X2T_ACTIVE* checks if there is an ongoing read transaction from the SDRAM block AX or BX. The provided variable of type *char* is then set to 1, if the transaction has finished the variable is 0. Similarly, the *TIKDIMM_IOC_T2Y_ACTIVE* comand checks for activity on the Y-side. Finally, *TIKDIMM-_IOC_X2Y_ACTIVE* set the argument to 1 if the corresponding X or Y-side memory controller is running.

Prepare read / write operations The *TIKDIMM_IOC_ABORT* command aborts a read or write operation to the specified SD RAM block. It immediately stops the operation and clears the associated FIFO buffer. The memory controller state machine returns to IDLE state. The argument is one or more of the constants: *TIKDIMM_ABORT_AX*, *_AY*, *_BX* or *_BY*. They can be connected with a bitwise OR.

Before *read* can be called the TIKDIMM must prepare this read operation. This is achieved by the command *TIKDIMM_IOC_SETUP_RD_AX*, *_BX*, *_AY* or *_BY* depending on the desired source block. The argument is of type *ioctlDataDesc*. This command sets the affected bus switch as necessary without regard to any ongoing memory operation.

The *write* system call needs a similar preparation. The *ioctl* commands have the prefix *TIKDIMM_-IOC_SETUP_WR_*. The argument of type *ioctlDataDesc* carries the destination address and the transfer length in bytes.

4.3.4 read

```
size_t read(int handle, void *buf, size_t nbytes)
```

The *read* call is implemented only as non blocking opeartion. The function tries to read *nbytes* from the TIKDIMM. It returns the number of read bytes or $-EAGAIN$ if no bytes could be read. The function never reads more bytes then the TIKDIMM FIFO buffer can hold and it rounds the parameter *nbytes* down to a multiple of eight. In the kernel space the function stores the data in a global buffer whose base address is on a 8-byte bounding in memory. The data is then copied to the buffer pointed to by *buf* in user space.

If the buffer full flag is set the function reads the whole FIFO buffer without further check. If the FIFO is not or not yet full when the function wants to start reading, it enters the so called single read mode. The *read* function checks the buffer status, before each following read access to the DIMM bus. If the buffer is empty the function returns with the number of read bytes.

4.3.5 write

As soon as the read/write data is copied from the user space to kernel space it is eight aligned.

```
ssize_t write(int handle, const void *buf, size_t nbytes);
```

First of all the function copies the number of bytes specified by *nbytes* from the buffer pointed to by *buf* to the kernel space. At most the function copies the TIKDIMM FIFO buffer's size. The parameter *nbytes* is rounded down to an eight divisible number, as the function writes quad-words. The write buffer allocated in the kernel space is eight aligned. This is importand because the used instruction to write on the DIMM bus requests a buffer with eight divisible address.

If the FIFO buffer is empty the function writes to the DIMM bus. It returns the number of written bytes. When the return value is 0 the FIFO buffer wasn't empty. The *write* call is non blocking.

4.3.6 Usage of the Driver, an example

This section illustrates the usage of the system calls provided by the TIKDIMM driver module *tikdimmkm.o*. First of all the devise is opened. The the driver and firmware version numbers are retrieved.

```
int handle;
handle = open("/dev/tikdimm", O_RDWR | O_NONBLOCK);

char strDriverVersion[256];
char strFirmwareVersion[8];
ioctl(handle, TIKDIMM_IOC_GET_DRIVER_VERS, &strDriverVersion);
ioctl(handle, TIKDIMM_IOC_GET_FIRMWARE_VERS, &strFirmwareVersion);

printf("The driver version %s is loaded and the TIKDIMM firmware is %s",
        strDriverVersion, strFirmwareVersion);
```

Here a write operation to the address 0x50 of the BX SDRAM is demonstrated. The data size is *N*. This *ioctl* call also turns the bus exchange switch of the X-side to the necessary position.

```
const int N = 0x100;
char *ptBuf;
ptBuf = new unsigned char[N];
int iBufOffset = 0;
int iCountDown = N;
int n;
ioctlDataDesc gIoctlData;

...

gIoctlData.nr = N;
gIoctlData.addr = 0x50;
ioctl(handle, TIKDIMM_IOC_SETUP_WR_BX, (char*) &gIoctlData);
```

After the setup the buffer contens is written to the device.

```
do
{
    n = write(handle, ptBuf + iBufOffset, iCountDown);
```

```
    if (n > 0)
    {
        iBufOffset += n;
        iCountDown -= n;
    }
} while (iCountDown > 0);
```

This code lines make the Target write *N* bytes to the Y-side SDRAM which is currently connected to the Target.

```
gIoctlData.addr = 0x200;
ioctl(handle, TIKDIMM_IOC_SETUP_T2Y, (char*) &gIoctlData);
do
{
    ioctl(handle, TIKDIMM_IOC_T2Y_ACTIVE, &b);
} while (b == 1);
printf("Transfer done.\n");
```

4.4 Library API

The following sections describe the member functions of the TIKDIMM class. In general the functions return '0' if successful and a negative value if an error occurred.

4.4.1 Initialisation

```
TIKDIMM();
```

The standard constructor takes no argument.

```
int Init()
int Init(const char * target)
```

This method must be called before any job for the TIKDIMM hardware can be created. It is safe to call it again but not necessary. The argument is a pointer to a pathname. The file is the configuration-bitstream for the Target FPGA. If the pointer is *NULL* or the version without argument is called the Target is not configured.

After this initialisation the object uses the default value for the maximum packet size (*PACKET_-MAX_SIZE*). The constant is defined in *tkdm_lib.h*. A predictable stream ratio of 1:1 is assumed. See section 4.4.4 for more information.

4.4.2 Main functions

```
int
CreateJob (unsigned char * inBuf, unsigned char * outBuf,
           unsigned int nBytes, unsigned int nOutBuf);
```

This function defines a job and add it to the job list. The data which is to process is pointed to by *inBuf*. The number of bytes is passed in *nBytes*, it should by a multiple of eight, since the TIKDIMM works with 64-bit quad-words. The output is copied to *outBuf*. The parameter *nOutBuf* tells the library the size of the output buffer.

The TKDM library has no internal buffers. The application must provide in and output buffers. They must be valid till the coresponding *Wait* call returns control to the user application. The processed data is then located at *outBuf*. Data at *inBuf* is not touched and still valid.

Since the jobs are stored in a list, the *CreateJob* method can be called an arbitrary number of times, before *Wait* is invoked.

```
int Wait()
```

This method processes the next job of the job list. It returns control when the job is done. The user application may wish to fork before *Wait* is called.

In case of an overflow of the output buffer, the method calls the function pointed to by *outBufferFull*. After the instantiation this is *OutBufferFullDefault*. This default handler prints a message. The *Wait* method clears the output buffer after the handler returns. The application can define a handler of its own.

```
int
SetOverflowHandler(void (*outBF)(unsigned char *, size_t));
```

The argument of this *SetOverflowHandler* method is the user defined handler. It has two arguments. The first one points to the output buffer that is about to overflow. The second one gives the number of bytes in the buffer, so the handler knows how many bytes it has to copy to a save place. This parameter *s* is not identical with the size of the output buffer, that was given in the *CreateJob* method, but of course it is never bigger.

```
void OutBufferFullHandler(unsigned char *pt, size_t s)
{
    printf("User handler\n");
    // pt: pointer to out buffer
    // s : number of bytes in pt
}
```

4.4.3 More Methods

```
int Reset()
```

The *Reset* function aborts all memory transfer of the TIKDIMM module. AX and AY are connected to the Abstraction Layer, BX and BY to the Target. The job list is cleared. The reset fails when a system call reports an error. Packet size and stream ratio are not changed by the *Reset* call.

```
int GetLibraryVersion()
```

The Library has a major and a minor version number. They are defined in *tkdm_lib.h* as macro constants (*LIBRARY_VERSION_MAJOR*, *_MINOR*). The function returns the sum of the minor and the major number left shifted by eight.

```
int SdramTest()
int SdramTest(unsigned long long N, unsigned long long  RamAddr);
```

This is a built in SDRAM test. It writes test data to each SDRAM block and read it back. The argument *N* is the number of bytes which are used for each SDRAM block. The desired start address in the on-board memory is given in *RamAddr*. Without arguments th function uses default values that are defined in the function body in *tkdm_lib.C*.

4.4.4 Length of the output data

The TIKDIMM class distinguishes between predictable and unpredictable size of the output data. In the first case the output data length has a fixed relation to the input data length. The *Wait* function calculates from the input data size the expected output size. The function counts the output bytes and uses this information to determine when the Target is done. Per default the library assumes that the Target produces as many output as input bytes. The method *SetStreamRatio* changes this relation.

In case of an unpredictable size of the output data the *Wait* function assumes an infinite output stream. If the read transfer from the Target times out the *Wait* function stops data reading.

```
int SetStreamRatio(unsigned int z, unsigned int n)
```

The input to output relation is set to z to n. If one of the parameters is 0 the TIKDIMM object treats the output data size as unpredictable.

```
int ChangePacketSize(unsigned int s)
```

The TIKDIMM class knows a packet size for the input stream, one for the output and a maximum packet size, this is equal or smaller than the size of an on-board SDRAM block. The function sets the maximum packet size to s. The relation from input to output packet size is equal to the stream relation set by *SetStreamRatio*. The input and output packet sizes are chosen as big as the maximum packet size allowes. If the output size is unpredictable, both packet sizes are set to the maximum.

4.4.5 Functions related to the Target FPGA

These functions return zero on success.

```
int ConfigureFromFile(const char *fname)
```

Configures the Target FPGA with a bitstream. The argument is a pointer to a string containing the filename of the bitsream. This file is generated by the Xilinx tools and has the extension *.bin*.

```
int ReadbackToFile(const char *fname)
```

This function reads the Target FPGA configuration and stores it in a file pointed by the argument. The file will be created or overwritten if it already exists.

```
int ReadTargetRegister(unsigned regAddr, unsigned * &ptRegData)
int ReadTargetRegister(unsigned * &ptRegData)
```

The Target design implements up to 64 16-bit registers. The macro *NR_OF_TARGET_REGISTERS* in *tikdimm.h* defines their number. The first argument is the register address, valid values are between 0 and *NR_OF_TARGET_REGISTERS - 1*. The second argument is set to a pointer to a static buffer containing the register value. This buffer remains valid until the next call of the function. The version with only one argument dumps all registers to the static buffer.

```
int WriteTargetRegister(unsigned regAddr, unsigned regData)
```

This function writes the second argument to the specified 16-bit register.

4.5 Reference Application

The reference application demonstrates the usage of the TKDM library. The project directory is *toolkit/-reference_app*, it contains the application source *refapp.C*, the makefile and the bitstream for the Target FPGA. The application assumes that the TKM library header and source are located at *toolkit/tkdm_lib*. Below a simplified version is discussed.

Unlike the driver the library and the reference application is written in C++. The application is compiled and liked with the following commands. The TKDM library uses the *list* container from the standard template library.

```
g++ -c -o refapp.o refapp.C
g++ -c -o tkdm_lib.o tkdm_lib.C
g++ refapp.o tkdm_lib.o -o refapp
```

The application source starts with the including of the necessary header files. Then an object *gModule* of the class *TIKDIMM* is globally instantiated. It only makes sense to create one *TIKDIMM* object, since the current driver is capable of supporting one module in slot 3. The *main* function allocates the input and output buffers for the job.

```
#include <stdio.h>
#include "../tkdm_lib/tkdm_lib.h"

TIKDIMM gModule;

int main()
{
    const int N = 0x800;
    unsigned char * in;
    unsigned char * out;
    in = new unsigned char[N];
    out = new unsigned char[N];
    ...
```

Now we produce some test data, they look like "0101010101010101", "0202020202020202",...

```
    int i, j, n;

    // Create Test Data
    for (i=0, j=0; i < N; j++)
    {
        for (n=0; n < 8 ;i++, n++)
            in[i] = j;
    }
```

The *Init* method loads the *top.bin* configuration-bitstream in the Target FPGA. the Target just copies data from the X-side to th Y-side. Moreover *gModule* is now ready to use.

```
    gModule.Init("./top.bin");
```

The overflow handler of section 4.4.2 can be installed.

```
    gModule.SetOverflowHandler(OutBufferFullHandler);
```

Now a ram-test is performed. It writes 4kB to the address 0x200.

```
// SDRAM Test
//------------------------------------------------
printf("Start SDRAM Test...\n");
gModule.SdramTest(0x1000, 0x200);
printf("...SDRAM Test finished\n");
```

It is time to create a job. The maximum packet size is changed from the default value (16MB) to 0x80 bytes.

```
// Target FPGA
//------------------------------------------------
gModule.ChangePacketSize(0x80);

gModule.CreateJob(in, out, N, N);
```

The job is now ready to process and waits in the job-list.

```
gModule.Wait();
```

When *Wait* returns the control flow to the application the job is processed and the output (a copy of the input buffer) is located in the buffer *out*. A job like the following would cause the library to call the handler (*OutBufferFullHandler*). The output buffer size defined in the fourth argument (*N/2*) is to small.

```
//gModule.CreateJob(in, out, N, (N/2));
```

Display the output.

```
//Print the result
printf("Result:\n");
unsigned long long * outL;
outL = (unsigned long long *)out;

for (i=0; i < N>>3; i++)
    printf("0x%016llX\n", outL[i]);
```

The Target design *top.bin* implements 16 registers. The application fills the 16-bit register with some test values.

```
// Target FPGA Register
//------------------------------------------------

printf("Test of the Target FPGA registers\n");
unsigned short * ptTargetRegister;
for (i = 0; i < 16; i++)
    gModule.WriteTargetRegister(i, i+(i<<8));
```

The register at address 5 for example can be read out.

```
// get eg. register 5
gModule.ReadTargetRegister(5, ptTargetRegister);
printf("Read register R%i = 0x%04X\n", 5, *ptTargetRegister);
```

The *ReadTargetRegister* method can dump all registers at once. For this purpose the version without the address argument is used.

```
    printf("Read all registers:\n");
    gModule.ReadTargetRegister(ptTargetRegister);
    for (i = 0; i < 16/*64*/; i++)
        printf("R%i = 0x%04X\n", i, *(ptTargetRegister + i));
```

Clean up the memory:

```
    printf("\nEnde\n");
    delete []in;
    delete []out;
    return 0;
}
```

4.6 Service module

This tool was developed for debugging. Memory transfers between the on-board SDRAM and the Abstraction Layer or the Target are possible. The tool can write to the TIKDIMM a set of test data generated by the module itself. The data can be read back and verified. This is useful to thest the firmware.

The service module *ram_km.o* is a kernel module. It makes use of the driver's TKM interface. The module provides an *ioctl* system call. This allowes an application in the user space to invoke some operations. Table 6 shows the implemented commands.

RTAPP_INIT	Generate a set of test data. The size is passed as argument.
RTAPP_WRAX	write to AX
RTAPP_WRAY	write to AY
RTAPP_WRBX	write to BX
RTAPP_WRBY	write to BY
RTAPP_RDAX	read from AX
RTAPP_RDAY	read from AY
RTAPP_RDBX	read from BX
RTAPP_RDBY	read from BY
RTAPP_AX2AY	Target reads from AX and writes to AY
RTAPP_BX2BY	Target reads from BX and writes to BY

Table 6: Ioctl commans for the service module

The module defines a task queue and creates a kernel thread, which periodically awakes. Each *ioctl* call creates a task. The thread works on the current task or takes the next from the queue before it goes to sleep. This allowes other processes to run.

The /proc filesystem is supported. The entry *ram_test* benchmarks the TKM interface and provides statistical information.

The module source consists of several files. They can be found at *pathname/ram_test*. The directory also holds files related to the project.

- *ram_mod.c* contains the module entry points, the handler for the /proc filesystem and the implementation of the *ioctl* system call.

- *kthread.c* provides functions to crate, clear and manage a kernel thread.
- *kthread.h* contains structures and prototypes. It is shared with *ram_mod.c*.
- *ram_km.o* is the kernel module.
- *app.c* is the source of the test application. It runs in user space. When it is called without an argument a help message is printed. The function provides several test, which are chosen by the argument.
- *app.h* is included from *ram_mod.c* and *app.c*. It defines macro constants for the *ioctl* call.
- The *Makefile* creates the kernel module. *make app* builds the test application *app*.
- *load.sh*. This script takes all necessary steps to load the kernel module. The device node and the /proc filesystem entry are created.
- *unload.sh* removes the module from the kernel.

5 Status and Future Work

5.1 Status

5.1.1 Initial firmware and driver version

The initial version of the firmware was tested intensely. It was found that the memory controller doesn't handle row boundaris corectely. By suppressing the refresh alert, during read and write access, this bug was fixed. With a kernel module, which access the driver through the TKM-interface, it is possibel to write and read from the TKDM module. These operations seem to work. Transfers from and to the Target FPGA showed problems. The reason is unclear.

The main disadventage of the above solution is that the driver is not efficien. It must ensure that the firmware doesn't miss a refresh alert. Therfore, the firmware was improved. The new version can react on refresh alerts at any time, so the driver can setup transfers with a length up to 16MB.

5.1.2 Latest firmware and driver version

A Linux driver (version 1.3.0) has been implemented which provides a rich interface to the user space. Read and write access through the TKM-interface work. The system call read seems to have a bug. The first quad-word of each call is lost. This bug appears not every time when read is called.

5.1.3 TKDM Library

The library provides an interface to user space applications. It is possible to use the TKDM module with a few function calls. The library was tested as good as possible.

5.2 Future Work

The firmware generates a puls when the driver accesses the FIFO buffer. This puls indicates to the FIFO that the driver wants to read the next quad-word. Latest experiments show that this puls comes too often, when the first quad-word is lost appears.

References

[1] O.Y.H. Cheung and P.H.W. Leong. Implementation of an fpga based accelerator for virtual private networks. In *Proc. of the 2002 IEEE International Conference on Field Programmable Technology (FPT'02)*, pages 34–41, 2002.

[2] K.H. Lee D.K.Y. Tong, P.S. Lo and P.H.W. Leong. A system level implementation of rijndael on a memory-slot based fpga card. In *Proc. of the 2002 IEEE International Conference on Field Programmable Technology (FPT'02)*, pages 102–109, 2002.

[3] P. Leong, M. Leong, O. Cheung, T. Tung, C. Kwok, M. Wong, and K. Lee. Pilchard - a reconfigurable computing platform with memory slot interface. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE CS, April 2001.

[4] Micron. *MT48LC4M32B2 - 1 Meg x 32 x 4 banks Synchronous DRAM*, Rev E edition, October 2002.

[5] Christian Plessl and Marco Platzner. TKDM - a reconfigurable co-processor in a PC's memory slot. In *Proceedings IEEE International Conference on Field-Programmable Technology (FPT'03)*, page to appear, December 2003.

[6] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 2001.

[7] Andreas Schweizer. Reconfigurable computing auf einem DIMM modul. Master's thesis, ETH Zurich, Computer Engineering and Networks Lab, March 2003. DA-2003.10.

[8] K.K. Ting, S.C.L. Yuen, K.H. Lee, and P.H.W Leong. An fpga based SHA-256 processor. In *Proceedings of the International Workshop on Field Programmable Logic and Applications (FPL)*, pages 577–585, 2002.

[9] Xilinx. *Xilinx Virtex-II 1.5V FPGA Family*, v2.3 edition, October 2002.

A Recovering from an oop's error

If the driver kernel module tries to access an invalid address, the kernel prints an oops error. The module can't be unloaded because its usage counter is different from zero. The application that caused the oops error has crashed and left an unbalanced *open* call.

The module *emergencykm.o* in *pathname/emergency_rmmod* can bring the usage counter to zero. Then *rmmod* can unload the driver. The script *decrease.sh* loads and unloads the emergency module. This operation decreases the usage counter of the driver (works only when the driver was built with DEBUG=yes).

This is a brutal hack, but can save the PC from a reboot.

B Benchmark

As benchmark the TKDM library memberfunction *SdramTest* is used. It writes 16 MB to each SDRAM block and read the data back. The member function is called by a minimal application. The time is measured with the Linux command *time*. The driver version is 1.3.0 and the firmware version is 16.

The result is

$$\begin{array}{cc} real & 3.931s \\ user & 2.946s \\ sys & 0.871s \end{array}$$

The measurement shows that the throughput, calculated from the total execution time of the userspace application is

$$\frac{4 \cdot 16\ MB}{3.931s} = 17.07Mbps$$

This is the performance that is available to the user application.

12th March 2004